

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

LI-PIN CHANG, TUNG-YANG CHOU, and LI-CHUN HUANG, National Chiao-Tung University

Multilevel flash memory cells double or even triple storage density, producing affordable solid-state disks for end users. As flash memory endures only limited program-erase cycles, solid-state disks employ wear-leveling methods to prevent any portions of flash memory from being retired prematurely. Modern solid-state disks must consider wear evenness at both block and channel levels. This study first presents a block-level wear-leveling method whose design has two new ideas. First, the proposed method reuses the intelligence available in flash-translation layers so it does not require any new data structures. Second, it adaptively tunes the threshold of block-level wear leveling according to the runtime write pattern. This study further introduces a new channel-level wear-leveling strategy, because block-level wear leveling is confined to a channel, but realistic workloads do not evenly write all channels. The proposed method swaps logical blocks among channels for achieving an eventually-even state of channel lifetimes. A series of trace-driven simulations show that our wear-leveling method outperforms existing approaches in terms of wear evenness and overhead reduction.

Categories and Subject Descriptors: D.4.2 [Operating Systems]: Storage Management—*Garbage collection*; B.3.2 [Memory Structures]: Design Styles—*Mass storage*

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Flash memory, wear leveling, solid-state disks

## ACM Reference Format:

Chang, L.-P., Chou, T.-Y., and Huang, L.-C. 2013. An adaptive, low-cost wear-leveling algorithm for multichannel solid-state disks. *ACM Trans. Embedd. Comput. Syst.* 13, 3, Article 55 (December 2013), 26 pages. DOI: <http://dx.doi.org/10.1145/2539036.2539051>

## 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 [Samsung Electronics 2006], frequent erase

---

This work is in part supported by research grant 98-2221-E-009-157-MY3 from the National Science Council, Taiwan, ROC, and a joint research project with ADATA Technology Co., Ltd.

Authors' addresses: L.-P. Chang, T.-Y. Chou, and L.-C. Huang, Department of Computer Science, National Chiao-Tung University, 1001 University Road, Hsinchu, Taiwan 300, ROC; corresponding author's email: [lpchang@cs.nctu.edu.tw](mailto:lpchang@cs.nctu.edu.tw).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2013 ACM 1539-9087/2013/12-ART55 \$15.00

DOI: <http://dx.doi.org/10.1145/2539036.2539051>

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 multilevel 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 [Samsung Electronics 2008].

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 Chang et al. [2010], 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 becomes 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, algorithm tuning for host-workload adaption and performance definition requires prior knowledge of flash access patterns, online human intervention, or both. Third, 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 and Du [2009] 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. [2007] 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. [2010] introduced a bitmap that indicates whether a flash erase unit is recently erased or not. However, using the recent erase history could blind wear-leveling algorithms, because the recency and frequency of erasing operations on flash erase units are mutually independent.

Existing wear-leveling designs subject wear evenness to tunable threshold parameters [Chang et al. 2010; Chang and Du 2009; Jung et al. 2007; Agrawal et al. 2008]. The system environment in which wear leveling takes place includes many conditions,

---

<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 [Global Unichip Corp. 2009].

such as flash-translation layer designs, flash geometry, and host disk workloads. Existing approaches require human intervention or prior knowledge of the system environment for threshold setting. However, there are problems of using manually tuned threshold. A wear-leveling algorithm may have good performance with a threshold in a system environment, but with the same threshold, it could cause unexpectedly high wear-leveling overhead or unsatisfactory wear evenness in a different system environment.

From a firmware point of view, implementation complexity primarily involves the applicability of wear-leveling algorithms. The dual-pool algorithm [Chang and Du 2009] uses five priority queues of wear information and a caching method to reduce the RAM footprints of these queues. The group-based algorithm [Jung et al. 2007] and the static wear-leveling algorithm [Chang et al. 2010] 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 new wear-leveling design, called the *lazy wear-leveling algorithm*, to tackle the three design challenges previously mentioned. First, this design stores only a RAM-resident counter indicating the average wear of the entire flash, achieving a tiny RAM footprint. Second, even though this algorithm uses a threshold parameter, it adopts an analytical model to estimate the overhead increase ratio with respect to different threshold settings, and then automatically selects a threshold for good balance between wear evenness and overhead. Third, the proposed algorithm utilizes the address-mapping information available in the disk-emulation algorithm, eliminating the need for adding extra data structures for wear leveling.

Modern solid-state disks are equipped 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 [Chang and Kuo 2002; Dirik and Jacob 2009] (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 remapping data among channels. The proposed approach has many benefits, including that (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) A dynamic threshold-adjusting strategy for block wear leveling.
- (3) An algorithm for wear leveling at the channel level.

The rest of this article is organized as follows. Section 2 reviews flash characteristics and prior work on flash translation and wear leveling. Section 3 presents a block-level wear-leveling algorithm, and Section 4 describes an adaptive tuning strategy for this algorithm. Section 5 introduces a strategy for wear leveling at the channel level. Section 6 reports our simulation results. Section 7 concludes.

## 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 2,048 plus 64 bytes. The 2,048-byte portion stores user data, while the 64 bytes are 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 100 K cycles [Samsung Electronics 2006], while this limit is 10 K or less in multilevel-cell flash [Samsung Electronics 2008].

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 [Gupta et al. 2009] better handles random write requests, but requires large mapping structures, making its implementation difficult when flash capacity is high. This article 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 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. Hybrid mapping creates groups of logical blocks and allocates (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 Figure 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. [2008], 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.

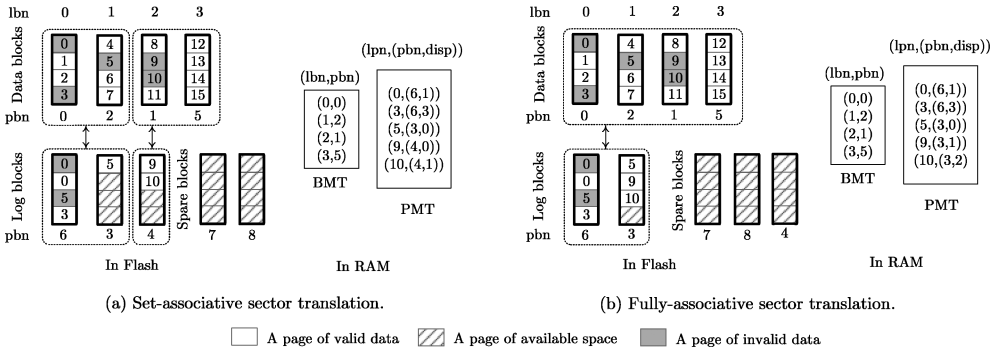


Fig. 1. Two flash-translation layer designs based on hybrid mapping. (a) The set-associative mapping scheme with  $N = 2$  and  $K = 2$ . Every group has two logical blocks, and a group is allocated to up to two log blocks. (b) The fully-associative mapping scheme. All logical blocks are in one big group, and all the log blocks are shared by the logical blocks in this big group.

[2007], 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 sharing all the log blocks.

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 blocks, the FTL collects valid data from the log block and the data block of this logical blocks, copies these valid data to a new spare block, and remaps 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 Figure 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, writes 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.

Hybrid mapping FTLs exhibit some common behaviors in the garbage-collection process regardless of their designs, that is, garbage collection never involves a data block if none of its page data have been updated. In Figure 1(a), erasing the data blocks at *pbn* 5 cannot reclaim any free space. Similarly, in Figure 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.

## 2.2. The Need for Wear Leveling

This section first introduces prior methods, discusses their drawbacks, and then points out how the proposed method 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 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<sup>®</sup> [2008] and Spansion<sup>®</sup> [2008], recommend using this approach. Chang et al. [2010] described a design of static wear leveling. However, Chang and Du [2009] found that 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

Table I. Comparison of Existing Algorithms for Block-Level Wear Leveling

Algorithm	Principle	RAM-resident data structures required	Threshold tuning
Static wear leveling [Chang et al. 2010]	Static wear leveling	A block erase bitmap	Manual
Group wear leveling [Jung et al. 2007]	Hot-cold swapping	Average erase counts of block groups	Manual
Dual-pool wear leveling [Chang and Du 2009]	Cold-data migration	All blocks' erase counts and their recent erase counts	Manual
Remaining-lifetime leveling [Agrawal et al. 2008]	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

large. Hot-cold swapping exchanges data in a lesser-worn block with data from a badly-worn block. Jung et al. [2007] 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 [2009] 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 [Agrawal et al. 2007]. In other words, stopping aging the small amount of badly-worn flash blocks mapped to frequently-updated data is more efficient than starting 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 [Agrawal et al. 2008; Chang and Du 2009]. Based on cold-data migration, Agrawal et al. [2008] proposed storing the remaining lifetimes and data temperatures of all flash blocks in RAM, and Chang and Du [2009] 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 32GB flash-storage device with 512KB 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 which a typical disk controller can afford (64 KB, mentioned in the Introduction). 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 I 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. [2010] adopted the ratio of the total erase count to the total number of the recently erased blocks, Jung et al. [2007] and Chang and Du [2009] used the difference among blocks' erase counts, and Agrawal et al. [2008] 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 [Chang and Du 2009] 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>2</sup> The latter case shows that the same threshold setting resulted in more data movement but did not achieve 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 to channel binding of logical pages, that is, the allocation of free flash pages to host data. Dynamic channel binding globally manages free pages across all channels. Chang and Kuo [2002] proposed dispatching page write requests to channels based on the update frequencies of these page data. Dirik and Jacob [2009] 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 [Agrawal et al. 2008; Park et al. 2010; Seong et al. 2010]. RAID-0 striping achieves the maximum channel-level parallelism in sequential reads 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 Chang [2010], hot data (i.e., frequently updated data) are small, usually between 4 KB and 16 KB. 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 the 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.

<sup>2</sup>These disk workloads were used in our experiments. See Section 6.1.

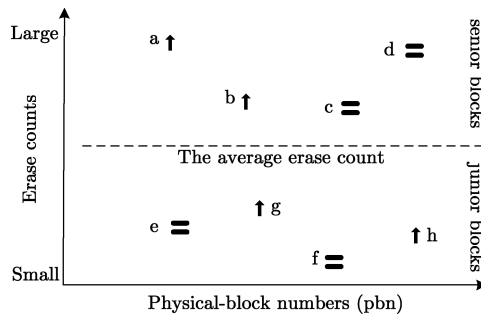


Fig. 2. Physical blocks and their erase recency and erase counts. An upward arrow indicates that a block is recently increasing its erase count.

### 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*.

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 with 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

This study proposes a new wear-leveling algorithm based on a simple principle: whenever a senior block's erase recency becomes high, relocate (i.e., remap) a logical block having a low update recency to this senior block. This algorithm, called the *lazy wear-leveling algorithm*, is named after its passive reaction to excessive flash wear.



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. As previously mentioned in Section 2.2, if a logical block has a low update recency, then garbage collection has no interest in erasing the flash block(s) mapped to it. Therefore, remapping logical blocks of low update recency is a key to preventing senior blocks from aging further. Lazy wear leveling considers logical blocks not related to any page-mapping information as having low update recency, because recent updates to logical blocks leave mapping information in the the page-mapping table. The logical blocks at *lbn* 3 in Figures 1(a) and 1(b) are such examples.

To remap 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, for example, blocks *e* and *f* in Figure 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 remapping 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 remapping.

Temporal localities of writes change occasionally. New updates to a logical block can neutralize the latest remapping 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 remapping operation for this senior block.

### 3.3. 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 pseudocode of lazy wear leveling. 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 remapping whenever necessary and then returns a *pbn*. Note that this output *pbn* may be different from the

---

#### ALGORITHM 1: Lazy Wear-Leveling Algorithm

---

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

**Output:** *p*: a substitute for the original victim block *v*

```

1:  $e_v \leftarrow \text{eraseCount}(v)$ 
2: if  $(e_v - e_{\text{avg}}) > \Delta$  then
3:   repeat
4:      $l \leftarrow \text{lbnNext}()$ 
5:   until  $\text{lbnHasPageMapping}(l) = \text{FALSE}$ 
6:      $\text{erase}(v)$ ;
7:      $p \leftarrow \text{pbn}()$ 
8:      $\text{copy}(v, p)$ ;  $\text{map}(v, l)$ 
9:      $e_v \leftarrow e_v + 1$ 
10:     $e_{\text{avg}} \leftarrow \text{updateAverage}(e_{\text{avg}}, e_v)$ 
11:  else
12:     $p \leftarrow v$ 
13:  end if
14: RETURN p

```

---

input  $pbn$ . The flash-translation layer erases the flash block at the  $pbn$  returned by Algorithm 1. The discussion in this section is based on hybrid mapping. See later sections for using lazy wear leveling with page-level mapping.

For the example of SAST in Figure 1(a), suppose that the flash-translation layer decides to merge data of the logical blocks at  $lbns$  0 and 1. The flash-translation layer calls Algorithm 1 before erasing each of the four physical blocks at  $pbns$  0, 2, 6, and 3. For the example of FAST in Figure 1(b), because FAST recycles the oldest log block at a time, the flash-translation layer calls Algorithm 1 before erasing the log block at  $pbn$  6 and the two related data blocks at  $pbns$  0 and 2. The rest of this section is a detailed explanation of Algorithm 1.

In Algorithm 1, the flash-translation layer provides the subroutines with leading underscores, 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 10 will carry out a remapping operation. Otherwise, Steps 12 and 14 return the victim block  $v$  intact. The loop of Steps 3 through 5 finds a logical block whose update recency is low. Step 4 uses the subroutine  $lbnNext()$  to obtain  $l$  the next logical block number to visit, and Step 5 calls the subroutine  $lbnHasPageMapping()$  to check if the logical block  $l$  has any related mapping information in the page-mapping table. As mentioned previously, to give junior blocks equal chances of getting erased, the subroutine  $lbnNext()$  must evenly visit all logical blocks. At this point, it is reasonable to assume that  $lbnNext()$  produces a linear enumeration of all  $lbns$ .

Steps 6 through 8 remap the previously found logical block  $l$ . Step 6 erases the original victim block  $v$ . Step 7 uses the subroutine  $_pbn()$  to identify the physical block  $p$  that the logical block  $l$  currently maps to. Step 8 copies the data of the logical block  $l$  from the physical block  $p$  to the original victim block  $v$ , and then remaps the logical block  $l$  to the former victim block  $v$  using the subroutine  $map()$ . After this remapping, Step 9 increases  $e_v$  since the former victim block  $v$  has been erased, and Step 10 updates the average erase count. Step 14 returns the physical block  $p$ , which the logical block  $l$  previously mapped to, to the flash-translation layer as a substitute for the original victim block  $v$ . In spite of the average erase count  $e_{avg}$ , Algorithm 1 is only concerned with the erase count of the victim block. Thus, this algorithm needs not store all blocks' erase counts in RAM. Instead, it reads the spare area of a victim block before garbage collection erases it.

### 3.4. Wear-Leveling Enhancements

This section presents two enhancements that lazy wear leveling can use. The first is specific to sequential-write workloads, and the second is particularly useful if the flash-translation layer is FAST.

**3.4.1. Workload-Specific Enhancement.** Algorithm 1 calls  $lbnNext()$  to select logical blocks for remapping. This function can linearly visit all logical blocks. However, this simple strategy could result in many ineffective remapping operations if the host workload consists of a lot of long write bursts. This is because files systems try to allocate contiguous disk space when writing large files. This behavior coincides with linearly enumerating logical blocks and can neutralize prior remapping operations on a set of consecutive logical blocks.

To solve this problem, this study proposes using a Linear Congruential Generator [Rosen 2003] for logical-block selection. Let the total number of logical blocks be  $n_l$ . Let  $p$  be the smallest prime number larger than  $n_l$ . Let  $s$  be an integer and  $0 < s < n_l$ . Let  $l_i$  be the logical-block number produced by the  $i$ th selection, and let  $l_0$  be an

arbitrary number in  $[0, n_l)$ . Lazy wear leveling selects logical blocks using the following recurrence relation.

$$l_{i+1} = (l_i + s) \% p,$$

where  $\%$  is the modulo operator. Notice that any  $l_i \geq n_l$  are not used. Because  $s$  and  $p$  are prime to each other, the period of selecting the same logical-block number is exactly  $n_l$ . Here,  $s$  is the *skip factor*, which should be larger than the total number of logical blocks that typical large files can have. This prevents lazy wear leveling from successively visiting two logical blocks belonging to the same large file. Our current implementation adopts  $s = 1000$  when the logical block size is 128 KB.

The loop in Algorithm 1 (i.e., Steps 3 to 5) checks whether a logical block has related mapping information in the page-mapping table. This check becomes difficult if the flash-translation layer caches a partial mapping table. To address this problem, Algorithm 1 can adopt an optional bitmap  $lbMod[]$  of logical blocks. For any logical block at  $lbn\ l$ ,  $lbMod[l] = 0$  initially, and the flash-translation layer sets  $lbMod[l] = 1$  if a write request modifies any of its logical pages. For example, in Figure 1(a), all bits of this bitmap are 1's except  $lbMod[3]$ . Garbage collection clears  $lbMod[l]$  after erasing the flash blocks related to the logical block at  $lbn\ l$ , because merging this logical block removes all its mapping information from the page-mapping table. With this bitmap,  $lbnHaspageMapping(l)$  at Step 5 reports TRUE if  $lbMod[l] = 1$ , or else reports FALSE.

**3.4.2. FTL-Specific Enhancement.** On garbage collection, FAST erases one log block at a time, that is, the oldest log block. Thus, FAST can delay merging a logical block until a valid logical page of this logical block appears in the oldest log block. Consider that FAST has a very large number of log blocks and the host frequently modifies a logical block. On the one hand, FAST can indefinitely postpone merging this logical block. On the other hand, lazy wear leveling does not use this logical block for remapping because its page updates keep leaving information in the page-mapping table. As a result, the (flash) data blocks mapped to this logical block can never attract attention from both garbage collection and wear leveling.

A simple enhancement based on the bitmap  $lbMod[]$  deals with this problem. When FAST erases the oldest log block, for every piece of page data in this log block, regardless of whether it is valid or not, FAST finds the the logical block number of this logical page and clears the corresponding bit in  $lbMod[]$ , as if FAST did not delay merging logical blocks. Note that SAST does not require this enhancement, because to improve log-block space utilization, SAST will not indefinitely delay merging logical blocks.

### 3.5. Lazy Wear Leveling and Page-Level Mapping

Although lazy wear leveling is primarily designed for hybrid mapping, its concept is applicable to page-level mapping. Like in hybrid mapping, in page-level mapping, lazy wear leveling copies data having low update recency to senior blocks to prevent these blocks from aging further. However, different from hybrid mapping, page-level mapping does not use logical block [Gupta et al. 2009], so lazy wear leveling needs a different strategy to find data having low update recency.

This study proposes using an invalidation bitmap. In this 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, 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. Lazy wear leveling then copies valid data from the flash block owning this zero bit to the victim block. This

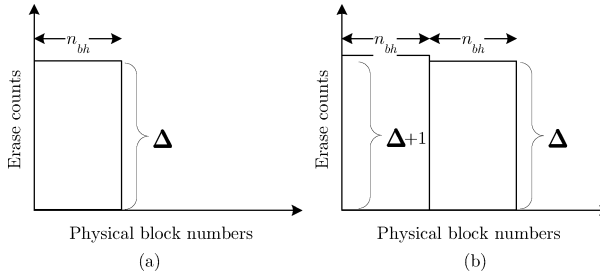


Fig. 3. Erase counts of flash blocks right before the lazy wear-leveling algorithm performs (a) the first remapping operation and (b) the  $n_{bh} + 1$ -th remapping operation.

scan-and-copy procedure repeats until it writes to all pages of the victim 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.

#### 4. SELF TUNING FOR BLOCK-LEVEL WEAR LEVELING

Lazy wear leveling subjects the evenness of block wear to a threshold parameter  $\Delta$ . A small value of  $\Delta$  targets even wear in flash blocks but increases the frequency of data movement. This section presents a dynamic tuning strategy for  $\Delta$  for achieving good balance between wear evenness and overhead.

##### 4.1. Overhead Analysis

Consider a piece of flash memory consisting of  $n_b$  physical blocks. Let immutable logical blocks map to  $n_{bc}$  out of these  $n_b$  physical blocks. Let the sizes of write requests be multiples of the block size, and let write requests be aligned to block boundaries. Suppose that the disk workload uniformly writes the mutable logical blocks. In other words, the flash-translation layer evenly increases the erase counts of the  $n_{bh} = n_b - n_{bc}$  physical blocks.

Let the function  $f(x)$  denote how many blocks garbage collection erases to process a workload that write  $x$  logical blocks. Consider the case  $x = i \times n_{bh} \times \Delta$ , where  $i$  is a nonnegative integer. As all request sizes are multiples of the block size and requests are block-aligned, erasing victim blocks does not cost garbage collection any overhead in copying valid data. Therefore, without wear leveling, we have

$$f(x) = x.$$

Now, consider wear leveling enabled. For ease of presentation, this simulation revises the lazy wear-leveling algorithm slightly: the revised algorithm compares the victim block's erase count against the smallest erase count instead of the average erase count. Figure 3(a) shows that right before lazy wear leveling performs the first remapping, garbage collection has uniformly accumulated  $n_{bh} \times \Delta$  erase counts in  $n_{bh}$  physical blocks. In the subsequent  $n_{bh}$  erase operations, garbage collection erases each of these  $n_{bh}$  physical blocks one more time and increases their erase counts to  $\Delta + 1$ . Thus, lazy wear leveling conducts  $n_{bh}$  remapping operations for these physical blocks at the cost of erasing  $n_{bh}$  blocks. These remapping operations redirect garbage-collection activities to another  $n_{bh}$  physical blocks. After these remapping operations, lazy wear leveling stops until garbage collection accumulates another  $n_{bh} \times \Delta$  erase counts in the new  $n_{bh}$

physical blocks. Figure 3(b) shows that lazy wear leveling is about to spend  $n_{bh}$  erase operations for remapping operations. Now let function  $f'(x)$  be analogous to  $f(x)$ , but with wear leveling enabled. We have

$$f'(x) = x + \left\lfloor \frac{x}{\Delta} \right\rfloor = x + i \times n_{bh}.$$

Under real-life workloads, the frequencies of erasing these  $n_{bh}$  blocks may not be uniform. Thus,  $f'(x)$  adopts a real-number coefficient  $K$  to take this into account:

$$f'(x) = x + i \times n_{bh} \times K.$$

The coefficient  $K$  depends on various conditions of flash management, such as flash geometry, host workloads, and flash-translation layer designs. For example, dynamic changes in temporal localities of write can increase  $K$  because the write pattern might start updating new logical blocks and neutralize the prior remapping operations on these blocks. Notice that the value of  $K$  can be measured at runtime, as will be explained in the next section.

Let the *overhead function*  $g(\Delta)$  denote the *overhead ratio* with respect to  $\Delta$ :

$$g(\Delta) = \frac{f'(x) - f(x)}{f(x)} = \frac{i \times n_{bh} \times K}{i \times n_{bh} \times \Delta} = \frac{K}{\Delta}.$$

It shows that the overhead of wear leveling is inversely proportion to  $\Delta$ . Now recall that lazy wear leveling compares victim blocks' erase counts against the average erase count rather than the smallest erase count. Thus, we use  $2\Delta$  as an approximation of the original  $\Delta$ . Because both  $n_b$  and  $n_{bh}$  are constant, the difference between using the average and the smallest can be accounted for by a constant ratio, which is further included in the runtime-measurable coefficient  $K$ . Thus, we have

$$g(\Delta) = \frac{K}{2\Delta}. \quad (1)$$

When  $\Delta$  is small, a further decrease in  $\Delta$  rapidly increases the overhead ratio. For example, decreasing  $\Delta$  from 4 to 2 doubles the overhead ratio.

## 4.2. A Strategy of Tuning $\Delta$

Small  $\Delta$  values are always preferred in terms of wear evenness. However, decreasing the  $\Delta$  value could cause an unexpectedly large increase in overhead. The rest of this section introduces a  $\Delta$ -tuning strategy based on the overhead growth rates.

Under realistic disk workloads, the coefficient  $K$  in  $g(\Delta)$  may vary over time. Thus, wear leveling must first determine the coefficient  $K$  before using  $g(\Delta)$  for  $\Delta$ -tuning. This study proposes tuning  $\Delta$  on a session-by-session basis. A session refers to a time interval in which lazy wear leveling contributed a predefined number of erase counts. Refer to this number as the session length. The basic idea is to find  $K_{cur}$  of the current session and use this value to find  $\Delta_{next}$  for the next session.

The first session begins with  $\Delta = 16$  (in theory it can be any number). Let  $\Delta_{cur}$  be the  $\Delta$  value of the current session. Figure 4 illustrates the concept of the  $\Delta$ -tuning procedure. During a runtime session, lazy wear leveling separately records the erase counts contributed by garbage collection and wear leveling. At the end of the current session, the first step (in Figure 4) computes the overhead ratio  $\frac{f'(x)-f(x)}{f(x)}$ , that is,  $g(\Delta_{cur})$ , and solves  $K_{cur}$  of the current session using Equation (1), that is,  $K_{cur} = 2\Delta_{cur} \times g(\Delta_{cur})$ .

The second step uses  $g(\Delta_{next}) = K_{cur}/(2\Delta_{next})$  to find  $\Delta_{next}$  for the next session. Basically, lazy wear leveling tries to decrease  $\Delta$  until the growth rate of the overhead ratio becomes equal to a user-defined limit  $\lambda$ . In other words, we are to find the  $\Delta$  value at which the tangent slope to  $g(\Delta_{next})$  is  $\lambda$ . Let the unit of the overhead ratio be one

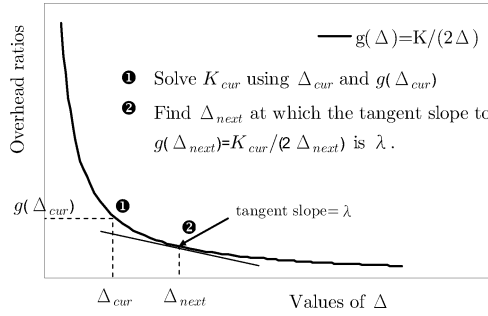


Fig. 4. Computing  $\Delta_{next}$  subject to the overhead growth limit  $\lambda$  for the next session according to  $\Delta_{cur}$  and the overhead ratio  $g(\Delta_{cur})$  of the current session.

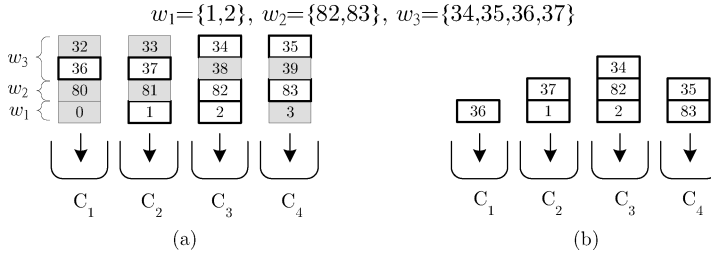


Fig. 5. Handling three write requests  $w_1$ ,  $w_2$ , and  $w_3$  using (a) synchronized channels and (b) independent channels. In this example, using synchronized channels doubles the flash wear, while using independent channels results in unbalanced flash wear among channels.

percent. Therefore,  $\lambda = -0.1$  means that the overhead ratio increases from  $x\%$  to  $(x + 0.1)\%$  when decreasing  $\Delta$  from  $y$  to  $(y - 1)$ . Now solve  $\frac{d}{d\Delta}g(\Delta_{next}) = \frac{\lambda}{100}$  for the smallest  $\Delta$  value subject to  $\lambda$ . Rewriting this equation, we have

$$\Delta_{next} = \sqrt{\frac{100}{-\lambda}} \sqrt{g(\Delta_{cur})\Delta_{cur}}.$$

For example, when  $\lambda = -0.1$ , if the overhead ratio  $g(\Delta_{cur})$  and  $\Delta_{cur}$  of the current session are 2.1% and 16, respectively, then  $\Delta_{next}$  for the next session is  $\sqrt{\frac{100}{0.1}} \sqrt{2.1\% \times 16} = 18.3$ .

The  $\Delta$ -tuning procedure uses the limit on the overhead-ratio growth rates and the session length. Because  $g(\Delta)$  is very large when  $\Delta$  is small,  $\lambda$  can be set to the boundary between near-linear and super-linear growth rates. Our experiments will show that  $-0.1$  is a good choice of  $\lambda$ , and wear-leveling results are not sensitive to the lengths of sessions because workloads have temporal localities of write.

## 5. CHANNEL-LEVEL WEAR LEVELING

### 5.1. Multichannel Architectures

Advanced solid-state disks use multichannel architectures for high data transfer rates [Agrawal et al. 2008; Kang et al. 2007; Seong et al. 2010; Park et al. 2010]. 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 these structures might not be individually programmable.

From the point of view of wear leveling, channels can be *synchronized* or *independent*. Figure 5 is an example. Let the mapping between logical pages and channels use the

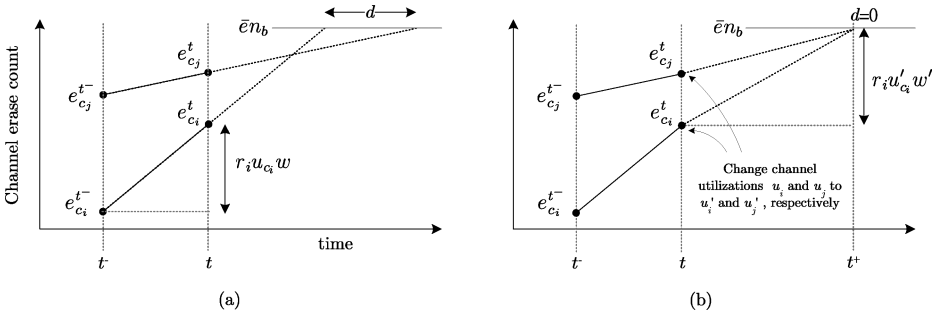


Fig. 6. Aligning the lifetime expectancies of two channels  $C_i$  and  $C_j$  for channel-level wear leveling. (a) These two channels reach their end-of-life at different times. (b) Change channel utilizations  $u_{c_i}$  and  $u_{c_j}$  to  $u'_{c_i}$  and  $u'_{c_j}$ , respectively, such that the lifetime difference becomes zero (i.e.,  $d = 0$ ).

RAID-0 style striping. Figure 5(a) depicts that all the channels write synchronously, even if a write request does 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 syncing channel operations is the reduced device lifetime. As Figure 5(a) shows, the channels writes 16 flash pages to modify only eight logical pages. Independent channels need not copy unmodified data for syncing channel operations, as shown in Figure 5(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 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* [Shang et al. 2011] 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 [Park et al. 2010].

### 5.2. 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 6 is an example of two channels  $C_i$  and  $C_j$ . Let every channel have the same total number of flash blocks  $n_b$ . Let a flash block endure  $\bar{e}$  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 reaches 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  $u_{c_i} \leq 1$

Table II. Symbol Definitions

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 $\sum_{j=0}^{y-1} u_{i,j} = u_{s_i}$

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 Figure 6(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 II 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 time  $t$ , respectively. Replacing  $u_{c_i}$ ,  $u_{c_j}$ , and  $d$  in the preceding equation 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 6(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.

### 5.3. 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.

Because real workloads have temporal localities of write, swapping logical blocks among channels can manipulate channels' future utilizations. To retain 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 stripe  $i$  and 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  $\sum_{i=0}^{x-1} |u_{c_i} - u'_{c_i}|$ . 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 nonzero utilization  $s$ .

Channel wear leveling should reduce the total number of logical blocks swapped. We found that in real workloads, a stripe of 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 [Chang 2010]. 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 predefined limitation. Figure 7 is a numeric example of channel wear leveling. In this example, the channel lifetime limit  $\bar{e}n_b$  is 10,000. Figure 7(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 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.

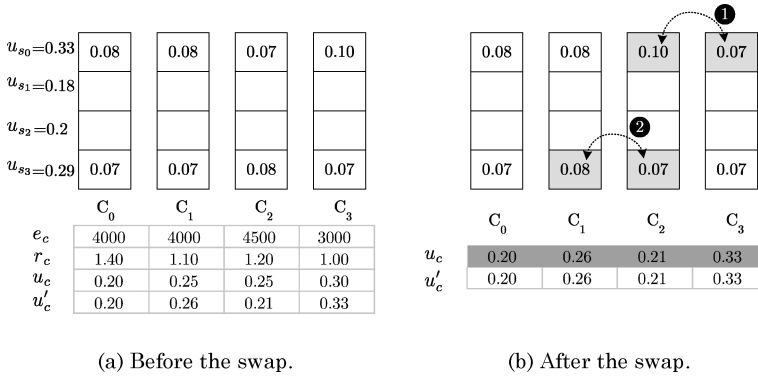


Fig. 7. Swapping logical blocks among channels for channel wear leveling.

Table III. Characteristics of the Experimental Disk Workloads

Workload	Operating System	File System	Logical Disk Size	Total Written	Avg. Req. Size	Disk Coverage <sup>†</sup>	Disk Coverage <sup>‡</sup>
PC	Windows XP	NTFS	40 GB	81.2 GB	11.5 KB	41.57%	48.54%
PM	Windows 7	NTFS	40 GB	43.3 GB	11.8 KB	2.93%	6.31%
MM	Windows CE	FAT32	20 GB	19.8 GB	59.6 KB	87.25%	87.26%
RND	Ubuntu 9	Ext4	16 GB	18.6 GB	4 KB	68.56%	99.61%

<sup>†</sup> fractions of disk space written during workload generation (in terms of 512 B sectors).

<sup>‡</sup> fractions of disk space written during workload generation (in terms of 512 KB logical blocks).

Figure 7(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 for effective channel wear leveling.

## 6. PERFORMANCE EVALUATION

### 6.1. Experimental Setup and Performance Metrics

We built a simulator and implemented various wear-leveling algorithms and flash-translation layers for evaluation. The simulator provides three options of the flash-translation layer: SAST [Park et al. 2008], FAST [Lee et al. 2007], and DFTL [Gupta et al. 2009]. The former two are representative designs of hybrid mapping, while the last one uses page-level mapping. The simulator also implements the proposed lazy wear leveling, static wear leveling [Chang et al. 2010], and dual-pool wear leveling [Chang and Du 2009]. Static wear leveling is widely used in the industry [Micron<sup>®</sup> 2008; Spansion<sup>®</sup> 2008], while dual-pool wear leveling delivers better performance [Chang and Du 2009].

Our experiments adopted four types of disk workloads, PC, PM, MM, and RND (see Table III). The PC workload was collected from a 40GB 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. The PM workload was produced by a Windows desktop running Postmark 1.5 benchmark [Katcher 1997] with the default settings except that the total number of transactions was set to 2,800,000. This workload has intensive activities of creating/writing/deleting small files. The MM workload was captured from a memory card of a Windows Mobile device. This workload repeatedly copied/deleted MP3 and video files to/from a 20GB memory card formatted

Table IV. Evaluation Results of Lazy Wear Leveling (LWL), Static Wear Leveling (SWL), and Dual-Pool Wear Leveling (DP) under the PC, MM, and RND Workloads. "no WL" Stands for not using Wear Leveling

Workload	Algorithm	largest EC	smallest EC	mean	STDDEV	Threshold	Stable
PC	no WL	939	0	270.1	283.1	—	no
	LWL	298	151	278.4	11.4	16	yes
	SWL	586	50	278.7	64.3	14	no
	DP	470	244	279.3	19.3	16	yes
PM	no WL	3960	0	270.3	885.5	—	no
	LWL	297	253	277.2	10.1	16	yes
	SWL	973	42	278.2	38.4	30	no
	DP	814	243	278.7	13.4	28	yes
MM	no WL	388	0	252.7	96.6	—	no
	LWL	299	198	260.3	11.4	16	yes
	SWL	338	195	259.8	17	4	no
	DP	338	227	254.8	6	14	no
RND	no WL	6746	0	6639.5	408.8	—	no
	LWL	6729	6108	6717.7	31.4	16	yes
	SWL	6743	6316	6663.7	38.4	2	no
	DP	6757	6661	6668.3	8.6	6	no

in FAT-32. This workload has many long write bursts. The RND workload was collected from a Linux box running Iometer [Open Source Development Lab 2003] on a 16GB hard drive formatted in ext4. The settings of Iometer were 100% random write with 4KB write requests.

This study uses the standard deviation of all flash blocks' erase counts to indicate the evenness of flash wear. The smaller the standard deviation, the more even the flash wear will be. This study also considers the mean (i.e., the arithmetic average) of all erase counts. The 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 4 KB and 512 KB, respectively. This is a typical MLC-flash geometry [Samsung Electronics 2008]. 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  $40 \text{ GB} * 1.025 = 41 \text{ GB}$ . 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.

## 6.2. Experimental Results: Block-Level Wear Leveling

**6.2.1. Lazy Wear Leveling vs. Existing Approaches.** This part of the experiment compares lazy wear leveling against static wear leveling and dual-pool wear leveling under the three disk workloads. These three wear-leveling algorithms have different definitions of their thresholds. For fair comparison, this experiment fixed  $\Delta$  of lazy wear leveling at 16, and adjusted the other two algorithms' thresholds to align their final erase-count means to that of lazy wear leveling. This experiment also 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 IV shows the experimental results. First, compare the results of using lazy wear leveling and the results of not using wear leveling at all. The standard deviations

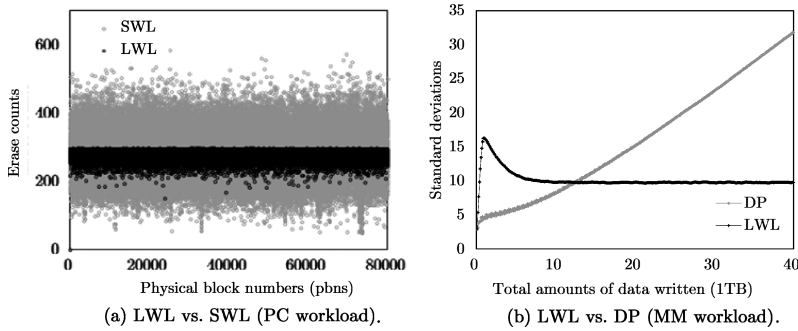


Fig. 8. (a) The final erase-count distributions of lazy wear leveling and static wear leveling under the PC workload (after writing 4 TB of data). (b) The runtime standard deviations of lazy wear leveling and dual-pool wear leveling under the MM workload.

of the PC workload is very large without wear leveling, and lazy wear leveling reduced the standard deviation by 96% (from 283 to 11), while increasing the mean by only 2.9% (from 270 to 278). This is because lazy wear leveling is very effective in the presence of temporal localities of write. Lazy wear leveling was even more successful under the PM workload, and reduced the standard deviation by 99% (from 886 to 10). This is because the PM workload confines the write traffic to only 6.3% of the entire disk space, and thus its temporal locality is better than that of the PC workload. Compared to the PC and PM workloads, the MM workload has a relatively small standard deviation without wear leveling. This is because the MM workload has many sequential and long write bursts. Lazy wear leveling is still useful in this case, reducing the standard deviation from 96 to 11. The RND workload has the largest standard deviation without wear leveling. Even though the write pattern of the RND workload is uniformly random, the extremely high garbage-collection overhead under this workload exaggerated the imbalance in flash wear. With lazy wear leveling, the standard deviation decreased from 408 to 31.

Next, focus on the comparison among different wear-leveling algorithms. Lazy wear leveling outperformed static wear leveling in terms of wear evenness in all cases. Interestingly, static wear leveling was unstable under all workloads. Figure 8(a) shows that under the PC workload, the final erase-count distribution of static wear leveling is more imbalanced than that of lazy wear leveling. A closer inspection of static wear leveling's behaviors revealed two causes of this performance difference. First, static wear leveling moves static data from a block to another, regardless of whether the target block is junior or senior. Under the PC workload, there was a 70% probability that static wear leveling would move data from a static block to a junior block. 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 remaps data to a junior block nor allows the flash-translation layer to write new data to senior blocks.

Results in Table IV indicate that dual-pool wear leveling was unstable under the MM and RND workload, while lazy wear leveling was stable. Figure 8(b) shows that the standard deviation of dual-pool wear leveling became worse than that of lazy wear leveling after the total amount of data written achieved was 13 TB. This is because flash blocks of the same wear information (either the same erase count or the same recent erase count) appear first-in first-out in the priority queues of dual-pool wear leveling. Thus, under the MM workload, writing large files can neutralize the prior efforts of wear leveling on a number of flash blocks (as mentioned in Section 3.4.1).

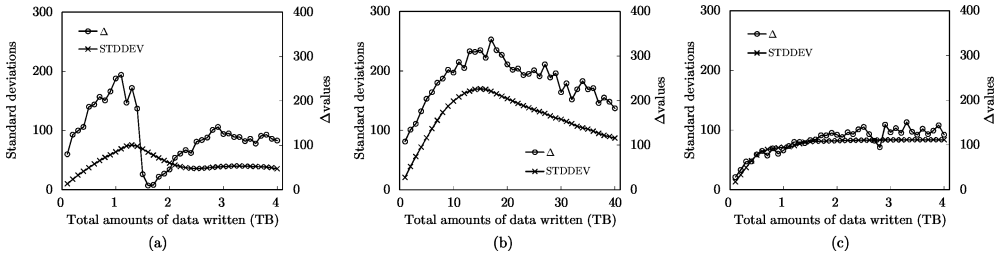


Fig. 9. Runtime  $\Delta$  values and standard deviations using the proposed dynamic  $\Delta$ -tuning method under (a) the PC workload, (b) the MM workload, and (c) the RND workload. The X-axes indicate the total amounts of host data written to the flash-translation layer.

Under the RND workload, a not-recently-updated logical block has a better chance of being updated, and thus this behavior coincides with the first-in first-out order in the priority queues. Lazy wear leveling avoids this problem using a nonlinear block selection policy. Even though dual-pool wear leveling is unstable, Table IV shows that its overhead is smaller than that of lazy wear leveling. This is because after performing data movement among blocks, dual-pool wear leveling hides these blocks for a while to see whether this data movement is effective in terms of wear leveling. This protection decreases the frequency of wear leveling operations and avoids some unnecessary data movement. Contrarily, lazy wear leveling selects not-recently-updated logical blocks for remapping, but under the random write pattern, these logical blocks have better chances to get updated in the near future.

Now focus on the space overhead of the three algorithms in terms of RAM footprints and flash space requirements. Let  $n_b$  and  $n'_b$  be the total number of physical blocks and logical blocks, respectively. Note that  $n_b > n'_b$ . Suppose that storing an erase count uses  $k$  bits. For RAM footprints, static wear leveling requires a block-erase bitmap of  $n_b$  bits. Dual-pool wear leveling uses  $kn_b$  bits to store all blocks' erase counts in RAM. It also requires five bit pyramids, each of which uses  $n_b - 1$  bits. Thus, its entire RAM footprint is  $kn_b + 5(n_b - 1)$  bits. Lazy wear leveling uses only  $k$  bits to store an average erase count in RAM. Adopting the optional bitmap  $lbMod[]$  requires an extra  $n'_b$  bits. Consider the experimental settings under the PC workload, we have  $n'_b = 81,920$  and  $n_b = 83,968$ . Let  $k$  be 16. From the previous discussion, the RAM footprints of dual-pool wear leveling, static wear leveling, and lazy wear leveling are 215 KB, 10.25 KB, and 16 bytes (plus 10 KB for the optional bitmap  $lbMod[]$ ), respectively. For flash space requirements, dual-pool wear leveling requires dedicated flash blocks for storing erase counts. Lazy wear leveling stores erase counts in page spare areas, so effectively, it does not cost extra flash pages. We had successfully implemented lazy wear leveling in a real solid-state disk. Interested readers are referred to Chang and Huang [2011].

**6.2.2. Dynamic  $\Delta$ -Tuning for Lazy Wear Leveling.** This experiment tested the proposed  $\Delta$ -tuning method under the three workloads. The session length for  $\Delta$ -tuning was 200, so  $\Delta$  adjusts after lazy wear leveling erased every 200 blocks. The value of  $\lambda$  was  $-0.1$ . Figure 9 depicts the runtime values of  $\Delta$  and standard deviations during this experiment. The X-axes denote the total amounts of host data written to the flash-translation layer. These results show useful insights into how different types of workloads require wear leveling: Figure 9(a) shows that under the PC workload,  $\Delta$  and the standard deviation were becoming stable after the workload produced about 1.2 TB of data. At this time, the last flash block whose erase count was zero started contributing erase cycles. Afterward, every flash block had been involved in wear leveling, and  $\Delta$  and the standard deviation steadily remained at around 80 and 50, respectively.

Table V. Experimental Results of Using Lazy Wear Leveling (LWL) and Static Wear Leveling (SWL) with Different Flash-Translation Layer Designs (i.e., SAST, FAST, and DFTL) under the PC Workload

FTL algorithm	WL algorithm	Threshold	Mean	STDDEV	Stable
SAST (hybrid mapping)	no WL	—	268.5	280.1	no
	LWL	16	279.1	10.8	yes
	SWL	14	279.6	59.2	no
FAST (hybrid mapping)	no WL	—	270.1	283.1	no
	LWL	16	278.4	11.4	yes
	SWL	14	278.7	64.3	no
DFTL (page-level mapping)	no WL	—	250	461.9	no
	LWL	16	230	29.7	yes
	SWL	18	228	70	no

Figure 9(b) shows that, surprisingly, lazy wear leveling refrained from using small  $\Delta$  values under the MM workload. Small thresholds ought to be good choices, because the standard deviation without wear leveling under the MM workload was not large, as Table IV shows. However, the MM workload has few temporal localities which are essential to the success of wear leveling. Thus, aggressive wear leveling unexpectedly resulted in a high overhead. Figure 9(c) shows that under the RND workload, the  $\Delta$  values and the standard deviations were at around 100 and 110, respectively, and barely varied. This is because the update frequencies of logical blocks never change (as the RND workload is purely random), and thus the cost of wear leveling with respect to the same  $\Delta$  value almost remained constant in sessions.

These results suggest that wear leveling is very useful to the PC workload, because this kind of workload has temporal localities of write. In contrast, wear leveling under the sequential-write workloads, such as the MM workload, is not as easy as we thought. Aggressive wear leveling could incur an unexpectedly high overhead.

This experiment also tested different settings of  $\lambda$  and the session length under the PC workload. When the session lengths were 200, 400, and 800, the final standard deviations were 47.1, 47.5, and 45.8, respectively. The overhead ratios (defined in Section 4.1) of using these three session lengths were 0.84%, 0.86%, and 0.94%, respectively. These results are not sensitive to different session lengths. When the values of  $\lambda$  were  $-0.1$  and  $-0.3$ , the standard deviations were 47.1 and 32.5, and the overhead ratios were 0.84% and 1.4%, respectively. The improvement upon standard deviation of using  $\lambda$  smaller than  $-0.1$  seems not worth the large overhead increase.

**6.2.3. Lazy Wear Leveling with Different FTL Algorithms.** Table V presents the results of evaluating lazy wear leveling and static wear leveling with FAST, SAST, and DFTL [Gupta et al. 2009]. SAST adopted  $N = 16$  and  $K = 32$  as its best settings.

When wear leveling was disabled, the large standard deviations show that flash wear was severely imbalanced under SAST, FAST, and DFTL. Compared to static wear leveling, lazy wear leveling achieved smaller standard deviations in this experiment. Additionally, lazy wear leveling was stable with the three flash-translation algorithms while static wear leveling was not. Results also show that in DFTL, using wear leveling even achieved lower erase-count means compared to not using wear leveling, because lazy wear leveling and static wear leveling can cluster infrequently updated data in flash blocks, and this behavior benefits garbage collection for page-level mapping in terms of overhead reduction.

**6.2.4. Lazy Wear Leveling with Different Over-Provisioning Ratios.** Figure 10 shows the results of using different over-provisioning ratios of flash space, ranging from 2.5% to 50%. First, focus on the block wear evenness shown in Figure 10(a). Lazy wear leveling

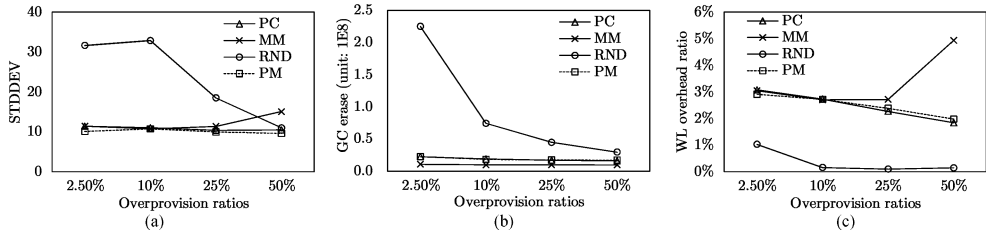


Fig. 10. Results of varying the over-provisioning ratios of flash space (a) the block wear evenness, (b) the overhead of garbage collection, and (c) the overhead ratio (as defined in Section 4.1) of wear leveling.

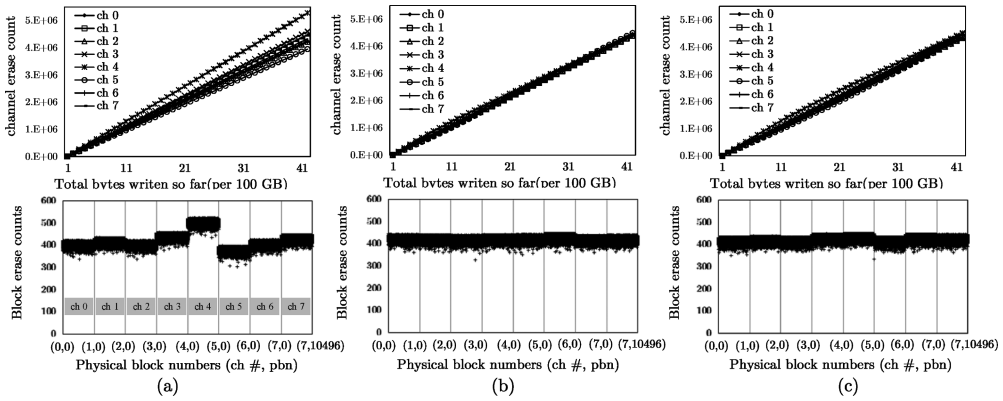


Fig. 11. Runtime channel erase counts and final distributions of block erase counts under the PC workload. (a) Not using channel-level wear leveling, (b) using channel wear leveling with a period length of 512 MB, and (c) using channel wear leveling with a period length of 1 TB.

achieved better wear evenness with large over-provisioning ratios. This is because, as shown in Figure 10(b), increasing the ratio will reduce the intensity of garbage-collection activities and thus alleviate the imbalance among block wear. Consequently, the overhead of wear leveling reduced accordingly, as shown in Figure 10(c). But in Figure 10, the results under the MM workload are an exception. To explain why, first notice that the disk coverage of the MM workload is 87% (as Table III indicates). Thus, the amount of senior physical blocks will be larger than that of infrequently updated logical blocks. Because lazy wear leveling insists on performing remapping for every senior block, in this case, it will incur extra overhead. Nevertheless, lazy wear leveling can gracefully cope with this problem using dynamic threshold adjustment. As pointed out in Section 6.2.2, lazy wear leveling backs off from aggressively intervening in block wear under the MM workload, because the large disk coverage of the MM workload automatically achieves wear leveling to a certain degree. Like the MM workload, the RND workload also has an almost full disk coverage (99%), and using large over-provisioning ratio will also increase the overhead of wear leveling. However, the RND workload requires little wear leveling, and Figure 10(c) shows that the increase in wear-leveling overhead is not noticeable.

### 6.3. Experimental Results: Channel-Level Wear Leveling

This part of the 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 6.2.2.

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 Figure 11(a) shows that without channel-level wear leveling, channels 4 and 5 were the most-worn and the least-worn channels, respectively. The bottom half of Figure 11(a) also indicates that even though the block wear in every channel was even, the erase counts of flash blocks from 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 Figures 11(b) and 11(c), channel wear leveling was invoked after the workload produced every 512 MB and 1 TB of data, respectively, and channel wear leveling could swap up to 50 logical blocks for each invocation. The upper halves of Figures 11(b) and 11(c) show that the channel-erase counts gradually converged at the end-of-life of channels, that is,  $\bar{e} \times n_b = 450 \times 10,752 = 4,838,400$ . This convergence in Figure 11(b) was faster than that in Figure 11(c), because in Figure 11(b), channel wear leveling was invoked more often. In the bottom halves of Figures 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.

## 7. CONCLUSION

This study tackles three problems of wear leveling: block-level wear leveling, adaptive tuning for block wear leveling, and channel-level wear leveling. Block-level wear leveling monitors the wear of all flash blocks and intervenes when block wear develops imbalanced. The tuning of block-level wear leveling seeks good balance between wear evenness and overhead under various workloads. Channel-level wear leveling aims at even channel lifetimes for maximizing the device-level lifespan.

This study presents lazy wear leveling for block-level wear leveling. Lazy wear leveling prevents senior blocks from further aging by moving infrequently updated data to these senior blocks. We found its implementation can be very simple based on two observations. First, flash blocks increase their erase counts via garbage collection on only. Thus, lazy wear leveling can identify senior blocks whenever garbage collection is about to erase a victim. Second, frequently updated logical blocks will leave mapping information in the page-mapping table, so lazy wear leveling can find these infrequently updated data by checking the mapping table. Lazy wear leveling subjects block-wear evenness to a threshold, and using the same threshold value may produce different costs and wear-evenness under various workloads. This study derives the overhead as a function of the threshold and proposes decreasing the threshold until the overhead can significantly increase. Our results show that wear level should refrain from using small thresholds for sequential and random workloads.

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 that its overhead is nearly negligible.

A recent study [Balakrishnan et al. 2010] suggests that SSDs in RAID configurations 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

- AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. 2007. A five-year study of file-system metadata. *Trans. Storage* 3.
- AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference on Annual Technical Conference (ATC'08)*. USENIX Association, 57–70.
- BALAKRISHNAN, M., KADAV, A., PRABHAKARAN, V., AND MALKHI, D. 2010. Differential raid: Rethinking raid for SSD reliability. *Trans. Storage* 6, 2, 4:1–4:22.
- CHANG, L.-P. 2010. A hybrid approach to NAND-flash-based solid-state disks. *IEEE Trans. Comput.* 59, 10, 1337–1349.
- CHANG, L.-P. AND DU, C.-D. 2009. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.* 15, 1, 1–36.
- CHANG, L.-P. AND HUANG, L.-C. 2011. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'11)*. ACM, New York, NY, 31–40.
- CHANG, L.-P. AND KUO, T.-W. 2002. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. 187–196.
- CHANG, Y.-H., HSIEH, J.-W., AND KUO, T.-W. 2010. Improving flash wear-leveling by proactively moving static data. *IEEE Trans. Comput.* 59, 1, 53–65.
- DIRIK, C. AND JACOB, B. 2009. 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)*. ACM, New York, NY, 279–289.
- GLOBAL UNICHIP CORP. 2009. GP5086 Datasheet. <http://www.globalunichip.com/4-10.php>.
- GUPTA, A., KIM, Y., AND URGONKAR, B. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, 229–240.
- JUNG, D., CHAE, Y.-H., JO, H., KIM, J.-S., AND LEE, J. 2007. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. ACM, 160–164.
- KANG, J.-U., KIM, J.-S., PARK, C., PARK, H., AND LEE, J. 2007. A multi-channel architecture for high-performance NAND flash-based storage system. *J. Syst. Archit.* 53, 9, 644–658.
- KATCHER, J. 1997. Postmark: A new file system benchmark. Tech. rep., TR3022, Network Appliance.
- LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *Trans. Embed. Comput. Syst.* 6, 3, 18.
- MICRON®. 2008. Wear-Leveling Techniques in NAND Flash Devices. Micron Application Note (TN-29-42).
- OPEN SOURCE DEVELOPMENT LAB. 2003. Iometer. <http://http://www.iometer.org/>.
- PARK, C., CHEON, W., KANG, J., ROH, K., CHO, W., AND KIM, J.-S. 2008. A reconfigurable FTL architecture for NAND flash-based applications. *ACM Trans. Embed. Comput. Syst.* 7, 4, 1–23.
- PARK, S.-H., PARK, J.-W., KIM, S.-D., AND WEEMS, C. C. 2010. A pattern adaptive NAND flash memory storage structure. *IEEE Trans. Comput.* 99.
- ROSEN, K. 2003. *Discrete Mathematics and Its Applications*. McGraw-Hill, New York.
- SAMSUNG ELECTRONICS. 2006. K9F8G08B0M 1Gb \* 8 Bit SLC NAND Flash Memory. Data sheet.
- SAMSUNG ELECTRONICS. 2008. K9MDG08U5M 4G \* 8 Bit MLC NAND Flash Memory. Data sheet.

- SEONG, Y. J., NAM, E. H., YOON, J. H., KIM, H., CHOI, J.-Y., LEE, S., BAE, Y. H., LEE, J., CHO, Y., AND MIN, S. L. 2010. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Trans. Comput.* 59, 905–921.
- SHANG, P., WANG, J., ZHU, H., AND GU, P. 2011. A new placement-ideal layout for multiway replication storage system. *IEEE Trans. Comput.* 60, 8, 1142–1156.
- SPANSION<sup>®</sup>. 2008. Wear Leveling. Spansion Application Note (AN01).

Received October 2011; revised February, June 2012; accepted December 2012