

# Chapter 10

## Efficient Wear Leveling in NAND Flash Memory



Yuan-Hao Chang and Li-Pin Chang

**Abstract** In the recent years, flash storage devices such as solid-state drives (SSDs) and flash cards have become a popular choice for the replacement of hard disk drives, especially in the applications of mobile computing devices and consumer electronics. However, the physical constraints of flash memory pose a lifetime limitation on these storage devices. New technologies for ultra-high density flash memory such as multilevel-cell (MLC) flash further degrade flash endurance and worsen this lifetime concern. As a result, flash storage devices may experience a unexpectedly short lifespan, especially when accessing these devices with high frequencies. In order to enhance the endurance of flash storage device, various wear leveling algorithms are proposed to evenly erase blocks of the flash memory so as to prevent wearing out any block excessively. In this chapter, various existing wear leveling algorithms are investigated to point out their design issues and potential problems. Based on this investigation, two efficient wear leveling algorithms (i.e., the evenness-aware algorithm and dual-pool algorithm) are presented to solve the problems of the existing algorithms with the considerations of the limited computing power and memory space in flash storage devices. The evenness-aware algorithm maintains a bit array to keep track of the distribution of block erases to prevent any cold data from staying in any block for a long period of time. The dual-pool algorithm maintains one hot pool and one cold pool to maintain the blocks that store hot data and cold data, respectively, and the excessively erased blocks in the hot pool are exchanged with the rarely erased blocks in the cold pool to prevent any block from being erased excessively. In this chapter, a series of explanations and analyses shows that these two wear leveling algorithms could evenly distribute block erases to the whole flash memory to enhance the endurance of flash memory.

---

Y.-H. Chang (✉)

Academia Sinica, Institute of Information Science, Taipei, Taiwan

e-mail: johnson@iis.sinica.edu.tw

L.-P. Chang

Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan

e-mail: lpchang@cs.nctu.edu.tw

© Springer Nature Singapore Pte Ltd. 2018

R. Micheloni et al. (eds.), *Inside Solid State Drives (SSDs)*,

Springer Series in Advanced Microelectronics 37,

[https://doi.org/10.1007/978-981-13-0599-3\\_10](https://doi.org/10.1007/978-981-13-0599-3_10)

## 10.1 Introduction

NAND flash memory has been widely adopted in various mobile embedded applications, due to its non-volatility, shock-resistance, low-power consumption, and low cost. It is widely adopted in various storage systems, and its applications have grown much beyond its original designs. The two popular NAND flash memory designs are single-level-cell (SLC) flash memory and multi-level-cell (MLC) flash memory. Each SLC flash-memory cell can accommodate 1-bit information while each  $\text{MLC}_{\times n}$  flash-memory cell can contain  $n$ -bit information. As  $n$  increases, the endurance of each block in MLC flash memory decreases substantially.<sup>1</sup> In recent years, Well-known examples are flash-memory cache of hard drives (known as TurboMemory) [13, 40, 48], fast booting devices (for Microsoft Windows Vista), and solid-state disks (SSD) (for the replacement of hard drives).

As the low-cost MLC flash-memory designs are gaining market momentum [11], the endurance of flash memory is an even more challenging problem. For example, the endurance of an  $\text{MLC}_{\times 2}$  flash-memory block is only 10,000 (or 5,000) erase cycles whereas that of its SLC flash memory counterpart is 100,000 erase cycles [35, 41]. As the number of bits of information per cell would keep increasing for MLC in the near future, the endurance of a block might also get worse, such as few thousand or even hundred erase cycles. This underlines the endurance issue of flash memory. However, improving endurance is problematic because flash-memory designs allow little compromise between system performance and cost, especially for low-cost flash storage devices. Such developments reveal the limitations of flash memory, especially in terms of endurance.

A NAND flash storage device or storage system, e.g., a solid-state disk (SSD) and flash cards, may be associated with multiple chips. Each chip is composed of one or more sub-chips or dies. Each sub-chip might have multiple planes. Each plane is organized in terms of blocks that are the basic unit for erase operations. A block is further divided into a fixed number of pages and can only endure limited erase cycles. A page (that is the unit of read and write operations) consists of a user area and a spare area, where the user area is for data storage, and the spare area stores house-keeping information such as the corresponding logical block addresses (LBAs), status flags, and error correction codes (ECCs). When a page is written with data, it is no longer available unless it is erased. This is called the “write-once property”. As a result, “out-place updates” are adopted so that data are usually updated over free pages. Pages that contain the latest copy of data (i.e., valid data) are considered as live (or valid) pages, and pages with old versions (i.e., invalid data) are dead (or invalid) pages. Therefore, address translation is needed to map logical addresses of data to their physical addresses, and “garbage collection” is needed to reclaim dead pages. Because each block has a limited number of erase

---

<sup>1</sup>In this chapter, we consider NAND flash memory, which is the most widely adopted flash memory in storage-system designs.

cycles, “wear-leveling” is needed to evenly erase blocks so as to prevent wearing out some blocks excessively.

Engineers and researchers have recently become concerned with how long flash storage devices can withstand daily use when they are adopted in applications with high access frequencies. The host systems, e.g., smart phones and notebooks, access their secondary storages (such as hard drives and SSDs) with temporal localities [6, 32, 33, 46]. Frequently updated data and rarely updated data coexist under such workloads. When reclaiming free space, block erases are always directed to the blocks with few valid data so as to reduce data-copy overheads. Thus, blocks having many static (or immutable) data are rarely chosen for erases, while other blocks are erased many times to circulate frequently updated data. As a result, some blocks are worn out when other blocks remain fresh. The problem of wearing out blocks is a crucial concern for new-generation flash memory, and *wear leveling* is the policy of evenly erasing all flash-memory blocks to keep all the blocks alive as long as possible. Strategies friendly to wear leveling can be adopted in various system layers, including applications, file systems, and firmware. To closely monitor wear in all blocks, the flash management strategies that are usually implemented as firmware implements wear leveling. However, wear leveling is not free, since extra data movement is required. Alleviating wear-leveling overheads is an important task, as wear leveling activities themselves wear flash memory too.

Many excellent wear leveling algorithms have been proposed by academia and industry. Updating data out of place is a simple wear-leveling technique [12, 23, 29, 31, 38]. However, this simple policy is vulnerable in the presence of static data because static data are rarely invalidated and need to be copied out before their residing blocks are erased. In order to reduce live-data-copying overhead, blocks storing a lot of static data rarely participate in the activities of reclaiming free space. Therefore, the key to wear leveling may be to encourage the blocks with static data to participate in block erases. Kim and Lee [20] and Chiang et al. [9] proposed value functions for choosing victim blocks. In their approach, a block receives a high score if it currently has few valid data or its number of accumulated erase cycles is low. Another technique is to erase blocks in favor of reclaiming free space most of the time, but periodically, a block is erased in favor of wear leveling [24, 47]. A typical strategy is to occasionally erase a random block. Wear leveling activities can also be completely detached from free-space reclaiming. Hot-cold swapping [6, 10, 17, 20, 27] involves swapping data in a frequently erased block with that in an infrequently erased block whenever the wear of all blocks is unbalanced.

These existing approaches share a common idea: encouraging infrequently erased blocks to contribute to erases cycles. Under the workload of most real access patterns, most block erases are contributed by a small fraction of blocks if wear leveling is not used. According to such observations, static wear leveling algorithms are proposed to move static data away from infrequently erased blocks [2, 7, 43]. However, some existing static wear leveling algorithms don’t consider the limited computing power or restricted RAM space, while some don’t consider the access patterns and data access frequencies [3, 4, 18, 39, 42]. As a result, these existing static wear leveling algorithms either consume too many hardware resources or

introduce too many overheads on extra live page copies and block erases. In order to achieve static wear leveling effectively with limited computing power, limited main memory, and limited overheads, two efficient wear leveling algorithms (i.e., the evenness-aware algorithm and dual-pool algorithm) are proposed and presented in this chapter. The *evenness-aware algorithm* [8] maintains a house-keeping data structure, i.e., a bit array, with a cyclic-queue-based scanning procedure to keep track of the distribution of block erases to prevent any static or cold data staying in any block for a long period of time. The objective is to improve the endurance of flash memory with limited overhead and without excessively modifying popular implementations of flash management designs, such as FTL, NFTL, and BL [1, 14, 16, 45]. The *dual-pool algorithm* [5] maintains one hot pool and one cold pool to maintain the blocks that store hot data and cold data, respectively, and the excessively erased blocks in the hot pool are exchanged with the rarely erased blocks in the cold pool to prevent any block being erased excessively. Whenever a block is excessively erased, it is filled with static data. In this way, such blocks stop participating in free-space reclaiming. This strategy helps conserve data movement because the major contributors of block erases are only a small fraction of all blocks. Second, blocks recently involved in wear leveling should be temporarily isolated from wear leveling activities. For example, after static data are written to a block which has been erased many times, the dual-pool algorithm decides how long this block should wait before it can contribute more erase cycles.

The rest of this paper is organized as follows: Sect. 10.2 presents the evenness-aware algorithm with the worst-case analysis. In Sect. 10.3, the dual-pool algorithm is presented with a real case study. Section 10.4 concludes this chapter.

## 10.2 Evenness-Aware Algorithm

### 10.2.1 Algorithm Design

#### 10.2.1.1 Overview

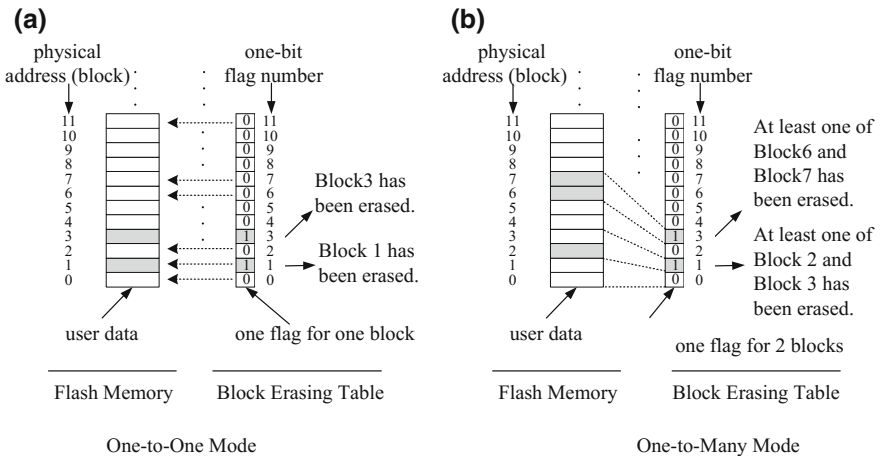
The motivation of the evenness-aware algorithm is to prevent static data from staying at any block for a long period of time. It minimizes the maximum erase-count difference between any two blocks, so flash memory lifetime is extended. This algorithm could be implemented as a module. In this algorithm, it maintain a *Block Erasing Table (BET)* that identifies the blocks erased during a given period of time (Sect. 10.2.1.2). The BET is associated with the process *SW Leveler* that is activated by some system parameters for the needs of static wear leveling (Sect. 10.2.1.3). When the SW Leveler runs, it either resets the BET or picks up a block that has not been erased so far (based on the BET information), and triggers the garbage collector to do garbage collection on the block (note that the selection procedure of a block must be performed efficiently and within a

limited time). Whenever a block is recycled by the garbage collection, any modification to the address translation is performed as in the original design of a flash management design. The SW Leveler can be implemented as a thread or as a procedure triggered by a timer or the garbage collector based on some preset conditions. Note that, whenever a block is erased, the BET must be updated by a triggering action to the SW Leveler. The design of the BET is scalable to accommodate rapidly increasing flash-memory capacity [34] and the limited RAM space on a controller.

### 10.2.1.2 Block Erasing Table

The Block Erasing Table (BET) attempts to remember which block has been erased in a pre-determined time frame, referred to as the *resetting interval*, so as to locate blocks of cold data. A BET is a bit array in which each bit corresponds to a set of  $2^k$  contiguous blocks where  $k$  is an integer that equals or exceeds 0. Whenever a block is erased by the Cleaner, the SW Leveler is triggered to set the corresponding bit as 1. Initially, the BET is reset to 0 for every bit. As shown in Fig. 10.1, information maintenance is performed in one-to-one and one-to-many modes, and one flag is used to track whether any one of the corresponding  $2^k$  blocks is erased. When  $k = 0$ , one flag is used for one block (i.e., in the one-to-one mode). The larger the value of  $k$ , the greater the chance in the overlooking of blocks of cold data. However, a large value for  $k$  could help reduce the RAM space required by a BET controller.

The worst case for a large  $k$  value occurs when hot and cold data co-exist in a block set. Fortunately, such a case is eventually resolved when hot data are



**Fig. 10.1** The mapping mechanism between flags and blocks. **a** One-to-One mode. **b** One-to-Many mode

invalidated. As a result, cold data could be moved to other blocks by the SW Leveler (see Sect. 10.2.1.3). The technical problem relies on the tradeoff between the time to resolve such a case (bias in favor of a small  $k$ ) and the available RAM space for the BET (bias in favor of a large  $k$ ).

Another technical issue is efficiently rebuilding the BET when a flash-memory storage system is attached. One simple but effective solution is to save the BET in the flash-memory storage system when the system shuts down, and then to reload it from the system when needed. Meanwhile, the whole BET is stored in flash memory and loaded to main memory in an on-demand fashion, so that the required main memory could be minimized. If the system is not properly shut down, we propose loading any existing correct version of the BET when the system is attached. Such a solution is reasonable as long as loss of erase count information is not excessive. Note that the crash resistance of the BET information in the storage system could be provided by the popular dual buffer concept. Scanning of the spare areas of pages when collecting related information should also be avoided because of the potentially huge capacity of a flash-memory storage system.

### 10.2.1.3 SW Leveler

The SW Leveler consists of two procedures in executing wear leveling: SWL-Procedure and SWL-BETUpdate (please see Algorithms 1 and 2). SWL-BETUpdate is invoked by the garbage collector to update the BET whenever any block is erased by the garbage collector during garbage collection. The SWL-Procedure is invoked whenever static wear leveling is needed. Such a need is tracked by two variables,  $f_{cnt}$  and  $e_{cnt}$ , which denote the number of 1s in the BET and the total number of block erases performed since the BET was reset, respectively. When the *unevenness level*, i.e., the ratio of  $e_{cnt}$  and  $f_{cnt}$ , equals or exceeds a given threshold  $T$ , SWL-Procedure is invoked to trigger the garbage collector to do garbage collection over selected blocks such that cold data are moved. Note that a high unevenness level reflects the fact that a lot of erases are done on a small portion of the flash memory.

Algorithm 1 shows the algorithm for the SWL-Procedure: the SWL-Procedure simply returns if the BET is just reset (Step 1). When the unevenness level, i.e.,  $e_{cnt}/f_{cnt}$ , equals or exceeds a given threshold  $T$ , the garbage collector is invoked in each iteration to do garbage collection over a selected set of blocks (Steps 2–15). In each iteration, it is checked up if all of the flags in the BET are set as 1 (Step 3). If so, the BET is reset, and the corresponding variables (i.e.,  $e_{cnt}$ ,  $f_{cnt}$ , and  $f_{index}$ ) are reset (Steps 4–7). The  $f_{index}$  is the index in the selection of a block set for static wear leveling and is reset to a randomly selected block set or to a predefined block set, e.g. 0. After the BET is reset, SWL-Procedure simply returns to start the next resetting interval (Step 8). Otherwise, the selection index, i.e.,  $f_{index}$ , moves to the next block set with a zero-valued flag (Steps 10–12). *Note that the sequential scanning of blocks in the selection of block sets for static wear leveling is very effective in the implementation. We surmise that the design approximates that of an*

actual random selection policy because cold data can virtually exist in any block in the physical address space of the flash memory. The SWL-Procedure then invokes the garbage collector to do garbage collection over a selected block set (Step 13) and moves to the next block set (Step 14) for the next iteration. We must point out that  $f_{cnt}$  and BET are updated by SWL-BETUpdate because SWL-BETUpdate is invoked by the garbage collector during garbage collection. The loop in static wear leveling ends when the unevenness level drops to a satisfactory value.

---

**Algorithm 1: SWL-Procedure**


---

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

```

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

```

---



---

**Algorithm 2: SWL-BETUpdate**


---

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

```

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

```

---

The SWL-BETUpdate is as shown in Algorithm 2: Given the address  $b_{index}$  of the block erased by the garbage collector, SWL-BETUpdate first increases the number of blocks erased in the resetting interval (Step 1). If the corresponding BET entry is not 1, then the entry is set as 1, and the number of 1s in the BET is increased by one (Steps 2–5). The remaining technical question is how to maintain

the values of  $e_{cnt}$ ,  $f_{cnt}$ , and  $f_{index}$ . To optimize static wear leveling,  $e_{cnt}$ ,  $f_{cnt}$ , and  $f_{index}$  should be saved to flash memory as system parameters and retrieved in the attachment of the flash memory. Notably, these values can tolerate some errors with minor modifications to SWL-Procedure in either the condition in Step 3 or the linear traversal of the BET (Steps 10–12). That is, if the system crashes before their values are saved to flash memory, it simply uses the values previously saved to flash memory.

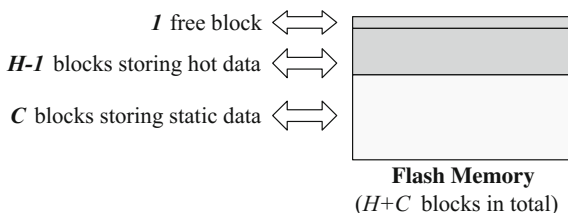
## 10.2.2 Worst-Case Analysis

### 10.2.2.1 Worst-Case Model for Extra Overheads

Block recycling overhead is indeed increased by the proposed evenness-aware algorithm. A very minor cause of the increase is the execution of SWL-BETUpdate whenever the garbage collector erases a block, i.e., the value updates of  $e_{cnt}$  and  $f_{cnt}$  as well as the BET flags (compared to the block erase time, which could be about 1.5 ms over a 1 GB MLC<sub>x2</sub> flash memory [28]). As astute readers might point out, the garbage collector might be triggered more often than before because of wear leveling. That might increase the number of block erases and live-page copyings. The increased overheads caused by extra block erases and extra live-page copyings are apparent in the following worst-case scenario: the flash memory contains blocks of hot data, blocks of static data, and exactly one free block in a resetting interval.

Figure 10.2 shows the worst-case model based on a block-level address translation mechanism. In the block-level address translation mechanism, each LBA is divided into a virtual block address (VBA) and a block offset, and a mapping table is adopted for VBAs and their physical block addresses (PBAs). For each write operation, a free block is allocated to save the data of the remaining valid pages of the original mapped block and the new data of the write operation. Assume there are  $(H - 1)$  blocks of hot data and  $C$  blocks of static data where the number of blocks in the system is  $(H + C)$ . The worst-case situation occurs when the  $C$  blocks are erased, only due to the evenness-aware algorithm. The worst case occurs when hot data are updated with the same frequency and only to the free block or the blocks of hot data, where  $k = 0$ . Sections 10.2.2.2 and 10.2.2.3 show the analyses for extra block erases and extra live-page copyings in the worst-case model, respectively.

**Fig. 10.2** Flash memory of only static data and hot data





### 10.2.2.2 Extra Block Erases

When  $k = 0$ , the BET contains  $(H + C)$  bits, i.e.,  $(H + C)$  1-bit flags. In each resetting interval, when the updates of hot data result in  $(T \times H)$  block erases, SWL-Procedure is activated to recycle one block of cold data for the first time because only  $H$  bits of the BET are set, and the unevenness level reaches  $T$  (i.e.,  $(T \times H)/H$ ). After one block of cold data is recycled by SWL-Procedure,  $(H + 1)$  bits of the BET are set, and the number of block erases reaches  $(T \times H + 1)$ . The unevenness level (i.e.,  $(T \times H + 1)/(H + 1)$ ) is then smaller than the threshold  $T$ . Thereafter, SWL-Procedure is activated to recycle one block of cold data on all other  $(T - 1)$  block erases resulting from hot data updates. Finally, this procedure is repeated  $C$  times such that all BET flags are set and the resetting interval ends. Therefore, the resetting interval has  $T \times (H + C)$  block erases. For every  $T \times (H + C)$  block erases in a resetting interval, SWL-Procedure performs  $C$  block erases. Therefore, the increased ratio of block erases (due to static wear leveling) is derived as follows:

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

The increased ratio is even worse when  $C$  is the dominant part of  $(H + C)$  (an earlier study [18] showed that the amount of non-hot data is often several times that of hot data). Table 10.1 shows different increased ratios in extra block erasing for different configurations of  $H$ ,  $C$ , and  $T$ . As shown in the table, the increased overhead ratio in extra block erasing is sensitive to the setting of  $T$ . Therefore, to avoid excessive triggering of static wear leveling,  $T$  must not be set too small.

### 10.2.2.3 Extra Live-Page Copyings

The extra overheads in live-page copyings due to the static wear leveling mechanism can be explored by the worst-case model. Let  $N$  be the number of pages in a block. Suppose that  $L$  is the average number of pages copied by the garbage collector when erasing a block of hot data. Thus, in the worst case, totally  $(C \times N)$  live-pages are copied when erasing  $C$  blocks of static data (due to the evenness-aware algorithm) in a resetting interval, and  $(T \times (H + C) - C) \times L$

**Table 10.1** The increased ratio of block erases of a 1 GB MLC<sub>×2</sub> flash-memory storage system

H	C	H:C	T	Increased ratio (%)
256	3,840	1:15	10	9.46
2048	2,048	1:1	10	5.03
256	3,840	1:15	100	0.95
2,048	2,048	1:1	100	0.50
256	3,840	1:15	1,000	0.09
2,048	2,048	1:1	1,000	0.05

**Table 10.2** The increased ratio in live-page copyings of a 1 GB MLC<sub>×2</sub> flash-memory storage system

H	C	H:C	T	L	$\frac{N}{T \times L}$	Increased ratio (%)
256	3,840	1:15	10	16	0.800	75.72
2,048	2,048	1:1	10	16	0.800	40.02
256	3,840	1:15	10	32	0.400	37.86
2,048	2,048	1:1	10	32	0.400	20.00
256	3,840	1:15	100	16	0.0800	7.57
2,048	2,048	1:1	100	16	0.0800	4.00
256	3,840	1:15	100	32	0.0400	3.79
2,048	2,048	1:1	100	32	0.0400	2.00
256	3,840	1:15	1,000	16	0.0080	0.76
2,048	2,048	1:1	1,000	16	0.0080	0.40
256	3,840	1:15	1,000	32	0.0040	0.38
2,048	2,048	1:1	1,000	32	0.0040	0.20

live-page copyings are performed in the course of regular garbage collection activities in a resetting interval. The increased ratio in live-page copyings, due to static wear leveling, can be derived as follows:

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

Table 10.2 shows varying increases in the ratios of live-page copyings for different configurations of  $H$ ,  $C$ ,  $T$ , and  $L$ , when  $N = 128$ . The increased ratio of live-page copyings can be estimated by  $\frac{N}{L}$  times the increased ratio of extra block erases. For example, when  $T = 100$ ,  $L = 16$ ,  $N = 128$ , and  $\frac{H}{C} = \frac{1}{15}$ , the increased ratio of block erases is 0.95% (the third row of Table 10.1) and its corresponding increased ratio of live-page copyings is 7.57%, i.e.,  $0.95\% \times \frac{128}{16}$  (the third row of Table 10.2). As shown in Tables 10.1 and 10.2, the increased ratios of block erases and live-page copyings would be limited with a proper selection of  $T$  and other parameters. The increased ratios could be limited to very small percentages of flash management strategies when the evenness-aware algorithm is supported.

## 10.3 Dual-Pool Algorithm

### 10.3.1 Algorithm Design

#### 10.3.1.1 Algorithm Concept

This section introduces the basic concepts of the dual-pool algorithm. Let write requests arriving at the flash storage device be ordered by their arrival times. Let the temperature of a piece of data be inversely proportional to the number of requests

between the two most recent writes to that data. A piece of data is *hot* if its temperature is higher than the average temperature of all data. Otherwise, the data is *cold* or *non-hot*. A block is referred to as a *young(fold) block* if its erase-cycle count is smaller(larger) than the average erase-cycle count of all blocks.

We say that a block contributes or accumulates erase cycles if garbage collection erases this block to reclaim free space. Garbage collection avoids erasing a block having many valid data. If a block has more cold data than other blocks, then it will stop contributing erase cycles. This is because cold data remains valid in the block for a long time. Conversely, if a block has many hot data, then it can accumulate erase cycles faster than other blocks. This is because hot data are invalidated faster than cold data, and the block can become a victim of garbage collection before other blocks. After the block is erased, it can again be written with many hot data, because writes to hot data arrive more frequently than writes to cold data. Thus, this block is again erased and is written with many hot data.

The dual-pool algorithm monitors the erase-cycle count of each block. If an old block's erase-cycle count is larger than that of a young block by a predefined threshold, wear leveling activities are triggered. Cold data are moved to the old block to prevent it from being erased by garbage collection. This strategy is referred to as *cold-data migration*. After this, the old block should stop accumulating erase cycles. Compared to encouraging young blocks to contribute erase cycles, this strategy reduces data-movement overhead. This is because only a small fraction of blocks are worn into old blocks, while the majority are young blocks. Right after cold data are written to an old block, the old block still has a large erase-cycle count. If we are not aware that the old block has been involved in cold-data migration, we may again write some other cold data to the old block. This pointlessly reduces the block's lifetime. Similarly, after a young block is involved in cold-data migration, cold data previously stored in the block are removed. At this point, the young block has no cold data, even though its erase-cycle count is small. So, right after a block is involved in cold-data migration, it should be protected from immediate re-involvement. This strategy is called *block protection*. The protection of an old block is no longer required when other blocks become older than it. The protection of a young block expires when it is worn into an old block.

The access patterns from the host to the flash storage devices can change periodically. For example, a user application in the host may finish using some files and then begin accessing other files. These application-level behaviors can change the frequency with which a piece of data is updated, and thus cold data can change into hot data. Consider an old block written with cold data for cold-data migration. The old block is then protected against cold-data migration. Now suppose that the cold data in the old block happens to become hot. The protected old block will again start participating in garbage collection, and continues to age without interruption from wear leveling because its protection cannot expire. Now consider a young block under protection. The block should accumulate erase cycles. If the young block happens to be written with many cold data, then it stops contributing erase cycles. The young block attracts no attention from wear leveling because its

protection cannot expire. This dilemma highlights the special cases that must be carefully considered by block protection.

### 10.3.1.2 The Dual-Pool Algorithm: A Basic Form

The dual-pool algorithm, as implied by its name, uses a *hot pool* and a *cold pool*. A pool is merely a logical aggregation of blocks. Initially, a block arbitrarily joins one of these two pools. Note that the dual-pool algorithm is not to write cold data to blocks in the cold pool. Instead, it migrates blocks storing cold data to the cold pool.

The dual-pool algorithm uses priority queues to sort blocks in terms of different wearing information. The following section defines some symbols for ease of presentation: Let  $C$  and  $H$  denote the cold pool and the hot pool, respectively. Each element in  $C$  and  $H$  is a block. Let  $U$  be a collection of all blocks.  $C \cap H = \emptyset$  and  $C \cup H = U$  are invariants. Let  $Q_p^w$  be a priority queue that prioritizes all blocks in pool  $P$  in terms of wearing information  $w$ . The larger the value of  $w$  is, the higher the priority is. Each element  $Q_p^w$  in  $w$  corresponds to a block. For block  $b$ , let function  $ec(b)$  present its erase-cycle count. In priority queue  $Q_p^w$ ,  $M(Q_p^w)$  is the element with the highest priority and  $m(Q_p^w)$  is the element with the lowest priority.  $M(Q_p^w)$  and  $m(Q_p^w)$  are referred to as the largest queue head and the smallest queue head, respectively. For example,  $m(Q_C^{ec})$  denotes the block with the smallest erase-cycle count of all the blocks in the cold pool.

The dual-pool algorithm adopts a user-configurable parameter **TH** to direct how even the wear of blocks is to be pursued. The smaller the value of  $TH$  is, the more aggressive the wear-leveling activities would be. Table 10.3 summarizes the symbol definitions, and the following section defines cold-data migration (CDM for short): **Cold-Data Migration (CDM)**: Upon the completion of block erase, check the following condition:

$$ec(M(Q_H^{ec})) - ec(m(Q_C^{ec})) > TH.$$

**Table 10.3** A summary of symbols used in the dual-pool algorithm

Symbol	Definition
$C$	The cold pool, a collection of blocks
$H$	The hot pool, a collection of blocks
$U$	A collection of all blocks. $C \cap H = \emptyset$ and $C \cup H = U$
$Q_p^w$	A priority queue that sorts blocks in pool $P$ in terms of information $w$
$M(Q_p^w)$	The element with the largest priority in $Q_p^w$
$m(Q_p^w)$	The element with the smallest priority in $Q_p^w$
$ec(b)$	The erase-cycle count of block $b$
$rec(b)$	The recent erase-cycle count of block $b$
<b>TH</b>	The threshold parameter for wear leveling

If this condition is true, then the largest erase-cycle count of the blocks in the hot pool is larger than the smallest count of the blocks in the cold pool by  $TH$ . Perform the following procedure:

<b>Step 1.</b>	Copy data from $m(Q_C^{ec})$ to $M(Q_H^{ec})$
<b>Step 2.</b>	Erase $m(Q_C^{ec})$ ; $ec(m(Q_C^{ec})) \leftarrow ec(m(Q_C^{ec})) + 1$
<b>Step 3.</b>	$C \leftarrow C \cup \{M(Q_H^{ec})\}$ ; $H \leftarrow H \setminus \{M(Q_H^{ec})\}$
<b>Step 4.</b>	$H \leftarrow H \cup \{m(Q_C^{ec})\}$ ; $C \leftarrow C \setminus \{m(Q_C^{ec})\}$

Because cold-data migration checks the condition immediately after a block is erased, block  $ec(M(Q_H^{ec}))$  must be the most-recently erased block if the condition is true. Whenever  $ec(M(Q_H^{ec})) - ec(m(Q_C^{ec}))$  is found larger than  $TH$ , it is deduced that, on the one hand, block  $m(Q_C^{ec})$  has not been erased for a long time because of the storing of many cold data. On the other hand, garbage collection had erased block  $M(Q_H^{ec})$  many times, because this block infrequently stores cold data. Next, migrate cold data from block  $m(Q_C^{ec})$  to block  $M(Q_H^{ec})$ . Step 1 moves data from block  $m(Q_C^{ec})$  to block  $M(Q_H^{ec})$  to complete cold-data migration. After this move, block  $M(Q_H^{ec})$  can stop being erased by garbage collection. Step 2 erases block  $m(Q_C^{ec})$  and increases the block's erase-cycle count. This erase does not affect the pool membership of block  $m(Q_C^{ec})$ .

Step 3 moves block  $M(Q_H^{ec})$  to the cold pool, and Step 4 moves block  $m(Q_C^{ec})$  to the hot pool. These steps swap the two blocks' pool memberships, and enable block protection. When the young block (previously  $m(Q_C^{ec})$ ) joins the hot pool, it may be younger than many blocks in the hot pool. That is because most of the blocks in the hot pool are old. The young block is then protected, because cold-data migration is not interested in a young block in the hot pool. Analogously, when the old block (previously block  $M(Q_H^{ec})$ ) migrates to the cold pool, it may be older than many blocks in the cold pool. The old block in the cold pool is then protected, as cold-data migration is concerned with the youngest block in the cold pool.

The young block in the hot pool (previously  $m(Q_C^{ec})$ ) starts accumulating erase cycles. When the block is worn into the oldest in the hot pool, it will again participate in cold-data migration. On the other hand, the old block in the cold pool (previously  $M(Q_H^{ec})$ ) now stops being erased. When the block becomes the youngest in the cold pool, it is again ready for cold-data migration.

### 10.3.1.3 Pool Adjustment

The cold pool collects blocks that store cold data. However, the cold pool may also contain blocks that have no cold data. This may be because all the blocks' pool memberships were arbitrarily decided in the very beginning, as all blocks' erase-cycle counts are initially zero. Another possible cause is that applications in

the host may change their data-access behaviors. These changes can turn a piece of cold data into hot data.

Garbage collection selects erase victims based on how many invalid data a block has, regardless the block's pool membership. If a block has no cold data, it will continue participating in garbage collection even if it is in the cold pool. In this case, the block's erase-cycle count increases without interruption from wear leveling. This is because cold-data migration always involves the youngest block in the cold pool. Similarly, if a block in the hot pool has many cold data, garbage collection avoids erasing this block. The block cannot be erased into the oldest block in the hot pool, and cannot attract attention from wear leveling.

To deal with this problem, the dual-pool algorithm introduces two operations, cold-pool adjustment (CPA for short) and hot-pool adjustment (HPA for short). These two operations identify and correct any improper pool membership in the blocks. Specifically, blocks' pool membership is adjusted according to how frequently they have been erased since their last involvement in cold-data migration. Hot-pool adjustment removes the blocks that do not accumulate erase cycles from the hot pool. Cold-pool adjustment removes the blocks that actively contribute erase cycles from the cold pool. To enable these operations to function, new block-wearing information (i.e., the *recent erase-cycle count*) is introduced. A block's recent erase-cycle count is initially zero. It increases as along with the erase-cycle count, but reset to zero whenever the block is involved in cold-data migration. Thus, cold-data migration includes a new step:

**(CDM) Step 5.**  $rec(M(Q_H^{ec})) \leftarrow 0; rec(m(Q_C^{ec})) \leftarrow 0$

The hot-pool adjustment and cold-pool adjustment operations also require new priority queues and queue heads, which are summarized in Table 10.4. Let function  $rec()$  return the recent erase-cycle count of a block. The hot-pool adjustment and cold-pool adjustment are then as follows:

**Cold-Pool Adjustment (CPA):** Upon completion of block erase, check the following condition:

$$rec(M(Q_C^{rec})) - rec(m(Q_H^{rec})) > TH.$$

**Table 10.4** A summary of the five queue heads used by the dual-pool algorithm

Queue heads	Belongs to	Used in
$M(Q_H^{ec})$	The hot pool	Cold-data migration and hot-pool adjustment
$m(Q_H^{ec})$	The hot pool	Hot-pool adjustment
$m(Q_H^{rec})$	The hot pool	Cold-pool adjustment
$m(Q_C^{ec})$	The cold pool	Cold-data migration
$M(Q_C^{rec})$	The cold pool	Cold-pool adjustment

If it holds, then the largest recent erase-cycle count of the blocks in the cold pool is larger than the smallest count of the blocks in the hot pool by  $TH$ . Perform the following steps:

**Step 1.**  $H \leftarrow H \cup \{M(Q_C^{ec})\}$ ;  $C \leftarrow C \setminus \{M(Q_C^{ec})\}$

If a block has a large recent erase-cycle count, then the block has contributed many erase cycles since the last time it was involved in cold-data migration. Cold-pool adjustment evicts such a block from the cold pool. This is because the last attempt to stop the block from being erased was not successful, or the block did not have cold data in the very beginning.

**Hot-Pool Adjustment (HPA):** Upon completion of block erase, check the following condition:

$$ec(M(Q_H^{ec})) - ec(m(Q_H^{ec})) > 2 \times TH.$$

If this condition holds, then in the hot pool the smallest erase-cycle count is smaller than the largest count by  $2 \times TH$ . Perform the following steps:

**Step 1.**  $C \leftarrow C \cup \{m(Q_H^{ec})\}$ ;  $H \leftarrow H \setminus \{m(Q_H^{ec})\}$

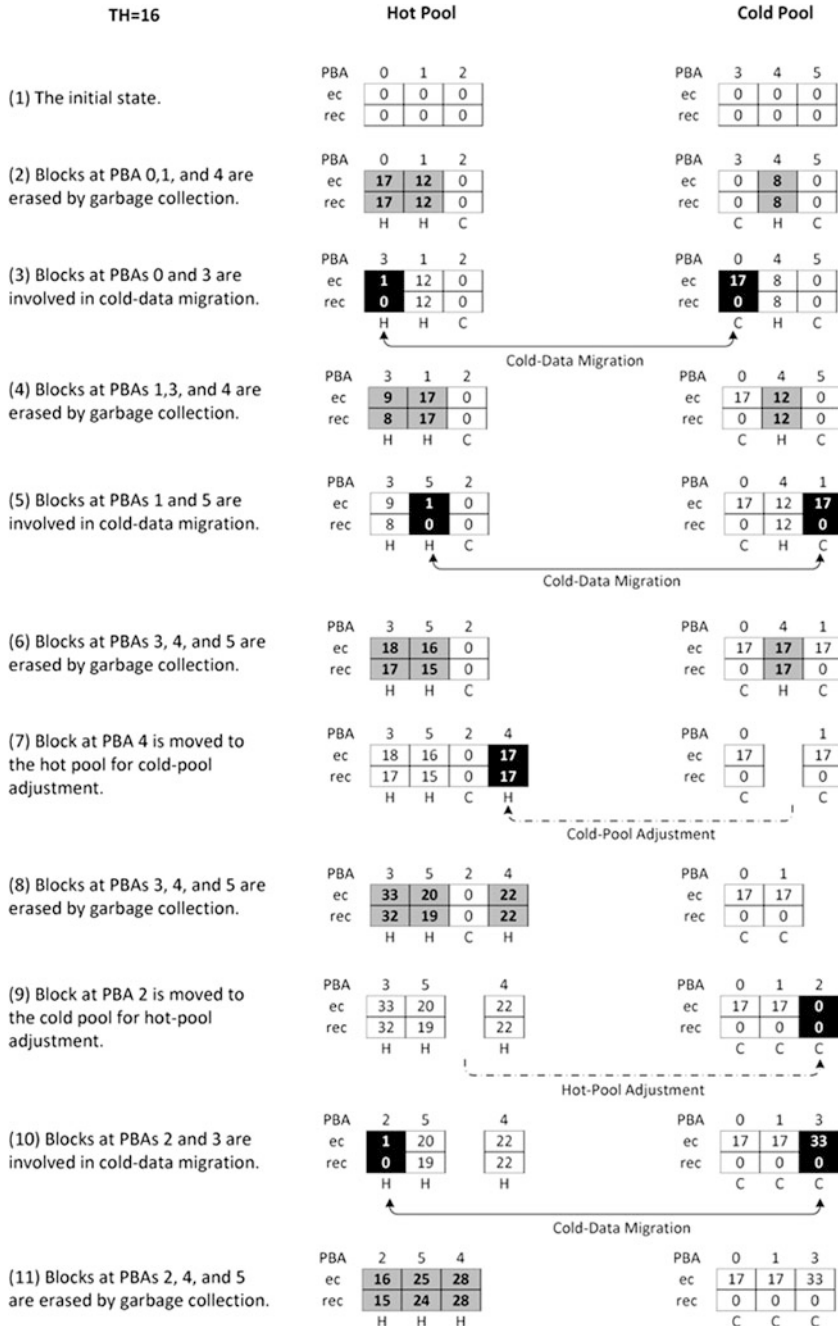
Whether or not a block should be written with cold data for wear leveling depends on the size of its erase-cycle count. If a block in the hot pool accumulates erase cycles more slowly than other blocks, then the block contains cold data, and the hot-pool adjustment operations removes this block from the hot pool. *Readers may question that why  $2 \times TH$  is in this condition.* It is to prevent hot-pool adjustment from conflicting with cold-data migration: when cold-data migration moves a young block from the cold pool to the hot pool, the young block's erase-cycle count is already smaller than the oldest block in the hot pool by  $TH$  (see the condition for cold-data migration). To prevent hot-pool adjustment from immediately bouncing the young block back to the cold pool, the condition of hot-pool adjustment allows additional  $TH$  cycles ( $2 \times TH$  in total).

In the worst case, every time after cold-data migration writes cold data to an old block and moves this block to the cold pool, the cold data become hot. Cold-pool adjustment can identify this old block and move it to the hot pool, after the block contributes  $TH$  more cycles of erase operations. Right after this, cold-data migration makes another attempt to write cold data to the block. So in this worst case, the dual-pool algorithm guarantees to involve this old block every other  $TH$  erase operations to this block.

#### 10.3.1.4 Algorithm Demonstration

This section presents an example demonstrating how the dual-pool algorithm accomplishes wear leveling.

In Fig. 10.3, there are six flash-memory blocks, labeled from PBA 0 to PBA 5. The threshold parameter  $TH$  is 16. In the illustration, each block corresponds to two



**Fig. 10.3** A scenario of the dual-pool algorithm. There are six flash-memory blocks, labeled from PBA 0 to PBA 5. Each block is associated with an erase-cycle count (ec), a recent erase-cycle count (rec), and the attribute of its data (hot or cold)



boxes, which indicate the block's erase-cycle count (ec) and recent erase-cycle count (rec). If a block currently stores cold data, then "C" appears under the block's boxes, and "H" otherwise. The example includes 11 steps. At each step, a block's boxes are shaded in gray if the block has been erased by garbage collection since the last step. A block's boxes are indicated black if it is currently involved in wear leveling. The following discussion refers to a block at PBA  $x$  as Block  $x$ , where  $x$  can be from 0 to 5.

In Step 1, the first three blocks join the hot pool and the rest join the cold pool. Step 2 shows that Blocks 0, 1, and 4 start accumulating erase cycles because they store no cold data. At this point, the largest erase-cycle count in the hot pool and the smallest erase-cycle count in the cold pool are 17 and 0, respectively. As this difference is greater than  $TH = 16$ , cold-data migration is triggered. Step 3 shows that the cold data in Block 3 are moved to Block 0, and the pool memberships are switched for both blocks. Notice that a block's wearing information sticks together with that block during cold-data migration. In Step 4, garbage collection erases Blocks 1, 3, and 4 because they had no cold data since Step 3.

Block 0, an old block previously involved in cold-data migration, is written with cold data and stops accumulating erase cycles since Step 3. Even though Block 0 is the oldest among all the blocks in the cold pool, it is now protected against cold-data migration because it is not youngest in the cold pool. In Step 5, cold-data migration is triggered by Blocks 1 and 5, and cold data are migrated from Blocks 5 to 1. In Step 6, Blocks 3–5 contribute some more erase cycles since Step 5. Note that after two cold-data migrations, Blocks 0 and 1, which were previously the contributors of erase cycles in Step 2, now store cold data in the cold pool and are no longer being erased.

In Step 6, Block 4 in the cold pool stores no cold data. In Step 7, it is evicted from the cold pool by cold-pool adjustment, because the difference between Block 4's recent erase-cycle count and the smallest recent erase-cycle count in the hot pool (i.e., that of Block 2) is greater than  $TH = 16$ . In Step 8, Blocks 3–5 keep accumulating erase cycles, and have done so since Step 5. In Step 9, hot-pool adjustment is triggered because the difference between the erase-cycle counts of Blocks 2 and 3 is greater than  $2 \times TH = 32$ . Hot-pool adjustment moves Block 2 to the cold pool. Right after Step 9, cold-data migration for Blocks 2 and 3 occurs in Step 10. In Step 11, garbage collection erases some more blocks. At this point, the wear of all blocks is considered even, with respect to  $TH = 16$ .

### ***10.3.2 Case Study: An SSD Implementation of the Dual-Pool Algorithm***

#### **10.3.2.1 The Firmware and Disk Emulation**

The SSD platform in this study is the FreeScale M68KIT912UF32 development kit [15, 25]. This platform integrates an MC9S12UF32 SoC (referred to as the SSD

controller hereafter), various flash-memory interfaces, and a USB interface. The controller contains a 16-bit MCU M68HCS12, 3 KB of RAM, 32 KB of EEPROM, a USB 2.0 interface controller, various flash-memory host controllers, and a DMA engine with an 1.5 KB buffer. The MCU is normally rated at 33 MHz. The NAND flash considered in this study is a 128 MB SmartMedia card (abbreviated as SM card hereafter). SM cards have the same appearance as bare NAND-flash chips in terms of physical characteristics. The block size and the page size of the SM card are 16 KB and 512 bytes, respectively, and it has a block endurance of 100 K erase cycles. Readers may notice that its geometry is finer than that of mainstream NAND flash memory [37]. However, the design and implementation of the proposed algorithm is independent of the block size and the page size.

An SSD presents itself to the host system as a logical disk,<sup>2</sup> so ordinary disk-based file systems (such as FAT and NTFS) are compatible with SSDs. The flash-translation layer (FTL), which is a part of SSD firmware, performs disk emulation [21, 22, 26, 44]. Basically, FTL implements a mapping scheme, an update policy, and a garbage-collection policy. For ease of presentation, this section introduces some necessary terms and assumptions: Let a disk be addressed in terms of disk sectors, each of which is as large as a flash-memory page. A *physical block* refers to a flash-memory block. Let the entire disk space be partitioned in terms of *logical blocks*, each of which is as large as a physical block. LBAs and PBAs are abbreviations of logical-block addresses and physical-block addresses, respectively. Let a *physical segment* be a group of contiguous physical blocks, and a *logical segment* be a group of contiguous logical blocks.

The FTL needs logical-to-physical translation because data in flash memory are updated out of place. However, a solid-state-disk controller cannot afford the space overhead of the RAM-resident data structures for this translation. To save RAM-space requirements, the FTL adopts a two-level mapping scheme. The first level maps eight logical segments to eight physical segments. This first-level mapping has a one-to-one correspondence. The first level uses a RAM-resident segment translation table (“segment L2P table” for short). This table is indexed by logical-segment numbers, and each table entry represents a physical-segment number. As the first level maps a logical segment to a physical segment, the second level uses a RAM-resident block translation table (“block L2P table” for short) to map the 1,000 logical blocks in the logical block to the 1,024 physical blocks in the physical segment. This table is indexed by logical-block addresses and each table entry represents a physical-block address. Each physical segment has  $1,024 - 1,000 = 24$  unmapped physical blocks, which are spare blocks for garbage collection and bad-block retirement. Thus, the SSD has a total volume of  $8 * 1,000 = 8,000$  logical blocks, while the SM card has  $8 * 1,024 = 8,192$  physical blocks.

---

<sup>2</sup>A logical disk is also referred to as a logical unit (i.e., LUN) [30].

The FTL sequentially writes all sectors of a logical block to the physical block mapped to this logical block, because the smallest granularity for address translation is one block. To translate an LBA into a PBA, first divide the LBA by 1,000. The quotient and the remainder are the logical-segment number and the logical-block offset, respectively. Looking up the segment L2P table and the block L2P table generates a physical segment number and a physical-block offset, respectively. The final PBA is calculated by adding the physical-block offset to the physical-segment number multiplied by 1,024.

For this FTL, there are two types of sector write operations: a write no larger than 4 KB (i.e., eight 512-byte disk sectors) and a write larger than 4 KB. A write larger than 4 KB effectively rewrites a logical block with the necessary copy-back operations: Unchanged sector data are copied from the logical block encompassing the written sectors, and combined with the newly written sector data. A spare block is allocated from the physical segment to which the logical block is mapped, and the combined data are then written to the spare block. The block L2P table is then revised to re-map the logical block to the spare block. The old physical block of the invalidated logical block is erased and converted to a spare block. Spare blocks are allocated in a FIFO fashion for fair use.

Writes no larger than 4 KB are handled in a different way. In this case, a separate spare block collects the newly written data. This spare block is referred to as a log block, as it can be seen as a log of small writes. Whenever the log block is full, the logical blocks modified by the writes recorded in the log block must be rewritten with copy-back operations to apply the changes. In this way, rewriting logical blocks is delayed until the log block is full. After rewriting all the involved logical blocks, the physical blocks previously mapped to the logical blocks and the spare blocks can be erased and converted to spare blocks. Note that the 4 KB threshold is an empirical setting, and this study provides no further discussion on it. erase and data copy activities for free-space reclaiming are referred to as *garbage collection*.

Figure 10.4 depicts a scenario of the proposed disk-emulation algorithm involving three logical blocks and five physical blocks. Let each physical block have four pages, and let each page be as large as a disk sector. A write is considered large if it is larger than two sectors. The left upper corner shows the initial state. Let a write be denoted by sector numbers enclosed within a pair of braces. Three small writes {0}, {0}, and {0, 1} arrive in turn. As they are small, they are appended to the free space in the log block at PBA 1 in Step 1. At this point, the log block is full. Step 2 then conducts copy-back operations to gather valid data from blocks at PBAs 0 and 1, and then rewrites the valid data to the block at PBA 3. Step 3 erases the blocks at PBAs 0 and 1. Step 4 revised the block L2P table. In Step 5, the fourth write {5, 6, 7} arrives. This write is large, and therefore requires that a logical block be rewritten. However, the unchanged data of Sector 4 are first copied from the block at PBA 4 to the log block at PBA 0. Step 6 then appends {5, 6, 7} to the log block, and Step 7 erases the block of invalid data. Step 8 then revises the block L2P table. Note that disk emulation is traditionally considered to be an issue independent of wear leveling. Refer to [19, 21, 22, 44] for further discussion on disk-emulation algorithms.

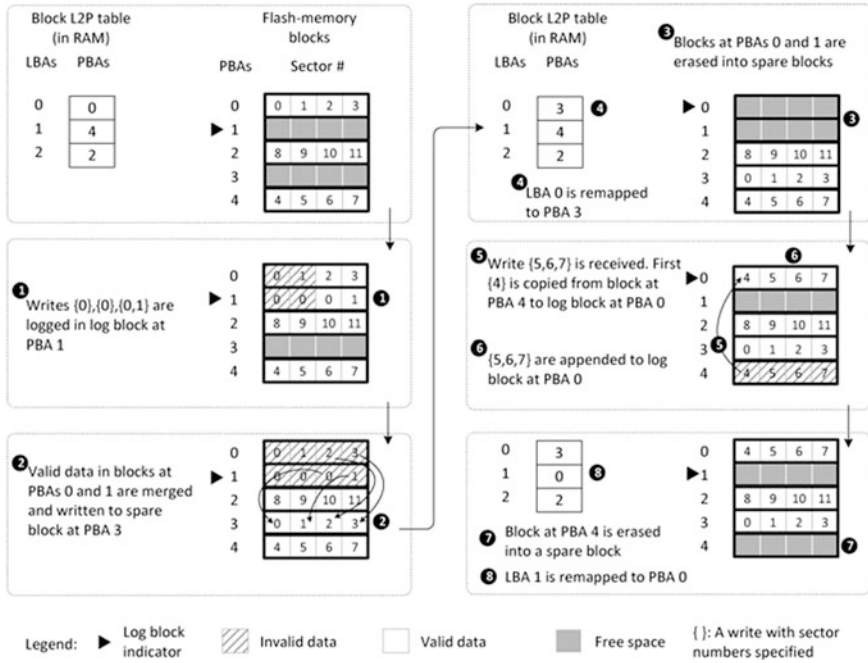


Fig. 10.4 A scenario of our disk-emulation algorithm

The segment L2P table is small enough to be kept in RAM because it has only eight entries. There are eight block L2P tables, one for each pair of a logical segment and a physical segment. As mentioned above, since RAM space is very limited, only two block L2P tables can be cached in RAM. Whenever a block L2P table is needed but is absent from RAM, the least-recently used table in the cache is discarded. The needed table is then constructed by scanning all the physical blocks of the corresponding physical segment. This scanning involves only the spare areas of every physical block's first page, which contain the mapping information.<sup>3</sup>

### 10.3.2.2 Block-Wearing Information and Priority Queues

The dual-pool algorithm keeps track of every block's wearing information. This includes an erase-cycle count, a recent erase-cycle count, and pool membership. Ideally, this information should be kept in RAM for efficient access. However,

<sup>3</sup>The scanning is read-only and does not affect wear leveling. Previous research has developed excellent methods for reducing the time overhead of this scanning. Refer to [19, 21] for details.

this is not feasible because the SSD controller has only about 1 KB of RAM as working space.

One option is to write a block's wearing information in its spare areas [26]. In this approach, a block's wearing information must be committed to one of its spare areas immediately after the block is erased. Later on, when user data are written, error-correcting codes and mapping information are also written to these spare areas. However, this approach can overwrite a spare area multiple times. This is prohibited by many new NAND flash [36, 37]. One alternative is to exclusively write the wearing information to a spare area, but this spoils the existing data layout in spare areas for disk emulation.

Our approach is to reserve one physical block for writing the wearing information. An on-flash block-wearing information table ("**BWI table**" for short) keeps the blocks' wearing information. A new BWI-table can be written to an arbitrarily allocated spare block, which means that the BWI table is subject to wear leveling. Since the entire flash memory is divided into eight physical segments, each segment has its own BWI table. A BWI table contains 1,024 entries, one for each physical block. Each table entry has 4 bytes, including a 18-bit erase-cycle count, a 13-bit recent erase-cycle count, and 1 bit for pool membership. Note that 13 bits are large enough for a recent erase-cycle count because it is reset upon cold-data migration. A BWI table is  $1,024 * 4 = 4$  KB large, so one 16-KB physical block can accommodate four revisions of a BWI table. If the block is full, another spare block is allocated for writing the BWI table, and the prior block is discarded for erase.

The on-flash BWI table can be entirely rewritten every time a block's wearing information changes. However, this method considerably increases write traffic to flash memory. Instead, the PBAs of the recently erased blocks are temporarily logged in a RAM buffer. In the current design, this buffer, named the erase-history table ("**EH table**" for short), has eight entries. If the EH table is full, a new version of the BWI table is written to the block reserved for the BWI table to apply the changes. After this, the in-RAM EH table is emptied.

Blocks are sorted in terms of different wearing information, and the dual-pool algorithm must check queue heads every time it is invoked. To scan the on-flash BWI table to find the queue heads is very slow. To reduce the frequency of BWI-table scanning, a small number of queue-head elements can be fetched for later use. For example, for fast access to  $M(Q_H^{ec})$ , after the BWI table is scanned, the wearing information of the two blocks with the two largest erase-cycles counts in the hot pool can be stored in RAM. An in-RAM queue-head table ("**QH table**" for short) is created for this purpose. The size of the QH table is fixed, and each of the five types of queue heads (shown in Table 10.4) is allocated to two table entries. A QH-table entry consists of a 2-byte PBA and 4-byte block-wearing information. Cold-data migration, hot-pool adjustment, and cold-pool adjustment check the QH table for queue heads. Wear leveling consumes QH-table entries and modifies the wearing information in the entries. A modified table entry is treated as an EH-table entry. The following section discusses when and how a QH table can be refreshed.

### 10.3.2.3 Segment Check-In/Check-Out

This section shows how the proposed wear-leveling data structures can be integrated into the segmented management scheme for disk emulation.

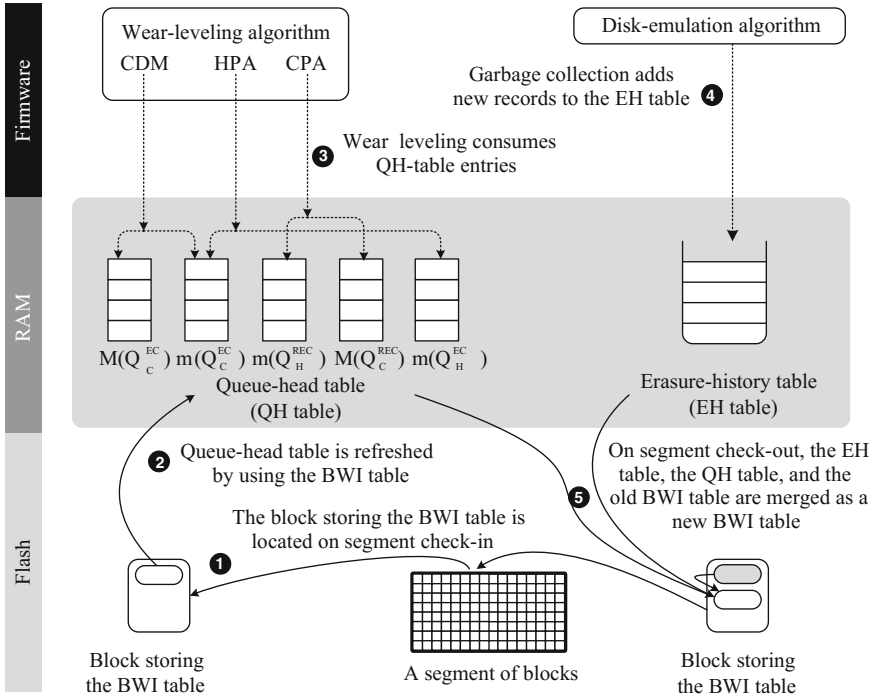
Disk emulation uses a two-level mapping scheme, as previously mentioned in Sect. 10.3.2.1. The segment L2P table is indexed by logical-segment numbers, has only eight entries, and is always stored in RAM. Second-level mapping manages the physical segments as if they were small pieces of flash memory. Each segment has an in-RAM L2P table, which maps 1,000 logical blocks to 1,024 physical blocks. Only two segments can have their block L2P tables cached in RAM. A segment is cached if its block L2P table is in RAM.

Each of the two cached segment uses an in-RAM EH table and an in-RAM QH table. Whenever a logical block is accessed, the corresponding physical segment is located by the segment L2P table. The dual-pool algorithm then checks if the segment's block L2P table, the EH table, and the QH table are in RAM. If they are absent, the following procedure, named *segment check-in*, is performed to bring them in: The in-RAM block L2P table is constructed by scanning the spare areas of each block's first page containing the mapping information. During scanning, if a block is found storing the on-flash BWI table, then the most up-to-date BWI table in the block is scanned to create the in-RAM QH table. By the end of this segment check-in procedure, the QH table and the block L2P table are ready. The in-RAM EH table is emptied, and the segment is all set for data access.

As the EH table continues to record the PBAs of erased blocks, sooner or later it will become full. In this case, a new version of the on-flash BWI table should be created to merge the wearing information in the current on-flash BWI table, the in-RAM EH table, and the in-RAM QH table. The QH table is involved because QH-table entries could have been switched to EH-table entries. This merging procedure, called the *BWI-table merge*, is as follows: First the block storing the current BWI table is located. The dual-pool algorithm creates a new BWI table in the same block right after the current BWI table. If there is no free space left, a new spare block is allocated. The four flash-memory pages storing the current BWI table are then copied to the new location. During copying a BWI-table page, the DMA engine first loads one of the four pages from flash memory into the DMA buffer, and then the dual-pool algorithm performs a three-way synchronization that involves the wearing information from the DMA buffer, the QH table, and the EH table. By the end of this merging procedure, the QH table is refreshed to contain new queue-head physical block addresses and their wearing information, and the EH table is emptied.

A segment's in-RAM data structures can also be evicted from RAM to accommodate those of a newly accessed segment. Before a segment vacates RAM space, its EH table and QH table must be merged with the on-flash BWI table. This process is called *segment check-out*. To check out a segment, the BWI-table merge procedure is first performed, and the in-RAM structures of the segment can then be discarded.

Figure 10.5 shows how by wear leveling, disk emulation, and segment operations use the proposed data structures. Step 1 shows that when a segment is checked



**Fig. 10.5** Relationship between the in-RAM/on-flash data structures and how they are used by wear leveling, disk emulation, and segment operations

in, the spare areas of the blocks in that segment are scanned to build the in-RAM block L2P table. This scanning process also locates the block storing the on-flash BWI table. Step 2 refreshes the in-RAM QH table of the segment with information in the on-flash BWI table. Step 3 shows that QH-table entries are consumed by wear leveling. If any block is erased by garbage collection, then a record of the erase is appended to the in-RAM EH table, as shown in Step 4. When the segment is checked out, Step 5 merges the information in the in-RAM QH table, in-RAM EH table, and on-flash old BWI table and writes it to a new BWI table on flash.

### 10.4 Conclusion

This work addresses a key endurance issue in the deployment of flash memory in various system designs. Unlike the wear leveling algorithms proposed in the previous work, two efficient wear leveling algorithms (i.e., the evenness-aware algorithm and dual-pool algorithm) are presented to solve the problems of the existing algorithms with the considerations of the limited computing power and memory space in flash storage devices. The evenness-aware algorithm proactively moves

static or infrequently updated data with an efficient implementation and limited memory-space requirements so as to spread out the wear-leveling actions over the entire physical address space. It proposes an adjustable house-keeping data structure and an efficient wear leveling implementation based on cyclic queue scanning. Its goal is to improve the endurance of flash memory with only limited increases in overhead and without extensive modifications of popular implementation designs. The dual-pool algorithm is to protect a flash-memory block from being worn out if the block is already excessively erased. This goal is accomplished by moving rarely updated data to excessively erased blocks. Because the micro-controllers of flash storage devices are subject to very tight resource budgets, keeping track of wear levels for a large number of blocks is a very challenging task. The dual-pool algorithm keeps only the most frequently accessed data in RAM, while the rest is written to flash memory.

## References

1. A. Ban, Flash file system. US Patent 5,404,485, in *M-Systems*, Apr 1995
2. A. Ban, Wear leveling of static areas in flash memory. US Patent 6732221 (2004)
3. A. Ban, R. Hasbaron, Wear leveling of static areas in flash memory, US Patent 6,732,221, in *M-systems*, May 2004
4. A. Ben-Aroya, S. Toledo, Competitive analysis of flash-memory algorithms, in *Proceedings of the 14th Conference on Annual European Symposium* (2006)
5. L.-P. Chang, On efficient wear-leveling for large-scale flash-memory storage systems, in *22nd ACM Symposium on Applied Computing (ACM SAC)*, Mar 2007
6. L.-P. Chang, T.-W. Kuo, Efficient management for large-scale flash-memory storage systems with resource conservation. *ACM Trans. Storage* **1**(4), 381–418 (2005)
7. L.-P. Chang, T.-W. Kuo, S.-W. Lo, Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* **3**(4), 837–863 (2004)
8. Y.-H. Chang, J.-W. Hsieh, T.-W. Kuo, Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design, in *DAC'07: Proceedings of the 44th Annual Conference on Design Automation* New York, NY, USA, (ACM, 2007), pp. 212–217
9. M.L. Chiang, P.C.H. Lee, R. Chuan Chang, Using data clustering to improve cleaning performance for flash memory. *Softw. Pract. Exp.* **29**(3), 267–290 (1999)
10. R.J. Defouw, T. Nguyen, Method and system for improving usable life of memory devices using vector processing. US Patent 7139863 (2006)
11. DRAM market-share games shifting from a knockout to a marathon; 4× nm process and multi-bit/cell as fundamental criteria to judge NAND Flash production competitiveness. Technical report, DRAMeXchange, Apr 2008
12. R.A.R.P. Estakhri, M. Assar, B. Iman, Method of and architecture for controlling system data with automatic wear leveling in a semiconductor non-volatile mass storage memory. US Patent 5835935 (1998)
13. Flash Cache Memory Puts Robson in the Middle. Intel
14. Flash-memory translation layer for NAND flash (NFTL). M-Systems (1998)
15. Freescale Semiconductor. USB Thumb Drive reference design DRM061 (2004)
16. FTL Logger Exchanging Data with FTL Systems. Technical Report, Intel
17. C.J. Gonzalez, K.M. Conley, Automated wear leveling in non-volatile storage systems. US Patent 7120729 (2006)



18. Increasing Flash Solid State Disk Reliability. Technical report, SiliconSystems, Apr 2005
19. J.-U. Kang, H. Jo, J.-S. Kim, J. Lee, A superblock-based flash translation layer for NAND flash memory, in *EMSOFT '06: Proceedings of the 6th ACM and IEEE International Conference on Embedded Software*, New York, NY, USA (ACM, 2006), pp. 161–170
21. H.-J. Kim, S.-G. Lee, An effective flash memory manager for reliable flash memory space management. *IEICE Trans. Inf. Syst.* **85**(6), 950–964 (2002)
22. J. Kim, J.-M. Kim, S. Noh, S.-L. Min, Y. Cho, A space-efficient flash translation layer for compact flash systems. *IEEE Trans. Consum. Electron.* **48**(2), 366–375 (2002)
23. S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, H.-J. Song, A log buffer-based flash translation layer using fully-associative sector translation. *Trans. Embed. Comput. Syst.* **6**(3), 18 (2007)
24. Micron Technology, *Wear-Leveling Techniques in NAND Flash Devices* (2008)
25. Microsoft, Flash-memory abstraction layer (FAL), in *Windows Embedded CE 6.0 Source Code* (2007)
26. Motorola, Inc., MC9S12UF32 System on a Chip Guide V01.04 (2002)
27. M-Systems, Flash-Memory Translation Layer for NAND Flash (NFTL) (1998)
28. M-Systems. TruffFS Wear-Leveling Mechanism, Technical Note TN-DOC-017 (2002)
29. NAND08Gx3C2A 8Gbit Multi-level NAND Flash Memory. STMicroelectronics (2005)
30. Numonyx, Wear Leveling in NAND Flash Memories (2008)
31. Open NAND Flash Interface (ONFi), Open NAND Flash Interface Specification Revision 2.1 (2009)
20. K. Perdue, *Wear Leveling* (2008)
32. C. Ruemmler, J. Wilkes, UNIX disk access patterns, in *Usenix Conference* (Winter 1993), pp. 405–420
33. D. Roselli, J.R. Lorch, T.E. Anderson, A comparison of file system workloads, in *Proceedings of the USENIX Annual Technical Conference*, pp. 41–54
34. M. Rosenblum, J.K. Ousterhout, The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* **10**(1) (1992)
35. Samsung Electronics, K9F2808U0B 16 M \* 8 Bit NAND Flash Memory Data Sheet (2001)
36. Samsung Electronics Company, K9GAG08U0 M 2G \* 8 Bit MLC NAND Flash Memory Data Sheet (Preliminary)
37. Samsung Electronics Company, K9NBG08U5 M 4 Gb \* 8 Bit NAND Flash Memory Data Sheet
38. SanDisk Corporation, Sandisk Flash Memory Cards Wear Leveling (2003)
39. D. Schmidt, Technical note: Trueffs wear-leveling mechanism (tn-doc-017). Technical report, M-System (2002)
40. Software Concerns of Implementing a Resident Flash Disk. Intel
41. Spectek, NAND Flash Memory MLC (2003)
42. M. Spivak, S. Toledo, Storing a persistent transactional object heap on flash memory, in *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems* (2006), pp. 22–33
43. STMicroelectronics, Wear Leveling in Single Level Cell NAND Flash Memories (2006)
44. S.P.D.-H.L.S.-W.L. Tae-Sun Chung, D.-J. Park, H.-J. Song, System software for flash memory: a survey, in *EUC '06: Embedded and Ubiquitous Computing* (2006), pp. 394–404
45. Understanding the Flash Translation Layer (FTL) Specification. Technical report, Intel Corporation (Dec 1998), <http://developer.intel.com/>
46. W. Vogels, File system usage in windows nt 4.0. *SIGOPS Oper. Syst. Rev.* **33**(5), 93–109 (1999)
47. D. Woodhouse, Jffs: the journalling flash file system, in *Proceedings of Ottawa Linux Symposium* (2001)
48. M. Wu, W. Zwaenepoel, eNVy: a non-volatile main memory storage system, in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1994), pp. 86–97