

# 國立交通大學

資訊科學與工程所

博士論文

高效能快閃記憶體轉換層設計之研究

A Study on the Design of High Performance  
Flash Translation Layers

指導教授：張瑞川 張大緯 教授

研究生：喬夢麟

中華民國一百零二年十二月

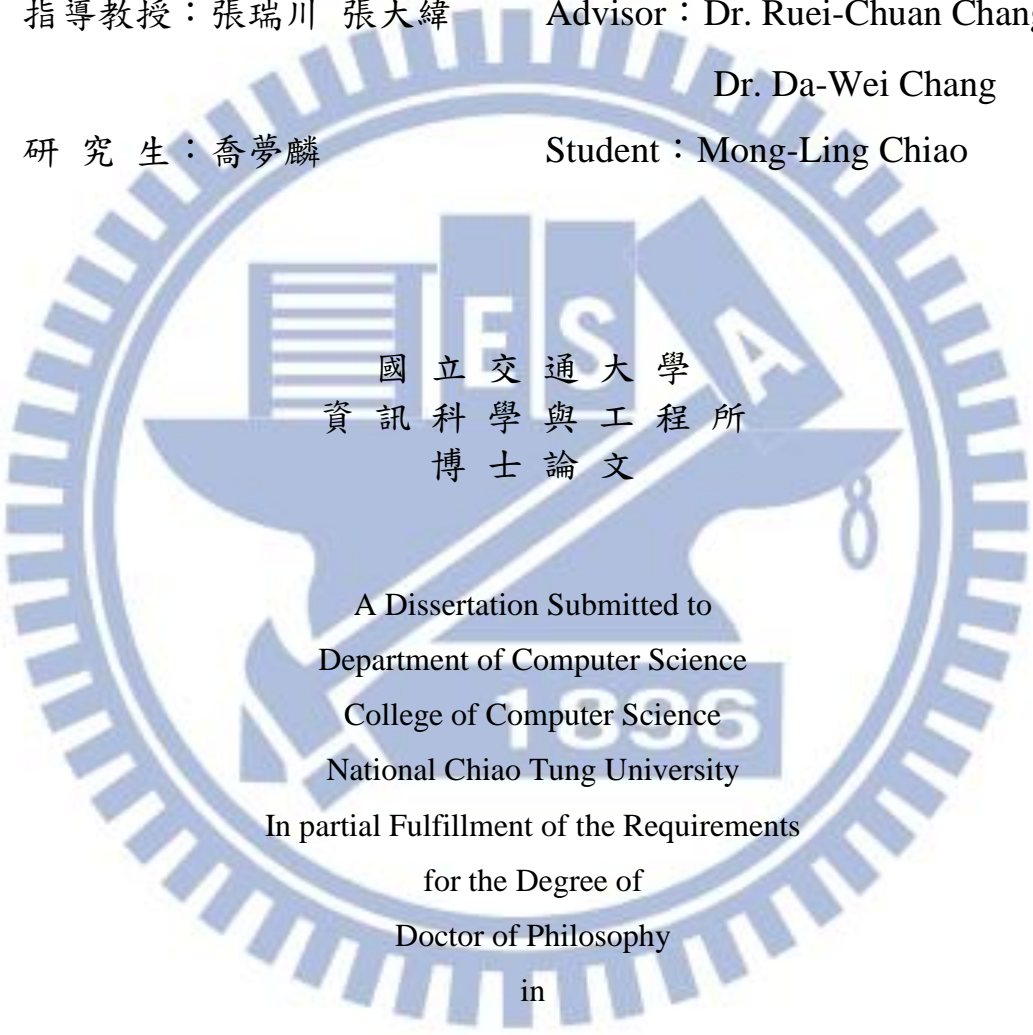
高效能快閃記憶體轉換層設計之研究  
A Study on the Design of High Performance  
Flash Translation Layers

指導教授：張瑞川 張大緯 Advisor：Dr. Ruei-Chuan Chang and

Dr. Da-Wei Chang

研究生：喬夢麟

Student：Mong-Ling Chiao



國立交通大學  
資訊科學與工程所  
博士論文

A Dissertation Submitted to  
Department of Computer Science  
College of Computer Science  
National Chiao Tung University

In partial Fulfillment of the Requirements  
for the Degree of  
Doctor of Philosophy  
in  
Computer Science

December 2013

Hsinchu, Taiwan, Republic of China

中華民國一百零二年十二月

# 高效能快閃記憶體轉換層設計之研究

研究生：喬夢麟

指導教授：張瑞川 博士  
張大緯 博士

國立交通大學  
資訊科學與工程研究所

## 摘要

快閃記憶體在支援磁碟使用的檔案系統時，需要在快閃記憶體上增加一個快閃記憶體轉換層，提供區塊裝置式介面。因為快閃記憶體有寫入前需抹除的特性，快閃記憶體轉換層使用異地更新及清除流程去回收含廢棄資料的區塊。這些回收的成本是快閃記憶體轉換層效能的關鍵，因為清除流程使用的動作，例如複製頁、抹除區塊等等，都非常耗時。為達高效能的目的，快閃記憶體轉換層應該要最小化清除成本。

為了索引邏輯頁的實體頁位置，快閃記憶體轉換層負責維護兩者之間的對映表格。混合式位置轉換快閃記憶體轉換層將快閃記憶體區分為兩個區域，大的資料區域使用邏輯區塊對應實體區塊的管理；小的日誌區域使用邏輯頁對實體頁管理。藉此控制對映資訊的數量，並且達到良好的效能。

本論文提出了兩個混合式位置轉換快閃記憶體轉換層。第一個名為 ROSE，其中包含了三項用來降低清除成本的新技術。首先，它透過避免連續寫入整個區塊的頁落入不同的區塊，藉此降低回收的成本；同時，不會將隨機寫入、不完全連續寫入，誤判為連續寫入，避免因誤判隨之而來的代價。其次，採用針對混合式位置轉換快閃記憶體轉換層來設計、同時考

量區塊的新舊與合併成本的清除方針，藉此提高清除的效率。最後，藉由延遲抹除尚有空白頁的廢棄區塊，並回收使用這些空白頁。

第二個快閃記憶體轉換層名為 HybridLog。透過有效率的使用備用區域，HybridLog 在所有的區塊進行日誌式的寫入，有效率的支援新型的 NAND 快閃記憶體。日誌式寫入能夠避免寫入無謂的空白資料進入頁，以及降低因為資料區域目標頁已經被寫過、而寫入日誌區域的機率。

我們透過模擬評估上述兩個我們所提出的快閃記憶體轉換層的效能。我們使用三個知名的混合式位置轉換快閃記憶體轉換層作為效能比較對象。評估結果顯示，我們所提出的快閃記憶體轉換層表現優於比較對象。



# A Study on the Design of High Performance Flash Translation Layers

Student : Mong-Ling Chiao

Advisors : Dr. Ruei-Chuan Chang  
Dr. Da-Wei Chang

Department of Computer Science  
National Chiao Tung University

## ABSTRACT

A Flash Translation Layer (FTL) provides a block device interface on top of flash memory to support disk-based file systems. Due to the erase-before-write feature of flash memory, an FTL usually performs out-of-place updates and uses a cleaning procedure to reclaim blocks with stale data. The cost of cleaning is a key factor to the performance of an FTL since cleaning involves time-consuming operations such as live page copying and block erasure. To achieve high performance, an FTL should minimize the cleaning cost.

To locate each logical page, an FTL manages the mapping (i.e., address translation) between logical page numbers (LPNs) and physical page numbers (PPNs). By dividing the flash memory into two areas, a large data area managed by coarse-grained address translation and a small area managed by fine-grained address translation, a hybrid address translation (HAT)-based FTL can achieve good performance while keeping the size of the mapping information small.

In this thesis, two novel HAT-based FTLs are proposed. The first FTL, called ROSE, includes three novel techniques for reducing the cleaning cost. First, it reduces high-cost reclamation by preventing data in an entire-block sequential write from being placed into multiple physical blocks while eliminating the cleaning cost resulting from mispredicting random or semi-sequential writes as sequential ones. Second, it uses a novel cleaning policy that considers both the block age and the cleaning cost in a HAT-based FTL for improving the cleaning efficiency. Third, it delays the erasure of obsolete blocks and reuses their free pages for buffering more writes.

The second FTL, called HybridLog, supports modern NAND flash memories by enabling log-style write in all the blocks and efficient use of spare area. The use of log-style write also achieves low cleaning cost by eliminating writes of dummy pages to the data blocks and by reducing the write traffic to the small-sized log area.

The performance of the two proposed FTLs is evaluated through simulation. Three well-known HAT-based FTLs are used for performance comparison. The evaluation results show that the two proposed FTLs outperform the HAT-based FTLs.

## 誌 謝

進入交大資工所九年了。首先要感謝的是我的指導教授張瑞川博士，給我一窺門徑的機會。在有限的時間內，讓門外漢的我有機會學習到學術的嚴謹與趣味。其次要感謝的是目前任職於成功大學，共同指導我的張大緯學長。張大緯學長細心指導我研究方法與撰寫論文的諸多細節，令我大開眼界且茅塞頓開。

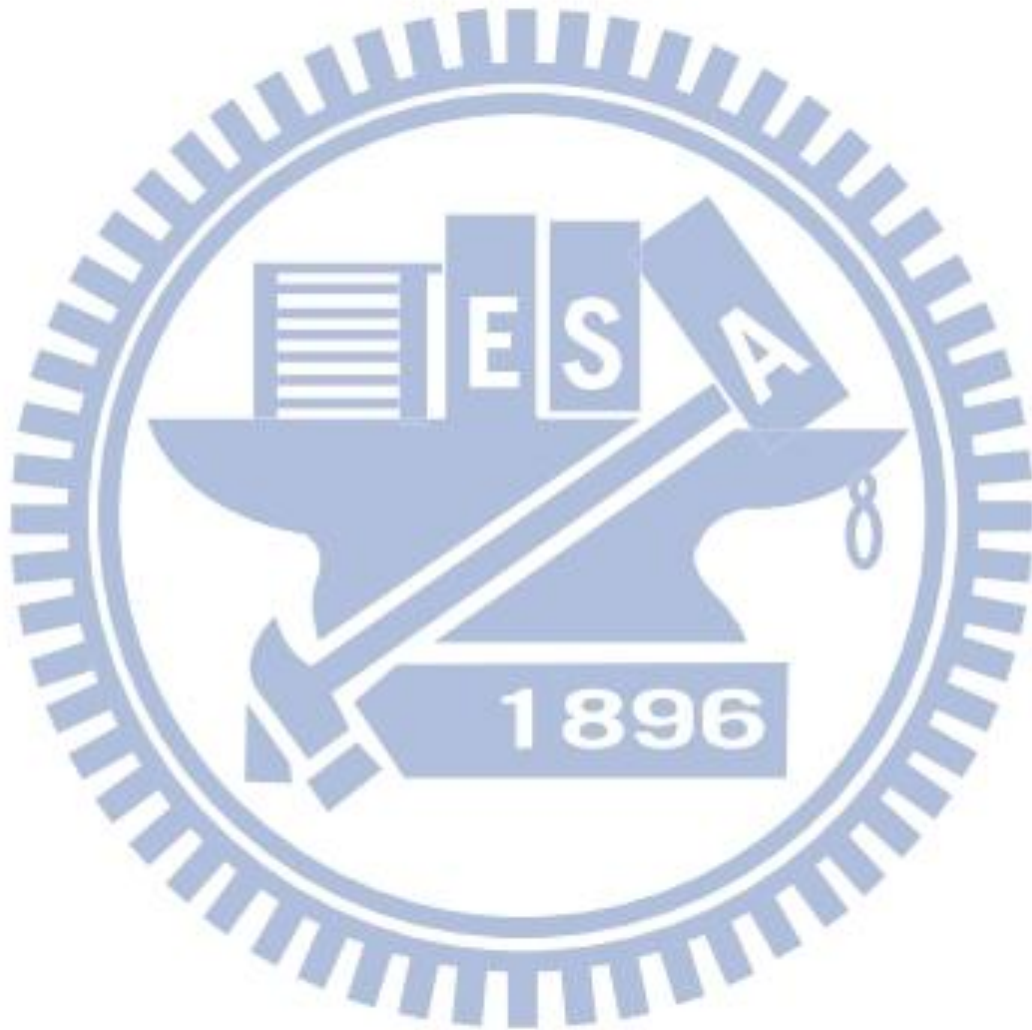
感謝我的家人，體諒我在這九年中冷落了他們，卻還是鼓勵我、支持我。感謝我的母親，沒有她含辛茹苦的培養我，我不可能有此機會來學習；感謝我的父親，讓我學習到敦厚待人的重要。感謝我的妻子，這段時間一個人操持家務、照顧小孩，讓我沒有後顧之憂。

# Table of Contents

摘要.....	i
Abstract.....	iii
誌謝.....	錯誤! 尚未定義書籤。
Table of Contents.....	iii
List of Tables.....	iiix
List of Figures.....	x
Chapter 1 Introduction.....	1
Chapter 2 Background and Related Work.....	6
2.1 Background and Terminology.....	6
2.2 Flash Translation Layers.....	8
2.2.1 BAST.....	13
2.2.2 FAST.....	13
2.2.3 AFTL.....	14
2.2.4 Superblock.....	15
2.2.5 LAST.....	15
2.3 Cleaning Policies.....	16
Chapter 3 The ROSE FTL.....	18
3.1 Architecture of ROSE.....	18
3.2 Entire-Block Writing.....	19
3.3 Merge-Aware Cleaning Policy.....	25
3.4 Free Page Reuse.....	33
3.5 Metadata Management in ROSE.....	37
Chapter 4 The HybridLog FTL.....	39
4.1 Architecture of HybridLog.....	40
4.2 Log-Style Writes.....	42
4.3 Spare Area Requirement of HybridLog.....	47
4.4 Reconstruction of Metadata.....	49
Chapter 5 Performance Evaluation.....	51
5.1 Experimental Setup and Traces.....	51
5.2 Performance Evaluation of ROSE.....	55
5.2.1 Effect of Entire-Block Writing.....	55
5.2.2 Effect of MARO Cleaning Policy.....	58
5.2.3 Effect of Free Page Reuse.....	64
5.2.4 Overall Performance of ROSE.....	66
5.3 Performance Evaluation of HybridLog.....	71
5.3.1 Effect of Log-Style Write.....	71

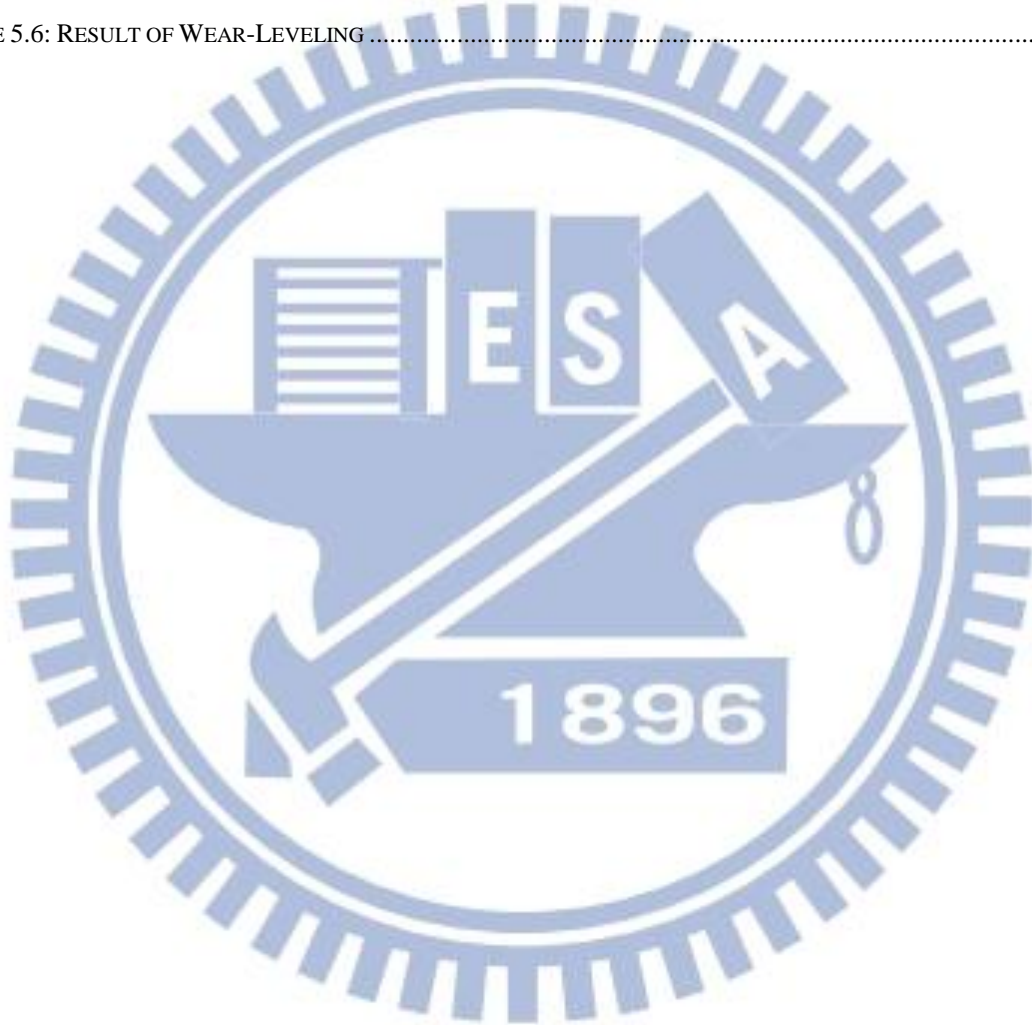


5.3.2 Cache Hit Ratio .....	74
5.3.3 Overall Performance of HybridLog .....	75
<b>Chapter 6 Conclusions and Future Work</b> .....	<b>77</b>
6.1 Conclusions .....	77
6.2 Future Work .....	78
<b>Bibliography</b> .....	<b>80</b>
<b>Vita</b> .....	<b>91</b>



# LIST OF TABLES

TABLE 4.1: COMMON FLASH MEMORY CONFIGURATION AND THE CORRESPONDING GROUP SIZES .....	48
TABLE 5.1: DEFAULT VALUES OF THE PARAMETERS .....	52
TABLE 5.2: TRACES .....	54
TABLE 5.3: PORTIONS OF ENTIRE-BLOCK WRITES .....	57
TABLE 5.4: STATISTICS OF FPR .....	66
TABLE 5.5: CLEANING COST WITH DIFFERENT LOG AREA SIZES (SECONDS) .....	69
TABLE 5.6: RESULT OF WEAR-LEVELING .....	70



# LIST OF FIGURES

FIGURE 2.1: THREE TYPES OF MERGE OPERATIONS .....	12
FIGURE 3.1: ARCHITECTURE OF ROSE .....	18
FIGURE 3.2: WRITE HANDLING UNDER FAST (A) AND EBW (B).....	23
FIGURE 3.3: AN EXAMPLE OF THE SWAP OPERATION.....	35
FIGURE 4.1: TRADITIONAL HAT ARCHITECTURE VS HYBRIDLOG ARCHITECTURE.....	40
FIGURE 4.2: TWO-LEVEL INTRA-BLOCK MAPPING .....	42
FIGURE 4.3: THE SPARE AREA FORMAT OF EACH WRITTEN PAGE.....	44
FIGURE 4.4: STEPS OF WRITING A PAGE.....	45
FIGURE 5.1: CLEANING COST OF THE METHODS FOR HANDLING SEQUENTIAL OVERWRITES (NORMALIZED TO THE CLEANING COST OF EBW).....	55
FIGURE 5.2: MISPREDICTION RATIOS AND COST OF PARTIALLY-FULL SW LOG BLOCK MERGES IN THE SEQUENTIALITY PREDICTION METHODS .....	57
FIGURE 5.3: CLEANING COST OF DIFFERENT CLEANING POLICIES (NORMALIZED TO THE CLEANING COST OF MARO) .....	58
FIGURE 5.4: CLEANING COST OF MARO AND LAST (NORMALIZED TO THE CLEANING COST OF MARO) .....	59
FIGURE 5.5: CLEANING COST WITH DIFFERENT $W_{AGE}$ .....	61
FIGURE 5.6: CLEANING COST WITH DIFFERENT A .....	61
FIGURE 5.7: CLEANING COST W/ AND W/O FPR (NORMALIZED TO THE CLEANING COST W/ FPR) .....	64
FIGURE 5.8: AVERAGE SPACE UTILIZATIONS W/ AND W/O FPR .....	65
FIGURE 5.9: CLEANING COST OF FAST, LAST AND ROSE (NORMALIZED TO THE CLEANING COST OF ROSE) .....	67
FIGURE 5.10: WRITE AMPLIFICATION RATIOS OF FAST, LAST AND ROSE.....	68
FIGURE 5.11: EFFECT OF THE NUMBER OF SEQUENTIAL WRITE LOG BLOCKS .....	68
FIGURE 5.12: NORMALIZED CLEANING COST W/ AND W/O LOG-STYLE WRITES .....	71
FIGURE 5.13: AVERAGE NUMBER OF DUMMY PAGES IN A DATA BLOCK .....	72
FIGURE 5.14: AVERAGE NUMBER OF PAGES COLLISION IN A DATA BLOCK WITH FREE PAGES.....	73
FIGURE 5.16: NORMALIZED CLEANING COST OF SUPERBLOCK, FAST, AND HYBRIDLOG.....	75

# Chapter 1

## Introduction

NAND flash memory is widely applied in computer and consumer electronic devices due to its small size, shock resistance, non-volatility and low power consumption. A NAND flash module is composed of a number of blocks, each of which is in turn composed of a number of pages. Typically, a NAND flash block contains 32 to 128 pages, and read/write operations are performed in units of a single page. In addition, a software component called Flash Translation Layer (FTL) is usually used to emulate a block device on top of the flash memory to support traditional disk-based file systems.

In contrast to RAM and disk, a page in the flash memory cannot be overwritten before being erased, and erase operations are performed in units of a whole block. Compared to the other flash operations, the erase operation is time-consuming. Moreover, the number of erase operations that can be done on a specific block is limited, usually between ten thousand and hundred thousand. To avoid erasing an entire block for each logical page overwrite, therefore, an FTL usually directs each page collision to a free physical page. The page containing the stale data is then reclaimed by a cleaning procedure. The cost of cleaning is a key factor to the performance of an FTL since cleaning involves time-consuming operations such as live page copying and block erasure. To

achieve high performance, an FTL should minimize the cleaning cost.

To locate each logical page, an FTL manages the mapping (i.e., address translation) between logical page numbers (LPNs) and physical page numbers (PPNs). Typically, the mapping can be done at two different granularities: page-level and block-level. Page level address translation (PAT) scheme maps each logical page to an individual physical page. Pages belonging to the same logical block can be mapped to different physical blocks. For a large NAND flash memory, such a fine-grained address translation scheme requires a large memory space to maintain the mapping table since each logical page has a corresponding entry in the table. In order to reduce the space requirement of the mapping table, block level address translation (BAT) scheme uses a more coarse-grained address translation approach that translates each logical block number (LBN) to a physical block number (PBN). Therefore, the number of entries in the mapping table can be greatly reduced. However, due to the block level address translation, BAT requires each logical page to be written only to its corresponding offset in a physical block, resulting in poor performance.

To combine the benefits of PAT and BAT, several FTLs based on the hybrid level address translation (HAT) scheme have been proposed [7, 18, 49, 51, 55, 57, 58, 59, 61, 74, 80, 84, 87, 88, 90]. In this scheme, most of the data are stored in data blocks managed via the BAT scheme. However, by storing hot pages (i.e., frequently-updated pages) in a limited number of log blocks, which

is managed by the PAT scheme, the HAT scheme can delay the erasure of some data blocks that are not fully occupied, increasing the space utilization. Moreover, the memory requirement is comparable to that of the BAT scheme since the pages managed via the PAT scheme are limited to a small number. Cleaning in HAT-based FTLs is done by reclaiming log blocks. When a log block needs to be reclaimed, it is merged with its corresponding data blocks.

In this thesis, we propose two HAT-based FTLs, ROSE and HybridLog. ROSE incorporates three novel techniques for reducing the cleaning cost. Firstly, it utilizes a technique called Entire-Block Writing (EBW) to prevent pages of an entire-block sequential write from being placed into multiple physical blocks, reducing the possibility of high-cost reclamation. Previous HAT-based FTLs achieve this by predicting sequentiality. However, mispredicting random or less-than-a-block writes as sequential writes leads to increased cleaning cost. EBW eliminates such misprediction, resulting in a lower cleaning cost. Secondly, ROSE uses a novel policy called Merge-Aware Round robin (MARO) to select a victim log block for reclamation when the log area has run out of its free space. In contrast to the previous cleaning policies that consider only the state of the candidates, MARO considers not only the state of the candidates (i.e., log blocks) but also the state of the data blocks that correspond to those candidates. Moreover, different from previous HAT-based FTLs, both the ages and the merge costs of the log blocks are considered at the same time in MARO. As

shown in the section 5.1, such consideration reduces the cleaning cost. Thirdly, ROSE utilizes a technique called Free Page Reuse (FPR) to increase the space utilization. FPR delays the erasure of a low-utilized data block and allows the free pages in that block to buffer further page overwrites, resulting in a lower cleaning cost.

HybridLog supports modern NAND flash memory by following the consecutive programming restriction in all the blocks and efficient using the spare area. With the development of flash memory, new restrictions are imposed on flash memory chips, and an FTL should follow these new restrictions so as to be applied on these modern chips. Specifically, a new programming (i.e., write) restriction called consecutive programming is imposed on most modern flash memories [13, 63], whereby pages have to be programmed in consecutive order (i.e., from lower-numbered pages towards higher-numbered pages) within a block. Moreover, Multiple Level Cell (MLC) NAND achieves lower cost by allowing multiple bits to be stored in a single cell. However, compared to Single Level Cell (SLC), MLC has a higher bit error rate and thus requires stronger ECC [17], which consumes more spare area space, preventing FTLs requiring large space of the spare area from being applied on it. To allow consecutive programming, HybridLog enables log-style writes to all the blocks (including the data blocks) in the flash memory. The log-style writes also helps to reduce the cleaning cost, improving the performance of the FTL. To support log-style

writes to all blocks, intra-block mapping information is stored in the spare area of each written page. Since only a small space is required in the spare area for the mapping information, many modern SLC/MLC flash memories can be supported.

Through simulation, we show the performance improvements of each of the three proposed techniques in ROSE. We also compare the performance of ROSE with FAST and LAST, two well-known and efficient HAT-based FTLs, under a variety of benchmarking and realistic workloads. The results show that ROSE outperforms the existing HAT-based FTLs by up to 47 times in terms of the cleaning cost. Due to the reduction on the cleaning cost, the flash write time is reduced by up to 1.6 times. The performance of HybridLog and two well-known FTLs are also compared. The performance results show that, HybridLog outperforms these two HAT-based FTLs by up to 17.8 times in terms of cleaning cost.

The rest of this thesis is organized as follows. The related efforts are described in Chapter 2. In Chapters 3 and 4, we describe the design and implementation of ROSE and HybridLog, respectively. The performance results are presented in Chapter 5. Finally, conclusions and future work are given in Chapter 6.



# Chapter 2

## Background and Related Work

### 2.1 Background and Terminology

An FTL maintains the state of all the pages in a flash storage. A page is *free* if the page has not been written after its last erasure. Free pages can be used to accommodate page writes. A free page becomes *live* after it has been written with user data. Since a live page cannot be overwritten before being erased, updating data in place is inefficient because each update should be preceded by a time-consuming erase operation. Thus, most FTLs handle page overwrites by adopting the out-of-place update mechanism, in which the new data are written to another free page and the live page that contains the old data becomes *dead*. Dead pages should be reclaimed by a cleaning procedure, which works as follows. First, one or more victim blocks are selected to be reclaimed according to a cleaning policy. Second, the live pages in the victim blocks are copied to free pages of other blocks. Finally, the victim blocks are erased. After the cleaning, all the pages in the selected blocks become free and can be used to satisfy future data writes. Cleaning is time consuming since it involves live page copying and block erasure. Therefore, the cost of cleaning is a key factor to the performance of an FTL. In this thesis, two metrics related to the cost of cleaning are used to measure

the performance of an FTL. The first one is the cleaning cost, which is defined as the time spent on the cleaning procedure resulting from the execution of a given workload. The second one is the Write Amplification Ratio (WAR) [29], which is defined as

$$\text{WAR} = (W + C)/W \quad (1)$$

where  $W$  and  $C$  represent the total request write time and the cleaning cost of the workload, respectively. The ratio 1.5 means that the time spent on cleaning is half of the total request write time of the given workload.

An FTL may erase a block that still contains free pages, which wastes the free pages. The free pages could have been used to buffer more writes and this waste could increase the cleaning cost. We define *space utilization* as the ratio of the number of occupied (i.e., nonfree) pages in a block to the total number of pages per block when the block is going to be erased. The value is 100% if a to-be-erased block contains no free pages. Increasing space utilization usually leads to reduction of the cleaning cost.

With the development of flash memory, new restrictions are imposed on flash memory chips, and an FTL should follow these new restrictions so as to be applied on these modern chips. Specifically, a new programming (i.e., write) restriction called consecutive programming is imposed on most modern flash memories [13, 63], whereby pages have to be programmed in consecutive order (i.e., from lower-numbered pages towards higher-numbered pages) within a block.

Moreover, Multiple Level Cell (MLC) NAND achieves lower cost by allowing multiple bits to be stored in a single cell. However, compared to Single Level Cell (SLC), MLC has a higher bit error rate and thus requires stronger ECC, which consumes more spare area space, preventing FTLs requiring large space of the spare area from being applied on it.

## 2.2 Flash Translation Layers

An FTL emulates a block device on top of flash memory to support traditional disk-based file systems. Typically, a request issued from a file system consists of a single or multiple adjacent sectors. In a flash storage system, the sector numbers are translated into logical page numbers and the translation is usually independent of the FTLs. In this thesis, the sizes of a sector and a page are 512 bytes and 2 Kbytes, respectively, and therefore, LPNs can be obtained by dividing the sector numbers by 4. Such translation can be regarded as a preprocessing task before the invocation of an FTL. An FTL hence treats each request as a number of adjacent logical pages and focuses on the address translation between LPNs and PPNs.

The address translation can be done at page level (i.e., the PAT scheme) or block level (i.e., the BAT scheme). PAT-based FTLs [4, 6, 9, 10, 13, 27, 30, 31, 32, 36, 47, 54, 60, 65, 66, 67, 75, 89] directly translate each LPN to a PPN and use the out-of-place update mechanism to handle page overwrites. In this scheme,

a logical page can be written to any physical page and cleaning is needed only when there are almost no free pages in the storage. Therefore, the cleaning cost is relatively small. However, this scheme requires a large memory space for a large-sized flash memory. For example, for an 8-Gbyte flash memory with page size 2 Kbytes, four million entries (i.e., 16 Mbytes if the size of each entry is 4 bytes) are needed in the mapping table.

Recently, several RAM-space-efficient PAT-based FTLs address this problem by storing the mapping information in the flash memory and caching the recently used information in RAM [25]. Cleaning in PAT-based FTLs is done by reclaiming blocks (i.e., copying live pages in victim blocks to blocks with free pages and then erasing victim blocks). After the reclamation, the erased blocks can be used to accommodate future writes.

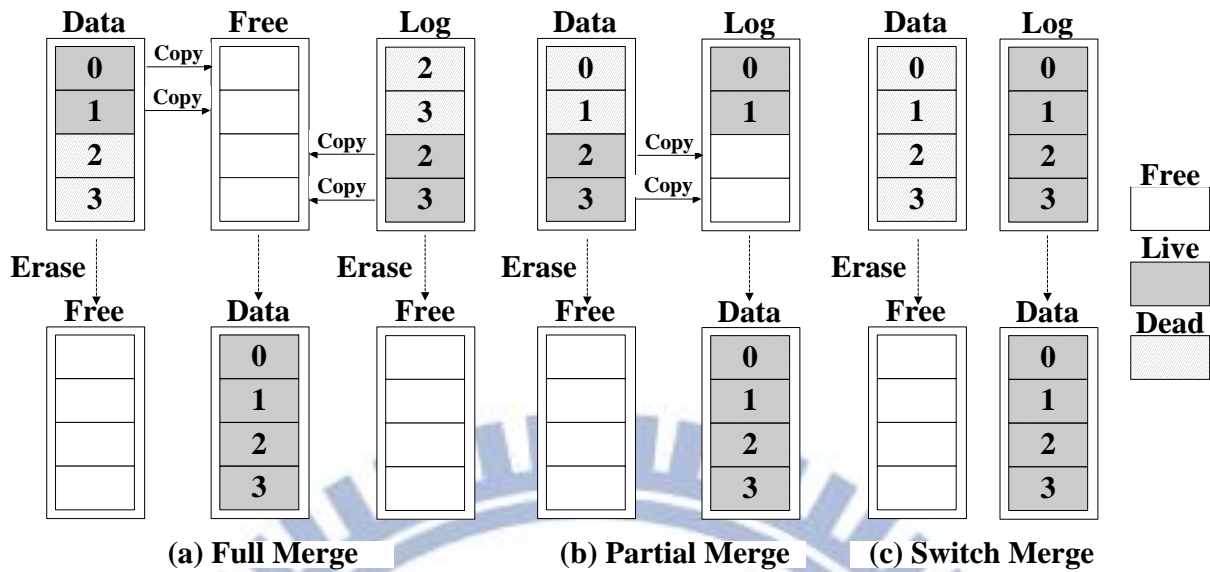
After a PAT-based FTL has selected a victim block for cleaning, it has to identify the live pages of the victim block. Querying/updating the mapping information of the live pages is needed during cleaning. If the FTL stores the page state and mapping information of each page in RAM, query/update of the mapping information can just be done in RAM. However, as mentioned above, a large RAM space would be required for a large sized flash memory. In memory-constrained consumer storages such as SDs or UFDs, mapping information of a PAT-based FTL can only be stored in the flash memory (and cached in RAM). Thus, after a victim block has been selected, extra flash memory read/write operations need to be performed to identify the live pages in

the victim block and to locate the physical locations of the mapping information of the live pages. Note that, victim blocks are cold blocks and thus their information is seldom cached in RAM. If there are many live pages in the victim block (i.e., high storage utilization) and the mapping information of the live pages are stored in many different mapping pages, many flash memory reads/writes are required for querying and updating the mapping information.

BAT-based FTLs achieve lower RAM consumption for the mapping information by using coarse-grained mapping [3, 19, 50, 70, 71, 72]. In the BAT scheme, each logical block has a corresponding data block to accommodate page writes to that logical block. The LPN is divided by the number of pages in a block to get the logical block number (i.e., the quotient) and the page offset (i.e., the remainder). The former is used to index the mapping table to get the physical address of the data block, and the latter is used to locate the target page in the data block. If the target page is live (i.e., page collision), in-place update is used. That is, the data block, say  $D$ , is reclaimed by copying all the up-to-date data of the logical block from  $D$  and the write request to a free block  $F$  and then erasing  $D$ . After the reclamation,  $F$  is used as the new data block. This reclamation is needed due to the limitation that each logical page can be written only to a fixed offset of a physical block. Such limitation usually leads to low space utilization since a significant amount of free pages might exist in the to-be-erased blocks (i.e., the block  $D$  mentioned above). For example, frequently updating a small number of pages in a logical block could easily lead to low space utilization of

the corresponding data block. Erasing these free pages, instead of using them to buffer page writes, increases the frequency of block reclamation. Moreover, in modern flash memory the consecutive programming restriction, if the target page is not consecutive to the last written page of the data block, dummy pages has to be written between the last written page and the target page (i.e., page padding).

Several hybrid-mapped FTLs have been proposed to achieve performance superior to block-mapped FTLs, while retaining the small size of the mapping information. In these FTLs, most of the blocks (i.e., data blocks) are managed via the BAT scheme. However, by managing a small number of log blocks via the PAT scheme to accommodate frequently updated pages, the space utilization is increased. HAT also utilizes the out-of-place update mechanism. Page writes that cannot be accommodated by the data blocks are satisfied by the log blocks, and the pages containing the old data become dead. Since the blocks managed by PAT are limited to a small number, the memory requirement of HAT is comparable to that of BAT. Cleaning in HAT-based FTLs is done by reclaiming log blocks. When a log block needs to be reclaimed, it is *merged* with its corresponding data blocks. After the merge, a free log block is obtained to accommodate future writes.



**Figure 2.1: Three types of merge operations**

As shown in Figure 2.1, three types of merge could occur depending on the status of the data and the log blocks. In Figure 2.1(a), a *full merge* can be done by copying the live pages either from the data block or the log block to a free block  $F$ , erasing both the data and log blocks, and then using  $F$  as the new data block. In Figure 2.1(b), a *partial merge* can be done by copying the live pages in the data block to the free space of the log block, erasing the data block, and then prompting the log block as the new data block. In Figure 2.1(c), all the up-to-date data were written in the log block sequentially and thus the merge operation can be done simply by switching the roles of the log and data blocks and erasing the original data block, which is called *switch merge*. Of the three types of merge operations, the switch merge has the lowest cost while the full merge results in the highest cost. Note, in some HAT-based FTLs, a log block might correspond

to multiple data blocks (i.e., the log block accommodates page overwrites belonging to multiple logical blocks) and thus reclaiming the log block requires multiple merges, each of which corresponds to a data block. In this thesis, we define the page density of a log block as the number of data blocks corresponding to it. In the following, several well-known HAT-based FTLs are described.

### **2.2.1 BAST**

BAST [45] allows each data block to have at most one dynamically allocated log block accommodating overwrites of that data block. When an allocated log block cannot accommodate the current write, it is reclaimed by merging with its data block. Moreover, if all the log blocks have been allocated, a further log block allocation would cause one of the allocated log block to be reclaimed. This FTL suffers from the log block thrashing problem [53] (i.e., frequent erasure of log blocks with low utilization) if the number of frequently updated blocks accommodating small random writes is larger than the number of log blocks.

### **2.2.2 FAST**

FAST [53] eliminates the above problem of BAST by using fully associative log blocks. That is, a log block can accommodate page overwrites of any data blocks. In FAST, one special log block called the SW log block is reserved for sequential overwrites and the other log blocks called RW log blocks are for



random overwrites. The SW log block corresponds to a single data block. If a sequential overwrite cannot be satisfied by the current SW log block, the SW log block is merged with its corresponding data block to get a free SW log block. A RW log block can correspond to multiple data blocks. If a random overwrite cannot be satisfied by the RW log blocks because all the RW log blocks are fully occupied, FAST selects a victim RW log block in a round-robin (RR) fashion and merges the victim with its corresponding data blocks. FAST may still erase low-utilized blocks. For example, a victim log block may be merged with multiple low-utilized data blocks.

### **2.2.3 AFTL**

AFTL [83] allows each data block to have at most one log block for satisfying overwrites of that data block. When a log block becomes full, its live pages are regarded as hot and the mapping information corresponding to these live pages is inserted into a page-level mapping table, which may cause the eviction of the mapping information of some other hot pages due to the limited memory space reserved for the mapping table. The eviction is based on LRU and each selected victim hot page will be migrated back to the corresponding data block or log block. Since each log block corresponds to a single data block, AFTL also has the block thrashing problem.

## 2.2.4 Superblock

Superblock [35, 37] allows a group of adjacent logical blocks to share a number of log blocks so as to increase the space utilization while keeping the page density of the log blocks low. The limitation of Superblock is that it stores the page-level mapping information of a block group in the spare area, which reduces the space for Error Correction Code (ECC). For example, Superblock requires 44 bytes of each per-page spare area, whose typical size is 64 bytes, on flash memory modules with 64 pages per block. As a consequence, only a 20-byte space is left for ECC, reducing the quality of the ECC. This problem gets worse for NAND flash modules with even more pages per block (e.g., 128) [78]. Based on the block grouping concept of Superblock, Park et al. proposed an offline method [64] to determine the values of the block group size and the maximum number of log blocks allocated for a block group, which can be applied on systems with fixed workloads.

## 2.2.5 LAST

Similar to FAST, the LAST FTL [52] serves sequential and random overwrites by using different log blocks. In LAST, multiple SW log blocks are used to satisfy concurrent write streams, and the set of the RW log blocks is divided into hot and cold blocks to reduce the merge cost. Although multiple SW log blocks are utilized, LAST may still erase low-utilized blocks when

less-than-a-block sequential writes corresponding to a significant number of logical blocks are presented.

## 2.3 Cleaning Policies

A number of cleaning policies [8, 9, 11, 16, 21, 23] that consider reclamation efficiency, such as greedy [77], cost-benefit [47], Cost-Age-Time (CAT) [15], and CICL [43], have been proposed. The greedy policy selects the block with the minimum number of live pages as the victim in order to minimize the cost of page copying. The cost-benefit policy selects the block with the maximum value of the following formula as the victim:

$$\frac{age * (1 - u)}{2u}$$

where  $u$  represents the ratio of number of live pages to the total number of pages in the candidate block, and  $age$  denotes the time since the last modification of the block. In the formula,  $(1 - u)$  and  $2u$  represent the benefit and cost of the reclamation, respectively. The  $age$  is considered to avoid reclaiming young blocks, whose pages are likely to be invalidated in the near future.

The CAT and CICL policies consider both reclamation efficiency and wear leveling [8, 12, 34, 68, 81, 82]. CAT selects the block with the minimum value of the following formula as the victim:

$$\frac{u * e}{age * (1 - u)}$$

where  $e$  and  $age$  represent the number of times the candidate block has been erased and the elapsed time since the last reclamation of candidate block, respectively. CICL selects the block with the minimum value of the following formula as the victim:

$$\lambda * e / (1 + e_{max}) + (1 - \lambda) * v,$$

where  $0 < \lambda < 1$ . In this formula,  $e_{max}$  denotes the maximum value of  $e$  among all the candidate blocks, and  $v$  represents the ratio of number of live pages to the total number of non-free pages in the candidate block. As shown in the formula, CICL selects the victim based mainly on wear leveling when  $\lambda$  is close to 1, which happens when the difference between the maximum and the minimum numbers of  $e$  among the candidates is large. On the contrary, it selects the victim mainly based on the reclamation efficiency when  $\lambda$  is close to 0, which happens when the difference between the maximum and the minimum numbers of  $e$  among the candidates is small.

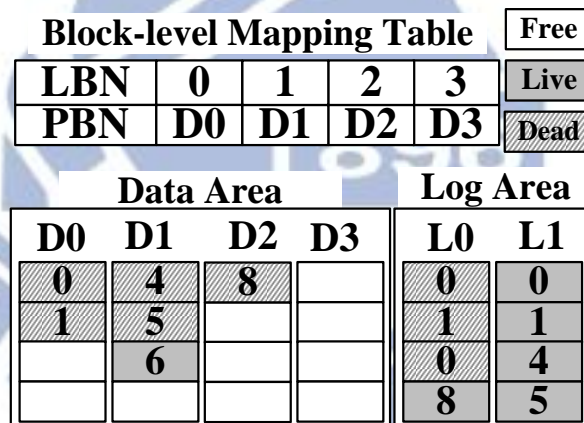
Basically, these policies are used in PAT-based FTLs. They select a victim block for reclamation based on the condition of the candidate block, which is not sufficient for log block reclamation in HAT-based FTLs. Specifically, log block reclamation involves merging the victim log block with its corresponding data blocks, which was not considered in these policies.

# Chapter 3

## The ROSE FTL

In this Chapter, we describe the ROSE FTL. ROSE incorporates three novel techniques to reduce the cleaning cost, namely, entire-block writing, merge-aware round robin cleaning policy, and free page reuse. In the following, the architecture of ROSE is first described, which is followed by the description of the techniques used in ROSE.

### 3.1 Architecture of ROSE



**Figure 3.1: Architecture of ROSE**

As shown in Figure 3.1, ROSE utilizes the HAT scheme, which divides the flash memory into two areas, a large *data area* managed by BAT and a small *log area* managed by PAT. The former contains a set of data blocks while the latter contains a set of log blocks. Each logical block has a corresponding data block

for accommodating the writes to that logical block. For each write to a logical page, ROSE writes the data to the target physical page in the data block if the physical page is free. If the write cannot be accommodated by the data block (i.e., the target page is not free), the data are written to the log area in the log order. In contrast to FAST and LAST, ROSE does not have special log blocks for storing sequential (over)writes. Instead, it relies on the entire-block writing technique mentioned in Section 3.2 to handle sequential writes. When the log area has run out of free pages, a victim log block is selected to be merged with its corresponding data blocks. That is, for each live page  $p$  in the victim log block, the live pages belonging to the same logical block as  $p$  are copied from the corresponding data block and log blocks (including the victim log block) to a new block, which serves as the new data block for the logical block. After the page copying, the victim log block and the corresponding data blocks become obsolete and can be erased.

## 3.2 Entire-Block Writing

In some HAT-based FTLs, a log block may correspond to multiple logical blocks. This leads to a higher cleaning cost for reclaiming the log block since merging the block with all its corresponding data blocks is required. To reduce the possibility of such high-cost reclamation, several HAT-based FTLs such as FAST and LAST reserve one or more log blocks to accommodate sequential

overwrites. Each of such log blocks, called a sequential write log block, corresponds to a single logical block. In an SW log block, each logical page is written to its corresponding offset (i.e., the data in  $i$ th logical page of a logical block are written to the  $i$ th physical page of the SW log block), hoping that the log block can be promoted as a new data block later in a switch merge.

Under the SW log block approach, with a given overwrite request  $R$  that contains pages belonging to a logical block  $B$ , the FTLs have to predict whether or not the other pages belonging to  $B$  will be overwritten in the near future. If they will, all the pages belonging to  $B$  should be placed in the same log block so that the reclamation of this log block can be done in a switch merge (i.e., erasing the data block corresponding to  $B$  and promoting the log block as the new data block). Therefore, if the other pages are predicted to be overwritten in the near future,  $R$  would be served by an SW log block.

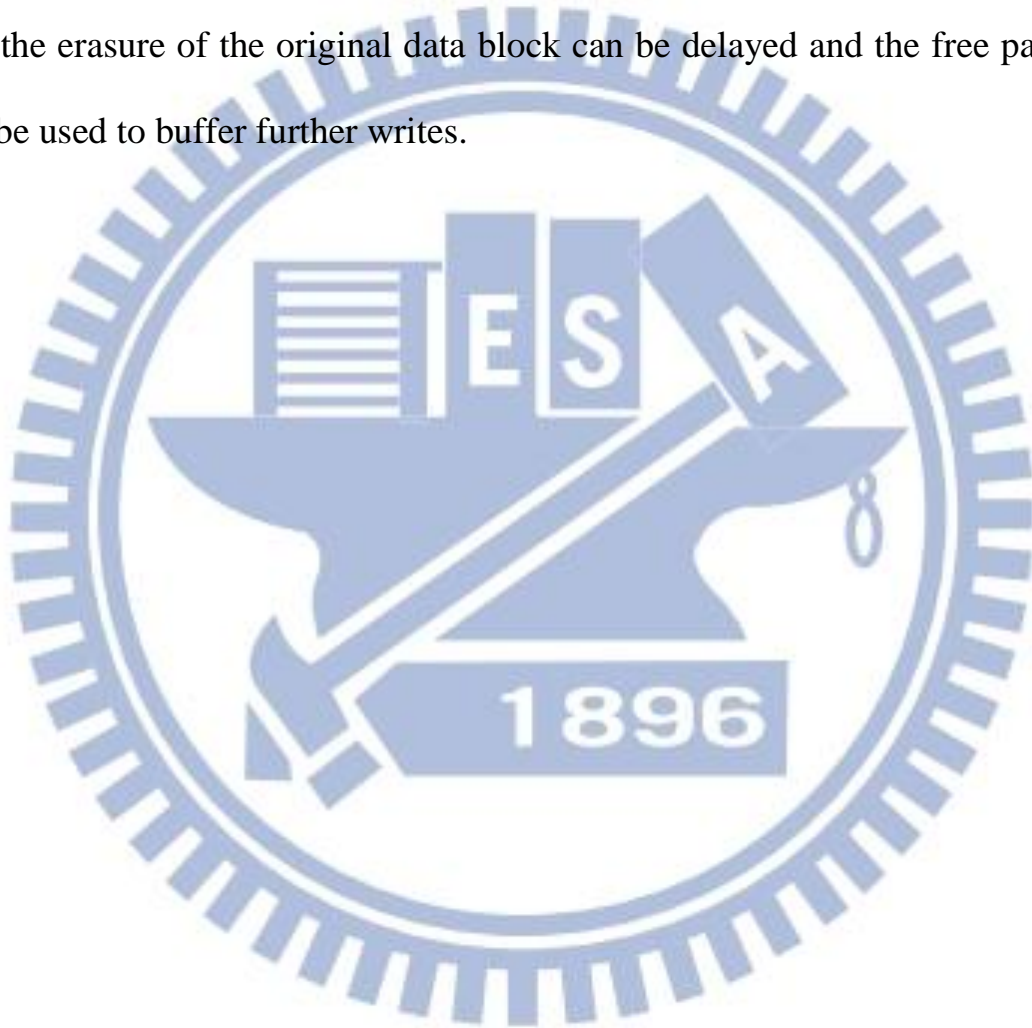
The main problem of the SW log block approach is that frequent misprediction would cause frequent block erasure. Specifically, FAST uses a single SW log block and serves an overwrite to a logical page by the SW log block if the page is the first page of a logical block or the page corresponds to the first free page of the SW log block. That is, FAST predicts that an overwrite to the first page of a logical block will be followed by overwrites to the other pages in that block. As a consequence, in FAST, a workload that repeatedly overwrites the first page of a logical block may cause the (partially full) SW log block to be repeatedly merged with its corresponding data block. LAST uses a small number

of SW log blocks and serves a write request via an SW log block if the size of the request is equal to or larger than a predefined threshold (e.g., 4 Kbytes). As a consequence, semisequential write requests (i.e., requests with sizes smaller than the block size but larger than or equal to the threshold) corresponding to a large number of logical blocks could lead to a large number of merges between partially full SW log blocks and their corresponding data blocks.

ROSE utilizes a fundamentally different approach for handling sequential overwrites. Specifically, it adopts a technique called Entire-Block Writing (EBW), which detects sequentiality in the current write request instead of predicting sequentiality. Therefore, misprediction would never occur. EBW detects entire-block overwrites and utilizes *free* blocks, instead of SW log blocks, to serve those writes. With EBW, each write request is divided into a number of page-level subrequests and block-level ones. For example, on a NAND flash module with 64-page blocks, a write request with 130 pages starting from LPN 0 will be divided into two block-level subrequests (i.e., for LPNs 0 to 63 and LPNs 64 to 127) and two page-level subrequests (i.e., for LPNs 128 and 129). For each page-level subrequest, the data are written to the log area in the log order if the subrequest is an overwrite. However, each block-level subrequest is served by a free block. Specifically, given a block-level subrequest that corresponds to logical block  $B$ , the data are written to the data block corresponding to  $B$  if the data block is originally free. Otherwise, the subrequest overwrites one or more logical pages belonging to  $B$  and thus EBW uses another free block, say  $F$ , to



serve this subrequest. After the subrequest has been served,  $F$  becomes the new data block corresponding to  $B$ , and the original data block (which contains no live pages) can be erased if cleaning is needed. Note that, such a free block is always available since HAT-based FTLs always reserve at least one free block for buffering the result of a full merge. With the FPR technique mentioned in Section 3.3, the erasure of the original data block can be delayed and the free pages in it can be used to buffer further writes.



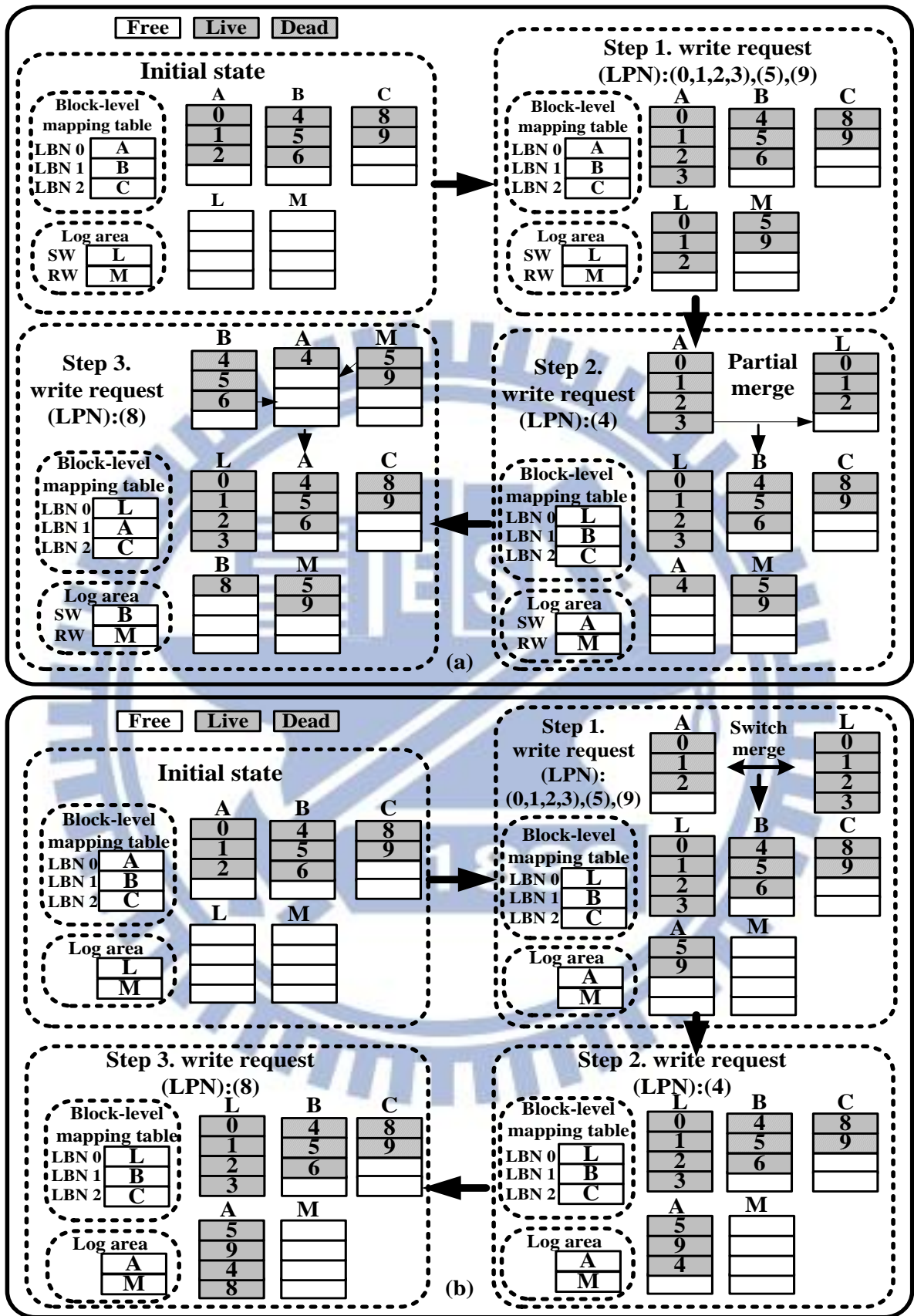


Figure 3.2: Write handling under FAST (a) and EBW (b)

Figure 3.2 illustrates an example showing the difference between EBW and the SW log block approach in FAST. Assume that the flash memory consists of three data blocks and two log blocks, with each block containing four pages, and initially blocks *A*, *B*, and *C* are the data blocks of logical blocks 0, 1, and 2, respectively. Figures 3.2(a) and 3.2(b) illustrate the handling of the page write sequence (0, 1, 2, 3, 5, 9, 4, 8) under the SW log block approach in FAST and the EBW approach, respectively. In Figure 3.2(a), pages 0, 1, and 2 are written to the SW log block *L* since page 0 is the first page of a logical block and pages 1 and 2 correspond to the first two free pages of the SW log block after the write of page 0. Page 3 can be served by the data block *A*, and pages 5 and 9 are served by the RW log block *M*. The same as page 0, page 4 also needs to be written to the SW log block since it is the first page of a logical block. This requires merging *L* with *A*. After the merge, *L* becomes the new data block. The old data block *A* is erased and becomes the new SW log block to accommodate page 4. Similarly, serving page 8 requires another merge. After the merge, *A* becomes the new data block. The old data block *B* is erased and becomes the new SW log block to accommodate page 8. Therefore, the cleaning cost under FAST involves erasing two blocks and copying three pages. In Figure 3.2(b), pages 0, 1, 2, 3 are served by a free block, say *L*, which then becomes the new data block of logical block 0 via a switch merge. The old data block *A* is erased. Since writes to pages 5, 9, 4, 8 are not entire-block overwrites, these pages are written to a log block, say *A*.

Therefore, the cleaning cost under EBW is only the erasure of one block.

As a result, EBW prevents the pages of an entire-block write from being placed into multiple log blocks, reducing the possibility of high-cost reclamation and achieving the goal of SW log blocks without using them. Moreover, since there is no need to predict whether or not a request should be served by an SW log block, log block reclamation resulting from misprediction is eliminated.

The effectiveness of EBW depends on the frequency of entire-block writes. Using MLC flash memory and multi-channel architectures in SSDs might lead to increased block size and reduced frequency of entire-block writes. Nevertheless, modern operating systems such as Windows 8 and new versions of Linux tend to issue very large write requests (e.g., larger than 2 Mbytes). Moreover, several flash-aware cache management techniques [1, 26, 28, 38, 41] such as FAB [33], BPLRU [42], and CFLRU [69] tend to produce entire-block writes. These help EBW to remain effective in modern computing systems.

### **3.3 Merge-Aware Cleaning Policy**

As described in Section 2.3, many cleaning policies such as greedy, cost-benefit, and CAT, select a victim block based on the condition of the candidate blocks. For example, the greedy policy selects the block with the minimum number of live pages as the victim in order to minimize the cost of page copying. Although these policies perform well in PAT-based FTLs, they are

not suitable for HAT-based FTLs since log block reclamation in a HAT-based FTL is different from block reclamation in a PAT-based FTL. Specifically, the former involves merging with data blocks, which was not considered in the above policies. For example, the cost of reclaiming a log block with three live pages is not necessarily lower than that of reclaiming another log block with six live pages since the former may involve copying more pages from the corresponding data blocks and erasing more blocks.

In this thesis, we propose a new cleaning policy called MARO for a HAT-based FTL. Similar to round robin, which is used in FAST, MARO prevents reclaiming young blocks. According to temporal locality, live pages in the young blocks might be invalidated in the near future. Therefore, delaying the reclamation of a young block will likely lead to less page copying and block erasing overhead. Moreover, when reclaiming a log block, MARO considers the merge cost, which is related not only to the state of the log blocks but also to the state of the data blocks corresponding to those log blocks.

In MARO, dead blocks will first be selected as the victims. If no such blocks are available, MARO selects an old block that has a low merge cost as the victim. Specifically, it selects the log block with the maximum value of *score*, where the *score* of a log block  $L_i$  can be expressed as

$$score(i) = age(i) * W_{age} - cost(i) \quad (2)$$

In (2),  $age(i)$  represents the elapsed time since the last reclamation of log

block  $L_i$ ,  $W_{age}$  denotes the weight of the block age, and  $cost(i)$  denotes the merge cost of  $L_i$ . As mentioned before, a log block  $L_i$  may correspond to multiple data blocks and thus reclaiming  $L_i$  involves merging it with all its corresponding data blocks. For ease of computation, we assume a full merge is performed between  $L_i$  and each of its data blocks. For each data block  $D_j$ , two sets of live pages should be copied to a new data block  $D_j'$ , which replaces the role of  $D_j$  after the merge. The first set is the live pages of  $D_j$ , and the second set is the live pages that correspond to the dead pages of  $D_j$ . The second set of live pages is stored in the log area (including the victim log block  $L_i$ ). After merging with all the corresponding data blocks, the victim log block and the data blocks are erased. Therefore, the merge cost can be expressed as

$$cost(i) = \sum_1^n (lpc_j + dpc_j) * C_{pc} + (n + 1) * C_{erase} \quad (3)$$

In (3),  $n$  denotes the number of data blocks corresponding to  $L_i$ . The  $lpc_j$  and  $dpc_j$  denote the numbers of live pages and dead pages in data block  $D_j$ , where  $1 \leq j \leq n$ , respectively. Finally,  $C_{pc}$  and  $C_{erase}$  denote the cost of copying a page and erasing a block, respectively. Note that in (3), the first part represents the cost of page copying and the second part represents the cost of block erasure. Since both page copying and block erasure can be done in either the foreground or the background, we consider the overall cost instead of dividing the cost into foreground and background parts.

From (3), the merge cost is related to the page density of the log block (i.e., the value of  $n$ ). Higher page density tends to result in higher merge cost. Moreover, the cost is also related to the state (i.e., number of live/dead pages) of data blocks corresponding to the log block. A larger number of live pages in the data blocks lead to higher merge cost. Similarly, since each dead page in the data block has a corresponding live page in the log area, which also needs to be copied to the new data block, a larger number of dead pages also lead to higher merge cost. In summary, the merge cost is related to the state of the log block and the corresponding data blocks, and the cost of block erasure and page copying. As shown in Figure 5.3, considering the merge cost in a cleaning policy results in more efficient reclamation than the previous policies that consider only the state of the log blocks such as CAT.

Although a dead page in the data block also results in the copying of a page, the *net cost* of a dead page is lower than that of a live page. This is because page copying corresponding to a dead page does have some benefits. Specifically, it causes the invalidation of a log page, say  $p$ , and hence reduces the cost of reclaiming the log block that contains  $p$  in the future. For example, it might reduce the page density of that log block so that the reclamation of that log block in the future will involve less erase operations. Therefore, (3) is modified as

$$cost(i) = \sum_1^n (lpc_j + \alpha * dpc_j) * C_{pc} + (n + 1) * C_{erase}, \quad (4)$$

where  $\alpha < 1$  and it denotes the ratio of the net cost of a dead page, when compared with the net cost of a live page. Traditional merge cost evaluation approach, in which  $\alpha$  is always equal to 1, completely ignores the benefit of the dead pages. On the contrary, MARO respects the benefit and hence always setting  $\alpha$  as smaller than 1. Substituting (4) to (2) yields

$$score(i) = age(i) * W_{age} - \sum_1^n (lpc_j + \alpha * dpc_j) * C_{pc} - (n + 1) * C_{erase}, \quad (5)$$

where  $\alpha < 1$ . In (5),  $W_{age}$  and  $\alpha$  are controlled by the system designers. A large value of  $W_{age}$  leads to a policy similar to round robin, and a zero value of  $W_{age}$  leads to a purely cost-driven policy. The parameter  $\alpha$  determines whether the benefit of the dead pages is regarded as significant. Other variables in (5) can be obtained from the runtime information or the datasheet of the flash device.

The differences between MARO and previous log block reclamation policies in HAT-based FTLs are as follows. First, MARO considers the age and the merge cost of a log block at the same time. FAST considers only the block age and thus could reclaim high-cost log blocks. LAST considers the merge cost. However, the block age is not considered when selecting a victim according to the merge cost. As mentioned above, live pages in the young blocks might be invalidated in the near future and thus delaying the reclamation of these blocks, as the MARO does, will likely lead to lower cleaning cost. In Section 5.1, we demonstrate that



lower cleaning cost can be achieved by considering both factors at the same time. Second, MARO uses a different merge cost evaluation approach that treats the net cost of copying a page corresponding to a dead page in a data block as lower than that of copying a page corresponding to a live page in a data block, and it uses the parameter  $\alpha$  to control the ratio of the former to the latter. In the previous approach such as that used in LAST, the two types of cost are treated as equal since the benefit of copying a page corresponding to a dead page in a data block is totally ignored. In Figure 5.6, we show that respecting the benefit and setting  $\alpha$  smaller than 1 could result in the reduction of the cleaning cost.

Although MARO tends to select an old block as the victim, which is helpful in wearing the log blocks evenly, global wear leveling that considers both the log blocks and the data blocks is beyond the scope of MARO. To achieve global wear leveling, an erased block is not used to serve the incoming write directly. Instead, it is returned to the free block pool of the storage, and the free block with the minimum erase count in the pool is used to serve the write. The erase count of a block represents the number of times the block has been erased. Moreover, a simple wear-leveling technique proposed in eNvy [84] is utilized. Assume that the blocks with the minimum and maximum erase counts are  $C$  and  $H$ , respectively. If the difference between the erase counts of  $C$  and  $H$  is larger than a threshold  $T_{hc}$ , the data of  $C$  and  $H$  are swapped.

Note that, the computation overhead needs to be addressed for the implementation of MARO. Instead of recomputing scores for all the log blocks

every time when cleaning is required, we amortize the score computation and the search of the maximum score over multiple flash memory operations. We maintain  $S_{data}(j)$ , the *sub-score* of each data block  $D_j$ , expressed as

$$S_{data}(j) = -(lpc_j + \alpha * dpc_j) * C_{pc}, \quad (6)$$

which is a part of (5) related to the state of a data block. When a page write causes the association between a data block  $D$  and a log block  $L$  (i.e., the write causes  $L$  to correspond to  $D$ ), the sub-score of  $D$  is added to the score of  $L$ . Each time when a page write changes the state of  $D$ , the sub-score of  $D$  is updated and the scores of the log blocks corresponding to  $D$  are also updated. Finally, when a page write causes the disassociation between  $D$  and  $L$ , the sub-score of  $D$  is subtracted from the score of  $L$ . Similarly, according to (5), the score of a log block  $L$  is added/subtracted by  $(-1 * C_{erase})$  each time when the page density of  $L$  is increased/decreased by 1.

Upon the first page write to a log block, the score of the log block is initialized as  $H$  minus  $C_{erase}$ , where  $H$  represents the initial block age multiplied by  $W_{age}$  and  $C_{erase}$  reflects the cost of erasing the log block. As mentioned before, the block age represents the elapsed time since the last reclamation of a log block, which can be implemented by using 0 as the initial block age and adding the ages of all the log blocks other than the erased log block by 1 when a log block is erased. However, this requires updating a large number of scores upon block erasure. Therefore, when a log block is erased, we keep the ages of the other log

blocks intact and subtract the initial block age by 1. Consequently,  $H$  is decreased by  $W_{age}$  each time when a log block is erased.

Searching the maximum scores efficiently is also an implementation issue. To reduce the search time during the cleaning procedure, the log area is divided into multiple clusters, each of which is in turn divided into multiple 8-block segments. Each time the score of a log block is updated, the maximum score in the corresponding cluster is searched and recorded. Therefore, during the cleaning procedure, only the maximum scores of the clusters need to be compared. To speed up the search time further, hardware circuits were implemented for the search of the intra-cluster maximum scores and the maximum score among the clusters.

From the above description, amortizing the score computation eliminates the multiplication operations. In addition, although  $\alpha$  is a floating point value, floating point operations can be avoided by multiplying (5) by a constant so that all the terms in (5) become integers. The multiplication can be done offline. As a result, the implementation of MARO does not require the SSD controller to perform multiplications or floating-point operations, suitable for current integer-processor based SSD controllers.

With the above amortization method, the worst case execution time of a page write occurs when the write changes the state of a data block associated with  $K$  log blocks, where  $K$  is equal to the number of pages per block. In this case,  $K$  scores need to be updated and the maximum scores in the corresponding clusters

need to be searched and recorded, which take  $O(K*N_{SC})$  time where  $N_{SC}$  is the number of segments in a cluster. Since  $K$  and  $N_{SC}$  are both constants, the time complexity of a page write is  $O(1)$ . The time complexity of the cleaning procedure is  $O(N_L)$ , where  $N_L$  is the number of log blocks in the storage. Such complexity is the same as LAST. In addition, the space complexity of ROSE is  $O(N_D+N_L)$ , where  $N_D$  is the number of data blocks in the storage, the same as many HAT-based FTLs such as FAST and LAST.

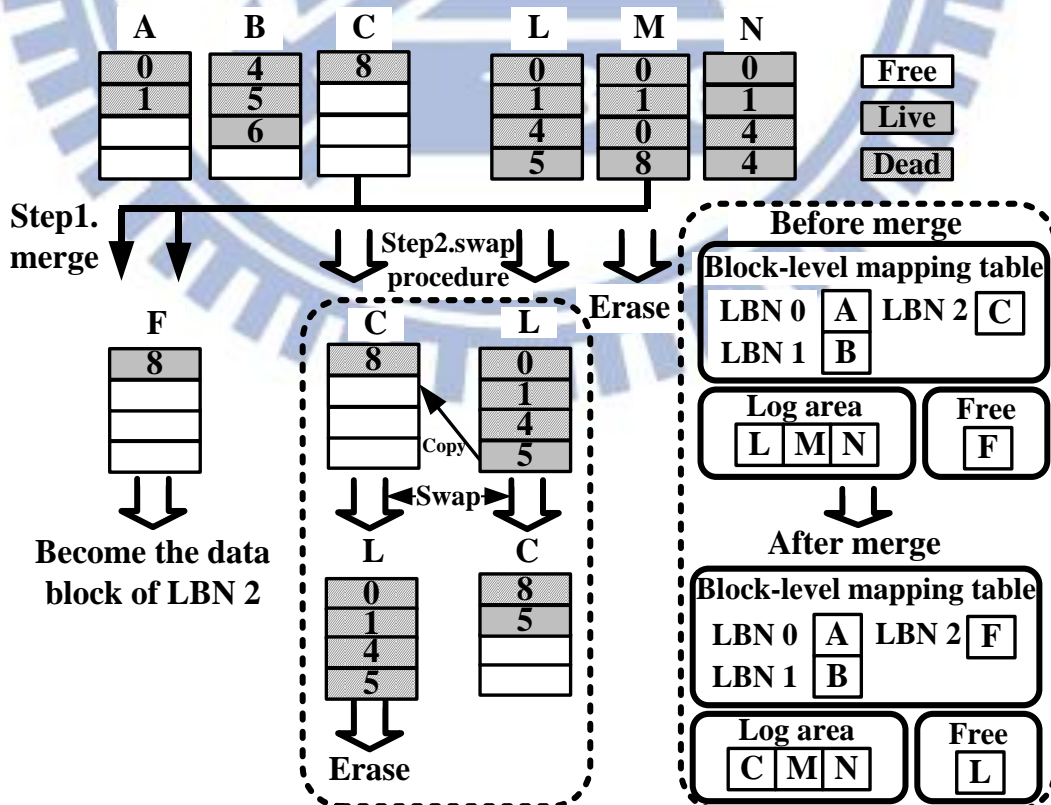
### 3.4 Free Page Reuse

As mentioned above, low space utilization can lead to high cleaning cost. In ROSE, we propose the Free Page Reuse (FPR) technique to increase the space utilization. FPR reuses free pages of obsolete blocks, which are to-be-erased blocks whose live pages have already been copied out. Therefore, an obsolete block contains only dead or free pages and FPR tries to reuse these free pages to buffer more page writes.

In ROSE, log blocks become obsolete only after they are full. However, obsolete data blocks could still contain free pages since they are managed by BAT [83]. Thus, FPR considers reusing free pages of obsolete data blocks. Instead of erasing an obsolete block  $O$ , FPR tries to select a full log block, say  $L$ , and *swaps* the roles of  $O$  and  $L$ . The procedure of the swap operation is as follows. First, the live pages of  $L$  are copied to  $O$ . Second,  $O$  is migrated to the log area

(i.e., become a new log block) and  $L$  is erased. Note,  $L$  is not migrated to the data area since it is not swapped with a valid data block. Instead, it is swapped with an obsolete block that originally needs to be erased.

The cost and benefit of the swap operation depends highly on both the number of free pages in the obsolete block, say  $f$ , and the number of live pages in the full log block selected for swap, say  $l$ . Specifically,  $l$  page copy operations are required and  $(f - l)$  free pages can be obtained to buffer further page overwrites after the swap. For effectiveness of the swap, FPR selects the full log block with the minimum number of live pages to swap. Note that a swap is performed only when the value of  $(f - l)$  is larger than zero. Otherwise, no swap is performed and ROSE just erases the obsolete block.



### Figure 3.3: An example of the swap operation

Figure 3.3 illustrates an example of the swap procedure. Assume that the flash memory consists of three data blocks ( $A$ ,  $B$ ,  $C$ ) and three log blocks ( $L$ ,  $M$ ,  $N$ ) (and an extra free block  $F$  for buffering the result of a full merge), with each block contains four pages. We also assume that data blocks  $A$ ,  $B$ , and  $C$  correspond to logical blocks 0, 1, and 2, respectively. The top of Figure 3.3 illustrates the state of the blocks after the page write sequence (0, 1, 0, 1, 4, 5, 8, 4, 5, 0, 1, 0, 8, 0, 1, 4, 4, 6). When a further write to logical page 6 arrives, the log area is full and thus a log block, say  $M$ , is reclaimed by merging the block with its corresponding data block (i.e., data block  $C$ ). The merge involves not only live page copying but also erasure of the two blocks. Specifically, without swapping,  $M$  and  $C$  are erased after the merge. FPR tries to avoid erasing  $C$  since it still has plenty of free pages. To avoid erasing  $C$ , FPR swaps log block  $L$  with  $C$  since the former is the full log block with the minimum number of live pages. As a result, page 5 is copied to  $C$ , which replaces the role of log block  $L$ , and  $L$  is erased. Therefore, with swapping,  $M$  and  $L$  are erased. Two more free pages are obtained due to the swap operation, and the cost is the copying of one page.

The above example illustrates the reclamation of a log block with page density of 1. In general, reclaiming a log block with page density of  $n$  may trigger  $m$  swap operations, where  $0 \leq m \leq n$ , and the benefit  $B_{swap}$  and extra cost  $C_{swap}$  of these swap operations can be expressed as

$$B_{swap} = \sum^m (f_i - l_i), \quad C_{swap} = \sum^m l_i, \quad (7)$$

where  $f_i$  denotes the number of free pages in obsolete (data) block  $i$  that is involved in swap, and  $l_i$  denotes the number of live pages in the full log block swapped with block  $i$ . From (7), a swap is beneficial if there are a large number of free pages in the obsolete blocks and a small number of live pages in the log blocks selected for swap. Generally, small random write dominated workloads can lead to a large number of free pages in the obsolete data blocks. Moreover, in a flash storage system with moderate number of log blocks, FPR can usually select a full log block with a very small number of live pages to swap.

Note that, the effectiveness of FPR might drop with the growth of the number of logical pages utilized by the file system. Traditionally, there is no way to allow a file system to notify the storage that a specific logical page is no longer utilized. Therefore, with the aging of the flash storage, the number of utilized logical pages grows and the expected value of the number of free pages in each obsolete data block might decrease. With the support of TRIM [76] and similar commands [20], the logical pages no longer been utilized by the file system can be released. Therefore, the drop in the effectiveness of FPR due to the aging of the flash storage can be avoided.

## 3.5 Metadata Management in ROSE

In this section, we describe the overhead of metadata management in ROSE. An FTL maintains metadata such as LPN-to-PPN mapping, page state, etc. Typically, the metadata are stored in RAM to allow fast updating. However, some information should be stored in the flash memory to allow the reconstruction of the metadata during power-on initialization.

To allow the reconstruction of the page state (i.e., live/dead/free) and the LPN-to-PPN mapping, most FTLs store the LPN and a sequence number, which is a monotonically increasing number associated with each write, in the spare area of each written page. By scanning all the pages in the flash memory during the power-on initialization, the states of all the pages can be determined. A page is *free* if the spare area is empty (i.e., containing all 1s), *live* if it has the newest sequence number among all the pages with the same LPN, and *dead* otherwise. The mapping information can be reconstructed from the LPNs of the live pages. In addition, erase counts can be stored in extra flash pages to handle the wear-leveling issue.

Besides the above information, HAT-based FTLs such as FAST, LAST and ROSE generally also store a 1-bit flag in the spare area to tell if a given block is a data or a log block (i.e., 0 for data block and 1 for log block). Traditionally, examining the flag in one of the written pages in a given block is sufficient to determine if the block is a data or log block. However, since FPR may change



the role of a data block to a log block, the flags in all the written pages in a block may need to be examined in ROSE. If a given block has any written page(s) with the flag set as 1, the block is a log block. Compared to traditional HAT-based FTLs, ROSE maintains two more types of metadata, the age and score of each log block. According to Section 3.2, the age can be stored in the spare area upon the first page write to a log block, allowing it to be reconstructed easily upon power-on initialization. With the presence of the page state and block age information, the scores can be reconstructed.

From the above description, all the metadata of ROSE can be constructed by scanning the spare areas and extra flash pages of the storage during initialization, which is  $O(N)$  in time complexity where  $N$  is the number of pages in the storage, the same as those in FAST and LAST. Note that, it is also possible to maintain all the metadata in extra flash pages so as to reducing the frequency of page scanning. In that approach, the time to construct the metadata would be proportional to the size of the metadata. Compared to FAST, the additional time for constructing the metadata in ROSE is the loading of the (age, score) pair for each log block, which is  $O(N_L)$  in time complexity where  $N_L$  is the number of log blocks in the storage.

## Chapter 4

### The HybridLog FTL

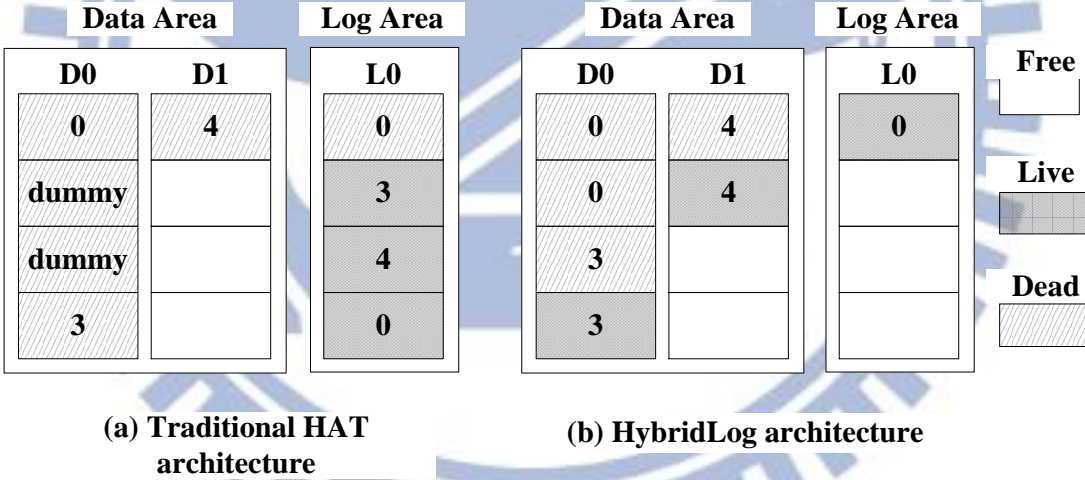
With the development of flash memory, new restrictions are imposed on modern flash memory chips, and an FTL should follow these new restrictions so it can be applied on these modern chips. Specifically, consecutive programming and higher space requirement of ECC are two typically restrictions of modern flash memory chips.

Many HAT-based FTLs cannot support consecutive programming efficiently since the constraint of the block-level mapping (i.e., each logical page can only be written to its corresponding offset in a physical block) is still valid for the data blocks. In these HAT-based FTLs, therefore, dummy page writes may still be required during workload execution. Although FTLs such as Superblock [35, 37] avoid this problem, they either have limited support to large-block MLC flash memory, due to the storing of a large amount of information in the spare area and thus prohibiting the use of strong ECC, or suffer from inferior performance.

We propose a HAT-based FTL called HybridLog to support modern NAND flash memory and achieve performance superior to existing HAT-based FTLs. In the following, the design of HybridLog is described.

# 4.1 Architecture of HybridLog

The same as traditional HAT-based FTLs, HybridLog divides the flash memory into two areas, a large data area containing data blocks managed by block-level mapping and a small log area containing log blocks managed by page-level mapping. Each logical block has an associated data block to accommodate writes to that logical block, and thus the user perceived storage size is the data area size. However, HybridLog adopts a novel architecture to allow consecutive programming and to reduce cleaning cost.



**Figure 4.1: Traditional HAT architecture vs. HybridLog architecture**

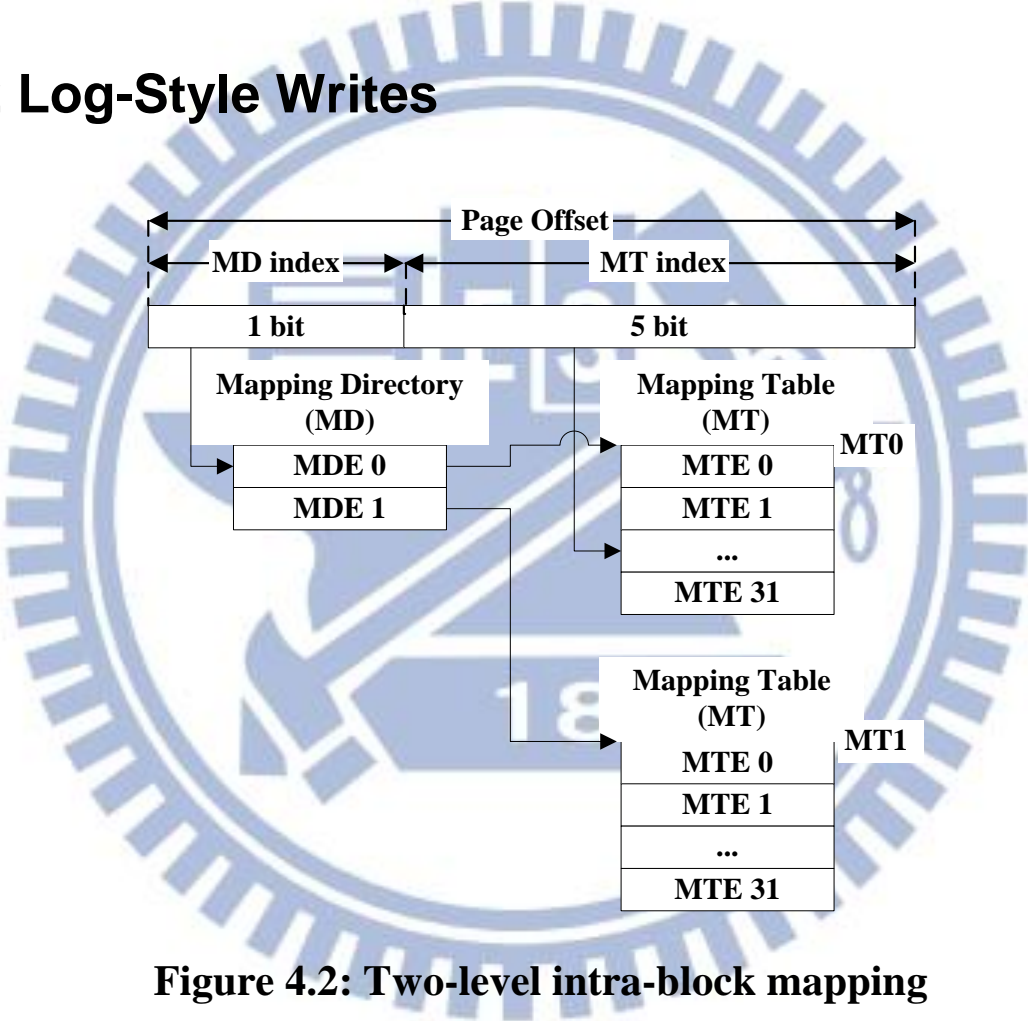
Different from traditional HAT architecture, the HybridLog architecture enables log-style writes to all the blocks in the flash memory, especially the data blocks, allowing consecutive programming. Since data blocks are written in a log order, log blocks are used only after a data block is full. The log-style writes

keep 100% utilization of the data blocks even under the random-write workloads.

Figure 4.1 illustrates an example showing the difference between HybridLog and the traditional HAT architecture. Assume that the flash memory consists two data blocks and one log block, with each block containing four pages. Figure 4.1(a) and Figure 4.1(b) illustrate the result of the page write sequence (0, 0, 3, 4, 3, 4, 0) under the traditional HAT and HybridLog architectures, respectively. In Figure 4.1(a), although the first write to (logical) page 0 can be served by  $D0$ , the second write to the page 0 has to be served by the log block due to page collision. Moreover, the first write to page 3 has to be served by the last physical page of  $D0$  due to the use of block mapping in the data area, and two dummy pages have to be written to the second and third pages of  $D0$  before the write of page 3 to follow consecutive programming. Such dummy page writes increase not only the write response time but also the WAR. The second writes to logical pages 3 and 4 also incur page collisions in  $D0$  and  $D1$ , respectively, and therefore these two page writes have to be satisfied by the log block. After the third write to page 0, the log block is full and cannot accommodate further page writes. In Figure 4.1(b), except the last page write to page 0, all page writes are proceeded in log-style in the corresponding data blocks  $D0$  and  $D1$ . The last write to page 0 is satisfied by the log block because the corresponding data block  $D0$  is full. After the last logical page is written, the log block still has three free pages to accommodate further page writes.

As a result, HybridLog not only eliminates unnecessary dummy page writes but also reduces the write traffic to the small-sized log area. Therefore, cleaning cost and WAR can be reduced. In the following, the technique to enable log-style writes in all blocks is described.

### 4.2 Log-Style Writes



**Figure 4.2: Two-level intra-block mapping**

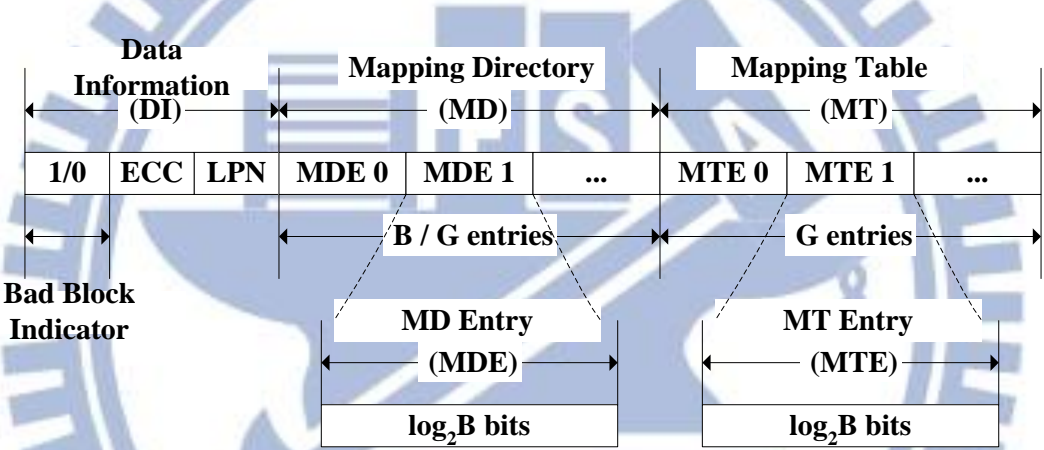
HybridLog uses the BAT approach to manage the data blocks. If the target page that needs to be written is not consecutive to the last written page in the data block, traditional BAT and HAT approaches fill dummy content to satisfy

the consecutive programming restriction of modern NAND flash memory. This causes overhead in both time and flash memory space, and the space overhead could be large for small random writes.

Although log-style writes in a data block can be achieved by using PAT for data blocks. This causes large RAM space for the mapping information. To enable log-style writes in a data block while preventing increasing the RAM size for the total mapping information, HybridLog stores the intra-block mapping information (i.e., the physical page offset of each logical page in a data block) in the spare area of each written page. The information is organized as a two-level mapping table. As shown in Figure 4.2, which assumes 64-page blocks, the first-level mapping is called the *Mapping Directory (MD)*. Each entry in the *MD* refers to a *Mapping Table (MT)*, and each entry in the *MT* records the physical page offset for the corresponding logical page. It can be regarded as each logical block being divided into a number of groups, with each group containing a fixed number of contiguous logical pages. Each *MT* records the mapping information of a group and the *MD* keeps track of the location of all the *MTs* in the logical block. In Figure 4.2, the block is divided into 2 groups with each group containing 32 contiguous logical pages. For each page write to a data block, the up-to-date *MD* and *MT*, derived from the information in the spare area of the last written page in that data block and the information of the to-be-written page, are stored in the spare area of the to-be-written page.

Figure 4.3 illustrates the spare area format of each written page, which is

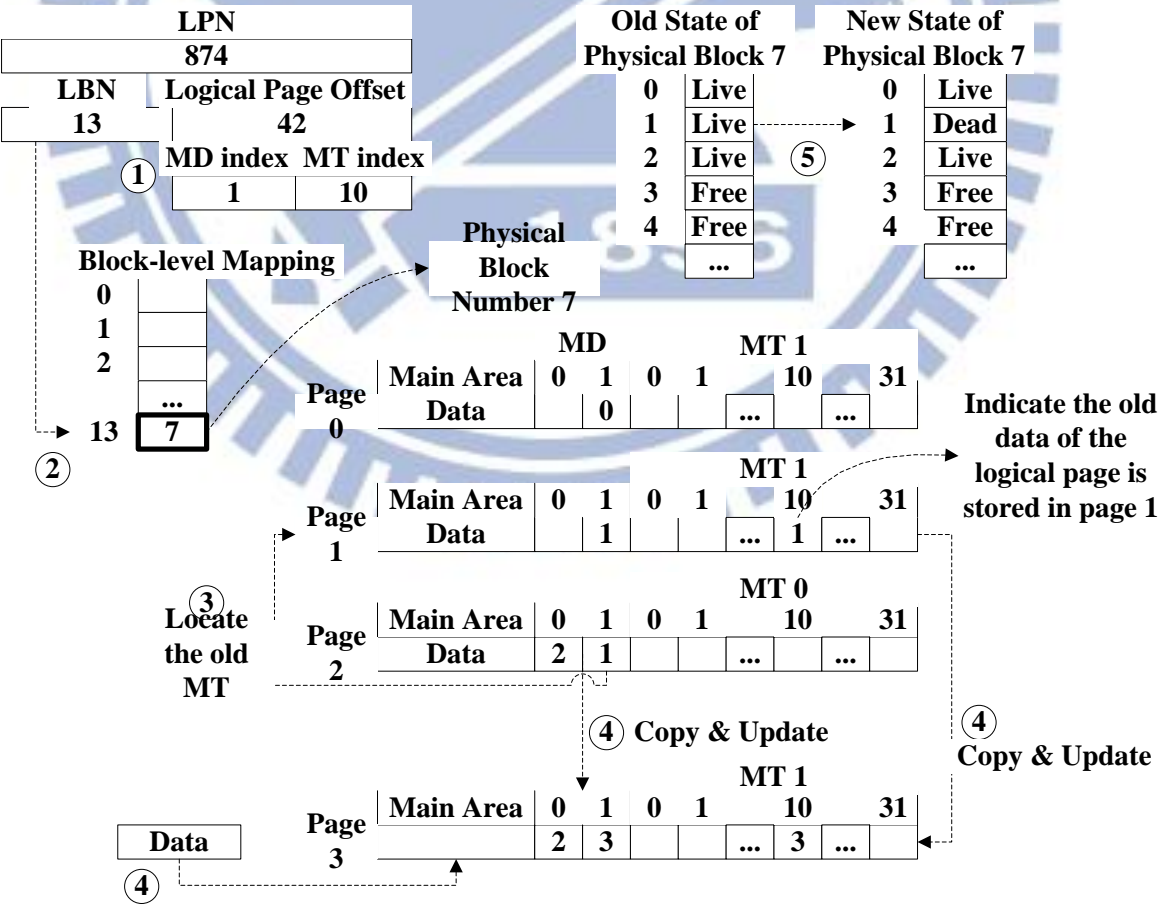
divided into 3 sections: *data information (DI)*, *MD* and *MT*. The *DI* contains bad block indicator, LPN and ECC, while the other sections are used for recording the mapping information. Assume  $B$  and  $G$  denote the number of pages per block and the number of pages per group, respectively, a  $B/G$ -entry *MD* and a  $G$ -entry *MT* are included in each spare area. Each entry has a size of  $\log_2 B$  bits since it is used to locate a page in the physical block. The size requirement of the spare area space will be analyzed later.



**Figure 4.3: The spare area format of each written page**

Figure 4.4 shows the steps of writing a page with LPN 874 to the corresponding data block, assuming 64-page blocks and 32-page groups. First, the LPN is divided into logical block number (LBN) 13 and page offset 42, and the latter is in turn divided into *MD* index 1 and *MT* index 10, meaning the page offset is stored in entry 10 of *MT1*. Second, the LBN is used to index the block-level mapping table to obtain the physical block number 7. In that physical block, the first free page will be used to accommodate the write. From the old state of physical block 7 shown in the top right of Figure 4.4, page 3 is

the first free page of physical block 7, and thus it is used to accommodate the write. Third, the location of the *MTI* is obtained by indexing the *MD* of page 2, the last written page in the physical block. From the figure, the entry 1 of *MD* refers to page 1, indicating that *MTI* is stored in the spare area of page 1. Moreover, the entry 10 of *MTI* also refers to page 1, meaning the old data of logical page 874 is stored in page 1. Fourth, entry 1 of *MD* and entry 10 of *MTI* are both updated to refer to page 3, and the data together with the latest mapping information are written to that page. Finally, the page 1, which stores the old data of the logical page, is marked as dead, and page 3 is marked as live.



**Figure 4.4: Steps of writing a page**



From this example, the content of the new  $MD$  and  $MT$  (i.e., the  $MD$  and  $MT$  in page 3) are obtained from the old  $MD$  and the old  $MT$ , respectively. Specifically, assuming logical page 1 is to be written to physical page  $p$  of the target data block, and  $MD(i)$  and  $MD'(i)$  denote the  $i$ th entry of the old and the new  $MD$ s, respectively, the  $MD$  and  $MD'$  can be obtained by (8) and (9), respectively.

$$MD = \begin{cases} NULL, & \text{if } p = 0, \\ MD \text{ section in the spare area of page } p - 1, & \text{otherwise.} \end{cases} \quad (8)$$

$$MD'(i) = \begin{cases} p, & \text{if } i = \lfloor 1/G \rfloor, \\ MD'(i), & \text{if } i \neq \lfloor \frac{1}{G} \rfloor \text{ and } MD \neq NULL, \\ NULL, & \text{otherwise.} \end{cases} \quad (9)$$

From (8), the old  $MD$  is stored in the last written page (i.e., page  $p-1$ ) of the target block. Generally, all the entries of the new  $MD$  except from the one that refers to the  $MT$  containing the logical page 1 are copied from the corresponding entries of the old  $MD$ . However, in the case of a first-page write to a physical block (i.e.,  $p = 0$ ), the old  $MD$  does not exist. In that case, all the entries of the new  $MD$  are set as  $NULL$ , except from the entry that refers to the  $MT$  containing the logical page 1. Similarly, assuming  $MT(i)$  and  $MT'(i)$  denote the  $i$ th entry of the old and the new  $MT$ s, respectively, the  $MT$  and  $MT'$  can be obtained by (10) and (11), respectively.

$$MT = \begin{cases} NULL, & \text{if } MD \left( \left\lfloor \frac{1}{G} \right\rfloor \right) = NULL, \\ MT \text{ section of the spare area referred by } MD \left( \left\lfloor \frac{1}{G} \right\rfloor \right), & \text{otherwise.} \end{cases} \quad (10)$$

$$MT'(i) = \begin{cases} p, & \text{if } i = 1, \\ MT(i), & \text{if } i \neq 1 \text{ and } MT \neq NULL, \\ NULL, & \text{otherwise.} \end{cases} \quad (11)$$

From the above description, it can be seen that page read/write requires additional spare area reads to lookup the intra-block mapping. To reduce the spare area reads, recently used intra-block mapping is cached in RAM. Each cache entry stores the mapping of a data block (i.e., the up-to-date  $MD$  and the associated  $MT$ s). Due to the temporal and spatial locality of page access, few cache entries are adequate for achieving a high cache hit ratio.

### 4.3 Spare Area Requirement of HybridLog

In the following, the spare area space requirement is analyzed. According to Figure 4.3, the space required by the intra-block mapping  $M_{spare\_area}$  can be expressed in (12). To allow the mapping to be fitted into the spare area, (13) should hold, given  $S$  and  $D$  denoting the sizes of the spare area and  $DI$ , respectively.

$$M_{spare\_area} = G * \log_2 B + \frac{B}{G} * \log_2 B \quad (12)$$

$$M_{spare\_area} \leq S - D \Rightarrow \left(G + \frac{B}{G}\right) * \log_2 B \leq S - D \quad (13)$$

From (13), a set of possible values of  $G$  (i.e., pages per group) can be obtained for given values of  $B$ ,  $S$  and  $D$ . Table 4.1 shows the common modern flash memory configurations and the corresponding possible values of  $G$ . Typically, the values of  $B$  (i.e., pages per block) are 64 and 128 for SLC and MLC, respectively. The value of  $S$  (i.e., spare area sizes) is typically 64 bytes for both SLC and MLC. The value of  $D$  is equal to the size of ECC plus the sizes of LPN (typically 4 bytes) and bad block indicator (typically 1 byte). In general, the number of bits required by the ECC depends on the flash type and the error correction algorithm. Most SLC and MLC modules require correcting 1-bit and 4-bit errors for each 512 bytes of data, respectively, and ECC sizes in this table are calculated based on satisfying that requirement by using the *BCH* algorithm [17], the most widely-used error correction algorithm in flash storages.

**TABLE 4.1: COMMON FLASH MEMORY CONFIGURATION AND THE CORRESPONDING GROUP SIZES**

FLASH TYPE	SLC	MLC
PAGE PER BLOCK	64	128
MAIN/SPARE AREA SIZE	2048/64 BYTES	2048/64 BYTES
ECC SIZE	7 BYTES	26 BYTES
PAGE PER GROUP	1, 2, 4, 8, 16, 32, 64	8, 16

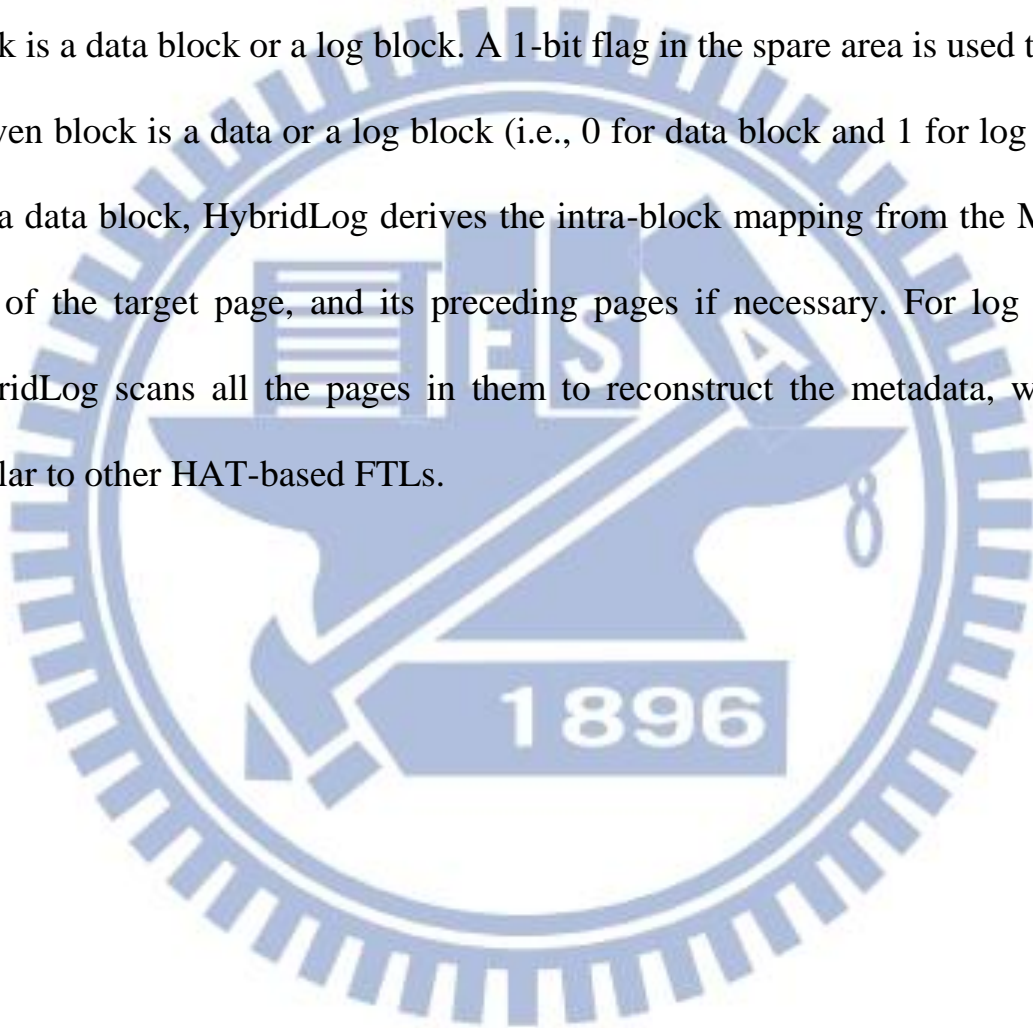
Although the Superblock FTL also stores the mapping information in the

spare area, it tightly limits the maximum number of physical blocks allocated to a logical block (i.e., 8 blocks). This could lead to high cleaning cost due to the use of small buffers to accommodate page updates of frequently-updated logical blocks. Moreover, Superblock consumes a larger amount of spare area space due to the recording of multiple physical block numbers, prohibiting its use on some types of MLC flash. For example, in Table 4.1, for the MLC NAND flash with main/spare area size as 2048/64 bytes, Superblock leaves only 8 bytes in the spare area for the ECC and thus it cannot be used on that type of flash memory. Decreasing the maximum number of physical blocks allocated to a logical block allows the support of more types of MLC flash with the cost of degraded performance. Although Superblock FTL can adopt an alternative approach, which stores the mapping information in the user area of dedicated pages called map pages, it incurs extra programming overhead for these map pages. In contrast, HybridLog uses the PAT approach in the log area and thus the entire log area can be used to buffer page updates of any logical blocks. In addition, it supports more MLC flash memories since only page offset information is stored in the spare areas.

## 4.4 Reconstruction of Metadata

In this section, we describe the reconstruction of the metadata after power resets in the HybridLog FTL. After a power reset, HybridLog scans each block

(from the last page to the first page) to find the last written page as the target page. If the target page is corrupted, HybridLog selects the previous page as the new target page. A corrupted page may exist due to sudden power failure. If all the written pages are corrupted (i.e., no target page is found), the block is abandoned. According to the target page, HybridLog determines whether this block is a data block or a log block. A 1-bit flag in the spare area is used to tell if a given block is a data or a log block (i.e., 0 for data block and 1 for log block). For a data block, HybridLog derives the intra-block mapping from the MD and MT of the target page, and its preceding pages if necessary. For log blocks, HybridLog scans all the pages in them to reconstruct the metadata, which is similar to other HAT-based FTLs.



# Chapter 5

## Performance Evaluation

A trace-driven simulator [2, 24, 44, 46, 48, 56] was developed for the performance evaluation. In addition to ROSE and HybridLog, three well-known HAT-based FTLs, FAST, LAST, and Superblock, were also implemented in the simulator for performance comparison. Section 5.1 presents the experimental setup and the workloads used in the simulation. Sections 5.2 and 5.3 show the performance of ROSE and HybridLog, respectively.

### 5.1 Experimental Setup and Traces

Table 5.1 shows the default values of the parameters in the simulator. An 80-Gbyte flash storage (i.e., 655360 blocks) is simulated. In all the experiments except from the one corresponding to Table 5.5, 2.5% of the storage (i.e., 16384 blocks) is reserved for the log area. In Table 5.5, the cleaning cost of the FTLs under different log area sizes are reported. In the LAST FTL, one-eighth log blocks are SW log blocks, which serve write requests with sizes equal to or larger than 8 Kbytes. The 8-Kbyte threshold is used since it results in the best performance in most of the traces. All the time-related values in Table 5.1 are obtained from the specification of the Samsung K9K4G08U0M NAND flash

chip [73]. Note that, the values of  $W_{age}$  and  $\alpha$  shown in Table 5.1 are used in all the experiments except for those corresponding to Figure 5.5 and 5.6. In the experiments corresponding to Figure 5.5 and 5.6, the values of  $W_{age}$  and  $\alpha$  are varied, respectively, to evaluate their effect on the cleaning cost.

**TABLE 5.1: DEFAULT VALUES OF THE PARAMETERS**

Parameters	Default values
Number of Blocks	655360
Number of Pages Per Block	64
Page Size	2 Kbytes
Block Erase Time	2000 us
Page Read/Write Time	88/263 us
$W_{age}$	1
$\alpha$	0.5
$T_{hc}$	10

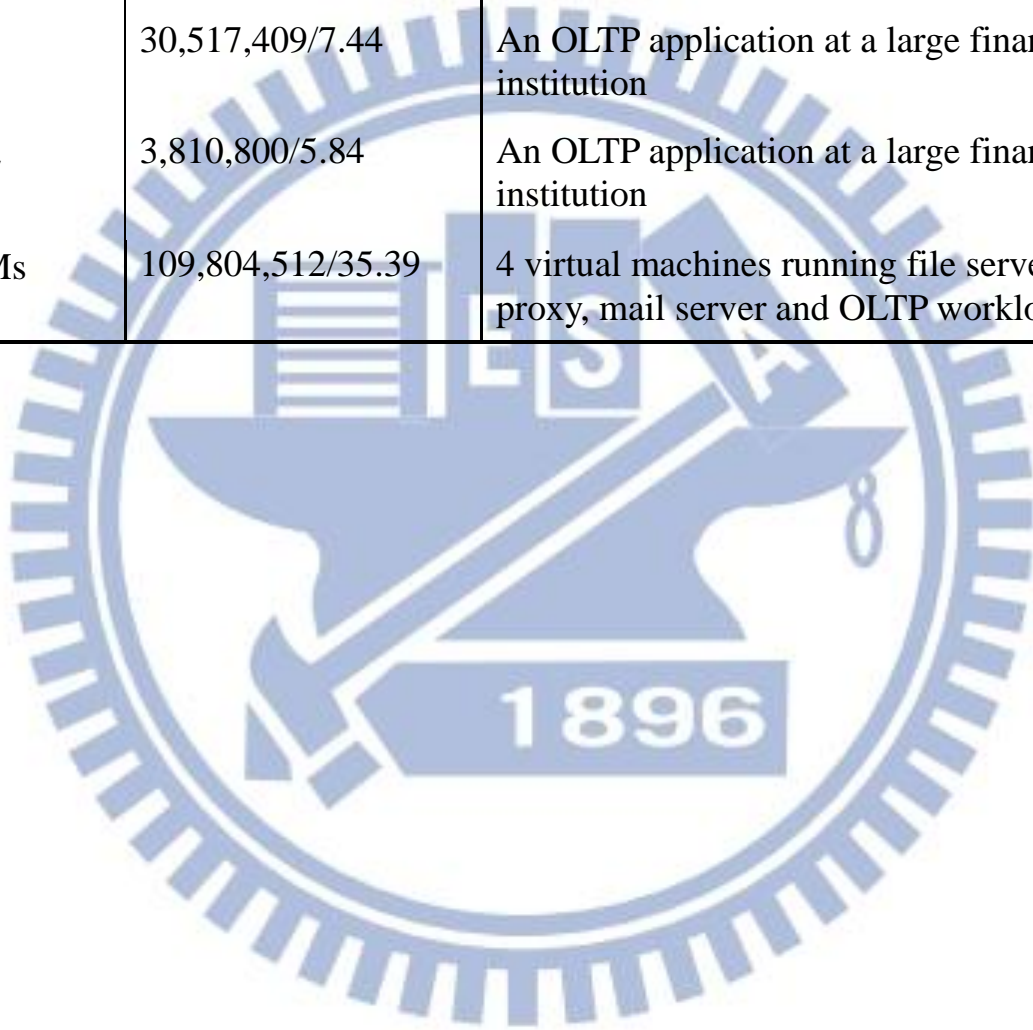
As shown in Table 5.2, six device-level traces are used in the experiments. The LinuxPC trace is a ten-day workload on a Linux laptop computer, which includes daily user activities such as web browsing, file browsing and editing, multimedia file playing, and program compilation. The Postmark trace is generated from the execution of the PostMark file system benchmark [39], which emulates the workload of an Internet email server. PostMark first creates 80,000 small files, and performs 1,000,000 transactions such as create, delete,

read, and append on the files. This causes a large number of small random writes to the storage. The LargeFile trace is the workload of creating and deleting MP3 files, whose average size is about 4 Mbytes, and is dominated by large sequential writes. The ratio of file creation to deletion is set as 10 and the workload terminates until the total number of existing files exceeds 10,000. The Fin1 and Fin2 traces obtained from [5] are workloads of OLTP applications running at two large financial institutions. The 4VMs trace is a mixed workload generated from the execution of four virtual machines on top of the VirtualBox-3.1.2 hypervisor. Each virtual machine, equipped with 768-Mbyte memory and 20-Gbyte virtual disk, runs one of the following workloads on the Linux kernel 2.6.31: file server, web proxy, mail server and OLTP. The workloads are obtained from the FileBench file system benchmark [22]. The number of 512-byte sectors written and the average size of the write requests in each trace are also shown in Table 5.2.



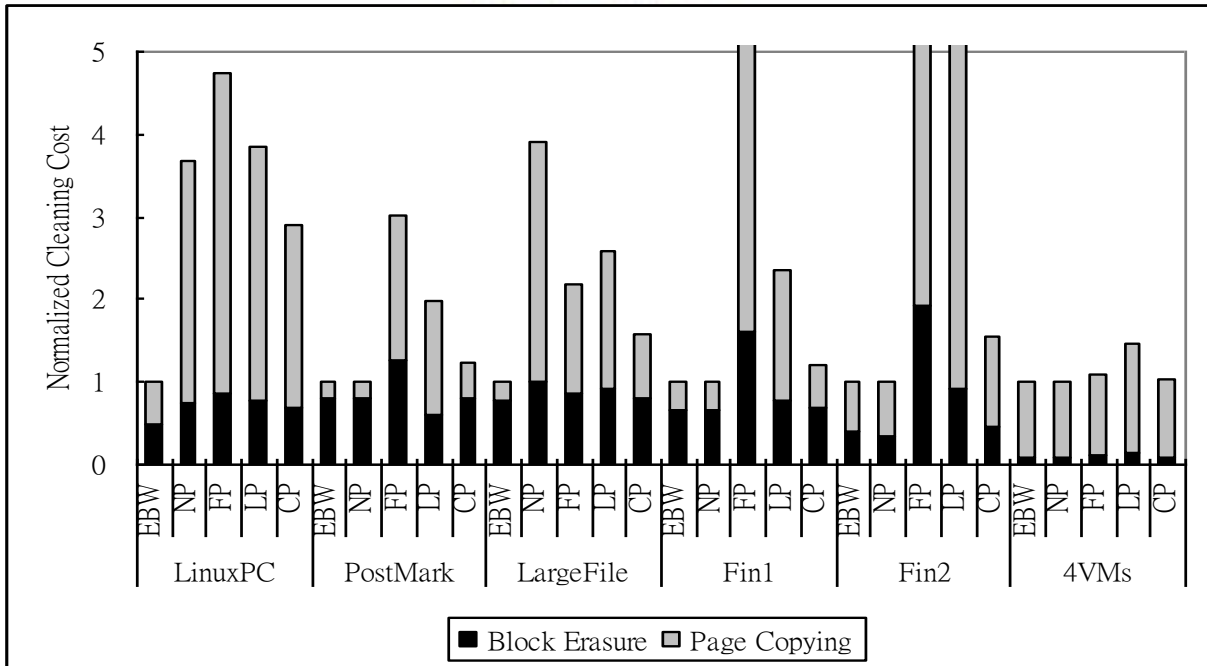
**TABLE 5.2: TRACES**

<i>Traces</i>	<i>Sectors written/Ave. write sizes (sectors)</i>	<b>Description</b>
LinuxPC	107,111,668/71.76	10-day user activities
Postmark	8,816,216/6.68	Running the PostMark benchmark
LargeFile	60,149,736/748.61	Creating and deleting MP3 files
Fin1	30,517,409/7.44	An OLTP application at a large financial institution
Fin2	3,810,800/5.84	An OLTP application at a large financial institution
4VMs	109,804,512/35.39	4 virtual machines running file server, web proxy, mail server and OLTP workloads



## 5.2 Performance Evaluation of ROSE

### 5.2.1 Effect of Entire-Block Writing



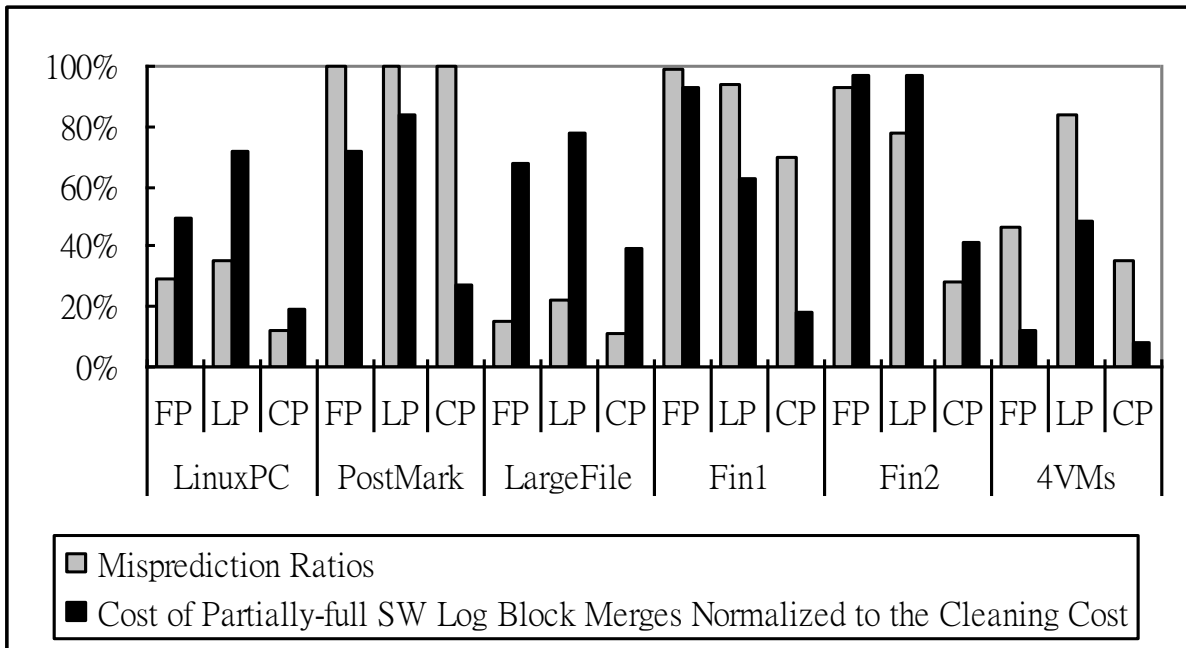
**Figure 5.1: Cleaning cost of the methods for handling sequential overwrites (normalized to the cleaning cost of EBW)**

The effectiveness of EBW is demonstrated by comparing its performance with different sequentiality prediction methods. Figure 5.1 shows the cleaning cost, which includes block erase time and page copying time, of different methods for handling sequential overwrites. In the figure, EBW corresponds to ROSE with EBW enabled, and both MARO and FPR disabled. FP, LP and CP correspond to three different prediction methods. FP denotes the FAST FTL,

which predicts sequentiality based on LPN. It serves a page collision by the SW log block if the following condition holds: the write is to the first page of a logical block or corresponds to the first free page of the SW log block. LP denotes a modified version of FAST that utilizes the prediction method of the LAST FTL (i.e., serves a write request via the SW log block if the following condition holds: the request size is equal to or larger than 8 Kbytes). CP denotes another modified version of FAST that utilizes a prediction method based on the combination of FP and LP. Specifically, it serves a write request via the SW log block if both of the conditions of FP and LP hold. Finally, NP denotes a modified version of FAST that does not utilize any techniques for detecting or predicting sequentiality (and thus no SW log blocks are used). Since the values of the traces have different orders of magnitude, they are normalized to the cleaning cost of EBW for easy illustration.

From Figure 5.1, although the prediction methods could result in lower cleaning cost in sequential write dominated workloads, mispredictions could occur quite frequently in random write dominated workloads such as *Postmark*, *Fin1* and *Fin2*, leading to increased cleaning cost in the latter workloads when compared to the FTL without using any techniques to predict or detect sequentiality. The misprediction ratio, which represents the ratio of the number of merges of the SW log block when the block is partially-full to the total number of merges of the SW log block, is shown Figure 5.2. A high misprediction ratio indicates that the SW log block is usually merged when it is

only partially-full. The total cost of such partially-full SW log block merges, normalized to the overall cleaning cost, is also presented in Figure 5.2.



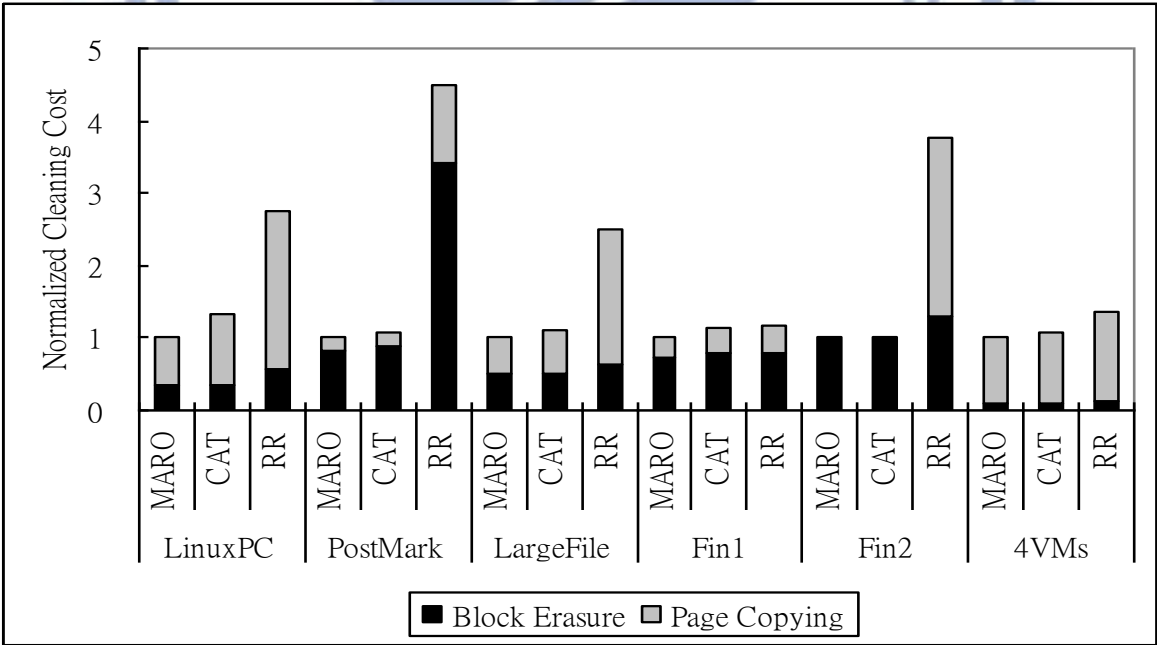
**Figure 5.2: Misprediction ratios and cost of partially-full SW log block merges in the sequentiality prediction methods**

**TABLE 5.3: PORTIONS OF ENTIRE-BLOCK WRITES**

Traces	Number of pages written by entire-block writes (% of the number of pages written)
LinuxPC	50%
Postmark	0%
LargeFile	72%
Fin1	1%
Fin2	6%
4VMs	9%

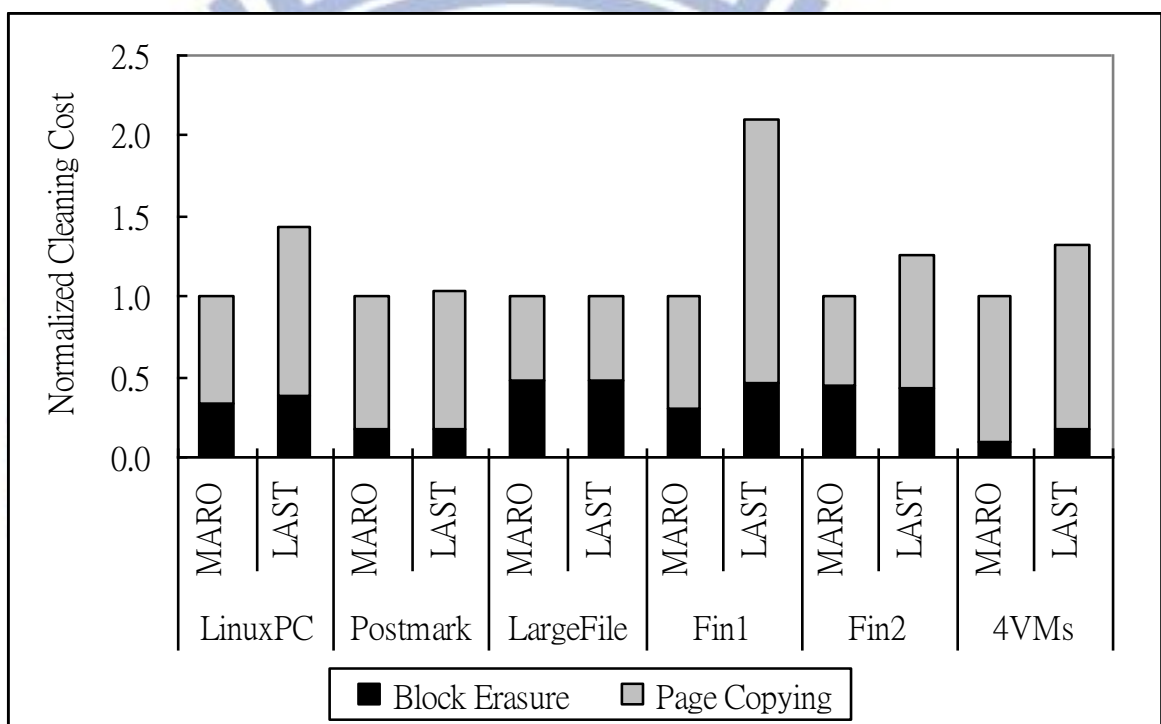
Figure 5.1 also reveals that, when compared to NP, EBW results in lower cleaning cost in sequential write dominated workloads, but without the negative effect (i.e., increased cleaning cost) in random write dominated workloads. Table 5.3 shows the percentage of the number of pages written by entire-block writes under each trace when EBW is used. From the table, entire-block writes occur more frequently in sequential write dominated workloads such as *LinuxPC* and *LargeFile*, allowing EBW to achieve lower cleaning cost under these traces.

### 5.2.2 Effect of MARO Cleaning Policy



**Figure 5.3: Cleaning cost of different cleaning policies (normalized to the cleaning cost of MARO)**

To evaluate the performance of MARO, we compare it with two policies, Round-Robin (RR), which is used in FAST, and Cost-Age-Time (CAT), an efficient block reclamation policy. Figure 5.3 shows the cleaning cost of RR and CAT normalized to that of MARO. From the figure, MARO outperforms RR by up to 3.5 times and CAT by up to 31%, respectively, under the traces.



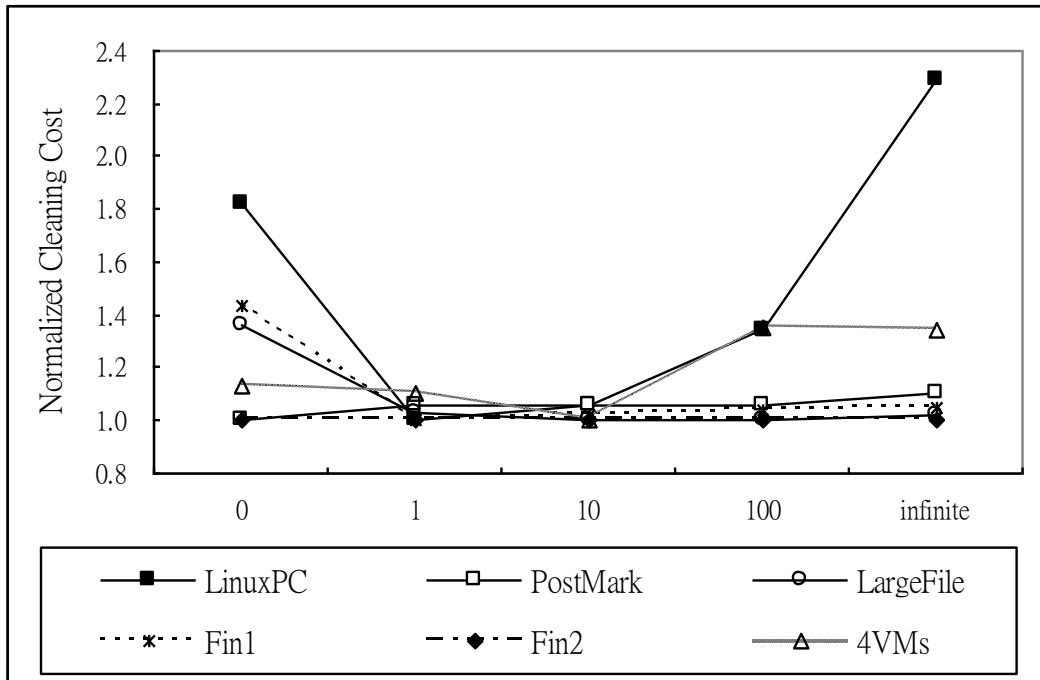
**Figure 5.4: Cleaning cost of MARO and LAST (normalized to the cleaning cost of MARO)**

We also implement MARO in the LAST FTL to compare the performance of MARO and the cleaning policy used in the LAST FTL (or simply the LAST policy). Figure 5.4 shows the cleaning cost of the two policies normalized to that of MARO. From the figure, MARO outperforms the LAST policy by up to 1.1

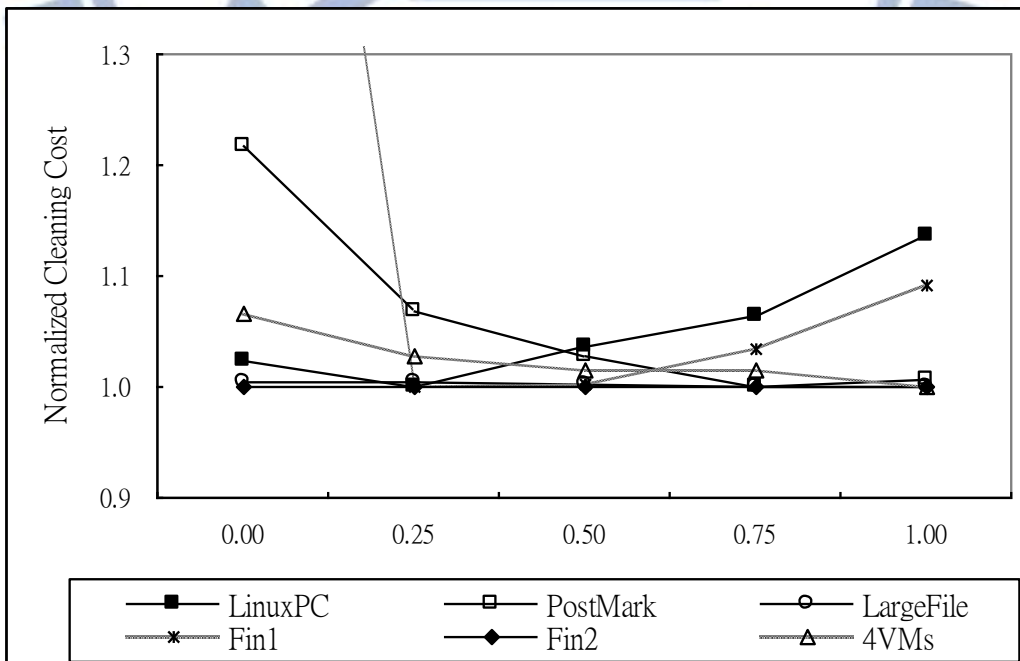
times. The results of Figure 5.3 and 5.4 reveal that the performance improvement of MARO comes mainly from the reduction of page copying cost. This is because the per-bit data copying cost is higher than the per-bit data erasing cost. From Table 5.1, the cost of copying only six pages is higher than erasing a whole block. Therefore, MARO would prevent a log block to be reclaimed if the reclamation involves copying a large number of pages. This leads to an effective reduction on the page copying overhead.

Next, we evaluate the performance of MARO under different values of  $W_{age}$  and  $\alpha$ . Figure 5.5 illustrates the cleaning cost of MARO with  $W_{age}$  ranging from 0 to infinity. Note that, setting  $W_{age}$  as infinity means that MARO selects victims only based on the block age (without considering the merge cost) when dead log blocks are not available. For each trace, five  $W_{age}$  settings are tested, and the results normalized to the minimum cleaning cost under these settings are reported. From the figure, setting  $W_{age}$  as 0 results in a relatively large cleaning cost in the *LinuxPC*, *Fin1* and *LargeFile* traces, and setting  $W_{age}$  as extremely large values also results in large cleaning cost in the *LinuxPC* and *4VMs* traces. This demonstrates that both the block age and the merge cost need to be considered. Exceptions appear in the *Postmark* and *Fin2* traces, under which  $W_{age}$  does not have significant effect on the cleaning cost. Note that, according to Figure 5.3 and 5.5, setting  $W_{age}$  as infinity still results in lower cleaning cost than RR. This is because, as mentioned in Section 3.3, MARO still reclaims dead log blocks, i.e., blocks with lowest reclamation cost, before selecting victims based on (5),

whereas RR does not consider merge cost at all.



**Figure 5.5: Cleaning cost with different  $W_{age}$**



**Figure 5.6: Cleaning cost with different  $\alpha$**

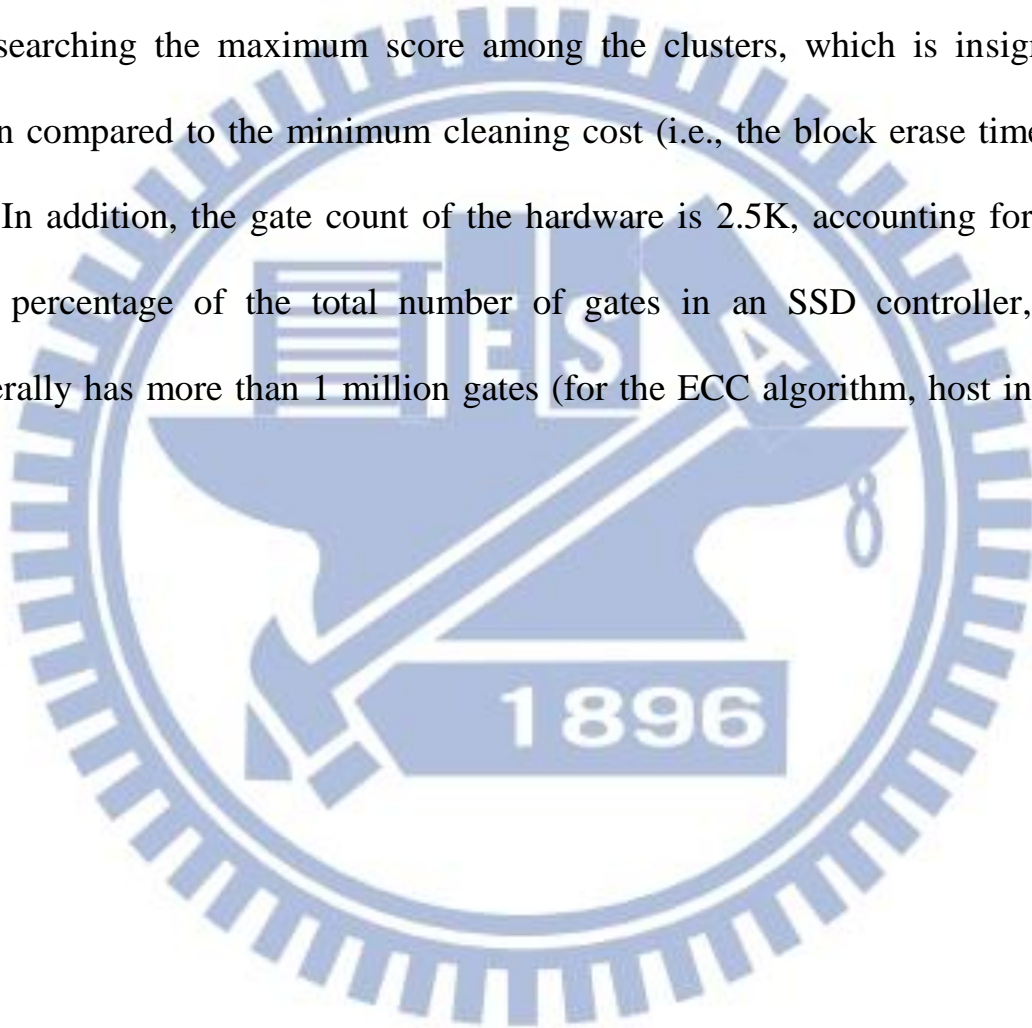


Figure 5.6 shows the cleaning cost of MARO with  $\alpha$  ranging from 0 to 1. For each trace, five  $\alpha$  settings are tested, and the results normalized to the minimum cleaning cost under these settings are reported. As mentioned before,  $\alpha$  is always smaller than 1 in MARO. The results corresponding to  $\alpha$  as 1 are reported to compare the existing merge cost evaluation approach and that used in MARO. As illustrated in the figure, setting  $\alpha$  as 1 does not always lead to the best performance, indicating that respecting the benefit of copying pages corresponding to dead pages of the data blocks helps to reduce the cleaning cost. For example, under the *LinuxPC* trace, the cleaning cost with  $\alpha$  as 1 is 15% larger than that with  $\alpha$  as 0.25. According to Figure 5.5 and 5.6, we set  $W_{age}$  as 1 and  $\alpha$  as 0.5 in the other experiments of this thesis.

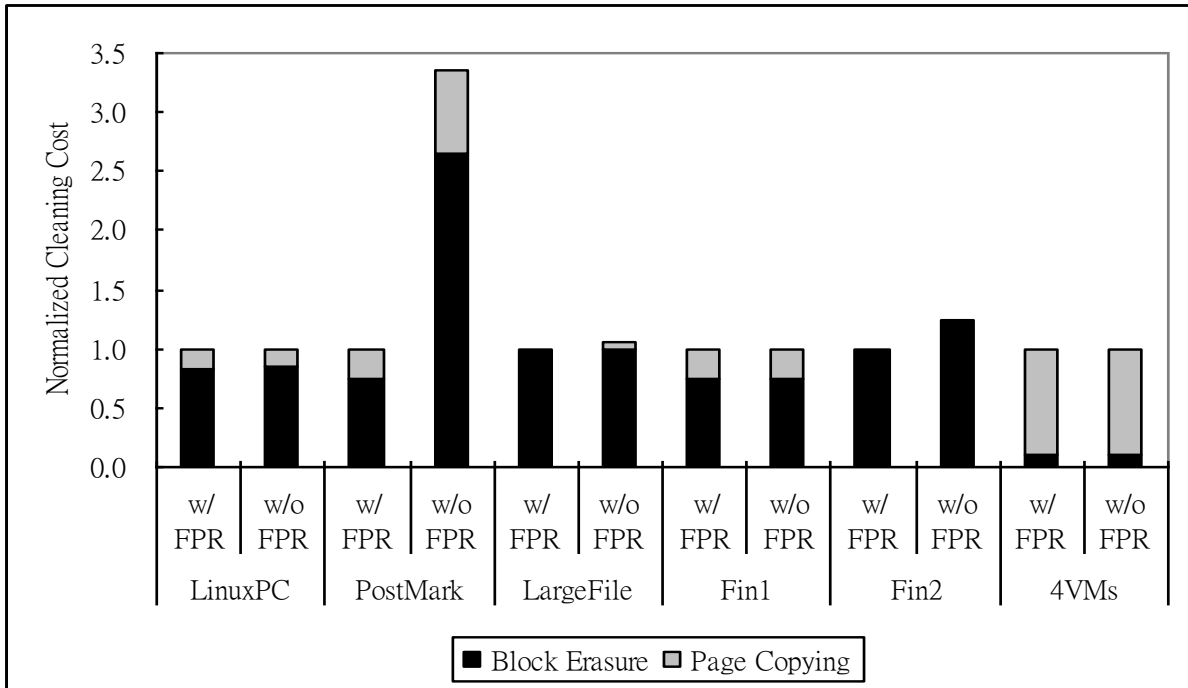
Below we present the time of score computation and maximum score searching. The execution time of the software part of MARO is obtained by ARMulator, which simulates a 200MHz ARM926 processor (16-Kbyte I-cache, 16-Kbyte D-cache and no floating point unit). The performance of the simulated processor is common for the processor units in state-of-the-art SSD controllers. The hardware part is implemented by Verilog HDL and synthesized by SYNOPSYS DesignVision with TSMC's 0.18 um cell library. The layout for the hardware design is generated with SYNOPSYS Astro (for auto placement and routing), and verified by MENTOR GRAPHIC Calibre (for DRC and LVS

checks).

For each page write, the worst case execution time of the MARO implementation is 7.6 us (including computation of scores and search of intra-cluster maximum scores), which can be totally hidden from the page write time (263 us). The size of a cluster is 8 segments. During cleaning, 1.2 us is used for searching the maximum score among the clusters, which is insignificant when compared to the minimum cleaning cost (i.e., the block erase time, 2000 us). In addition, the gate count of the hardware is 2.5K, accounting for only a tiny percentage of the total number of gates in an SSD controller, which generally has more than 1 million gates (for the ECC algorithm, host interface, etc.)

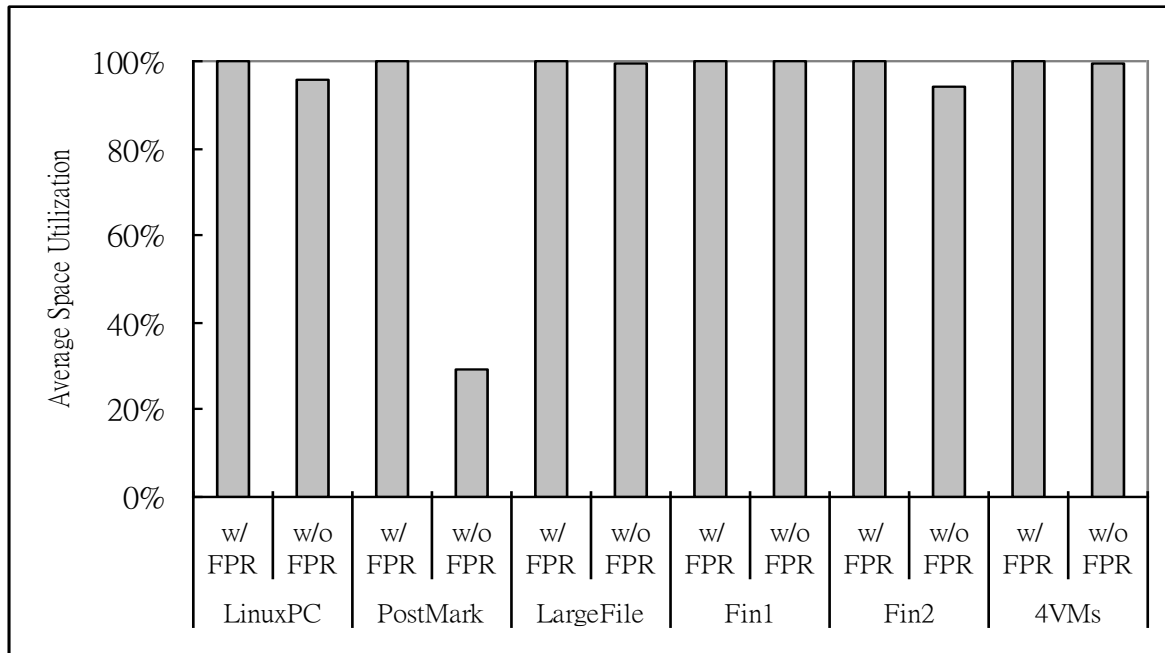


### 5.2.3 Effect of Free Page Reuse



**Figure 5.7: Cleaning cost w/ and w/o FPR (normalized to the cleaning cost w/ FPR)**

To evaluate the performance of FPR, we measure the cleaning cost with and without the presence of FPR. As shown in Figure 5.7, FPR is effective in *Postmark* and *Fin2*. Specifically, it reduces the cleaning cost by 70% and 19% in the *Postmark* and *Fin2* traces, respectively. The reason can be seen in Figure 5.8, which shows the average space utilizations with and without the presence of FPR. In Figure 5.8, the two traces have lower space utilizations when FPR is not present, meaning that some free pages, which can originally be used to accommodate more writes, are erased. FPR increases the space utilizations under these traces, which leads to reduction of the cleaning cost.



**Figure 5.8: Average space utilizations w/ and w/o FPR**

As mentioned in Section 3.4, FPR reuses the free pages of an obsolete block through a swap operation. Table 5.4 shows the average number of pages copied during each swap operation, the average number of free pages obtained by each swap operation and the FPR ratio, under each trace. The FPR ratio is the ratio of the number of swap operations to the number of block erase operations under a trace. In ROSE, an obsolete block can be migrated to the log area through the swap operation at most once before being erased, and therefore FPR ratio can never be larger than 1. A higher FPR ratio means that swap operation occurs more frequently in that workload. From Table 5.4, the cost (i.e., the number of copied pages) of each swap operation is usually small compared to the benefit (i.e., the number of obtained free pages) it brings. FPR is more effective in

reducing the cleaning cost under the *Postmark* and *Fin2* traces due to their higher FPR ratios.

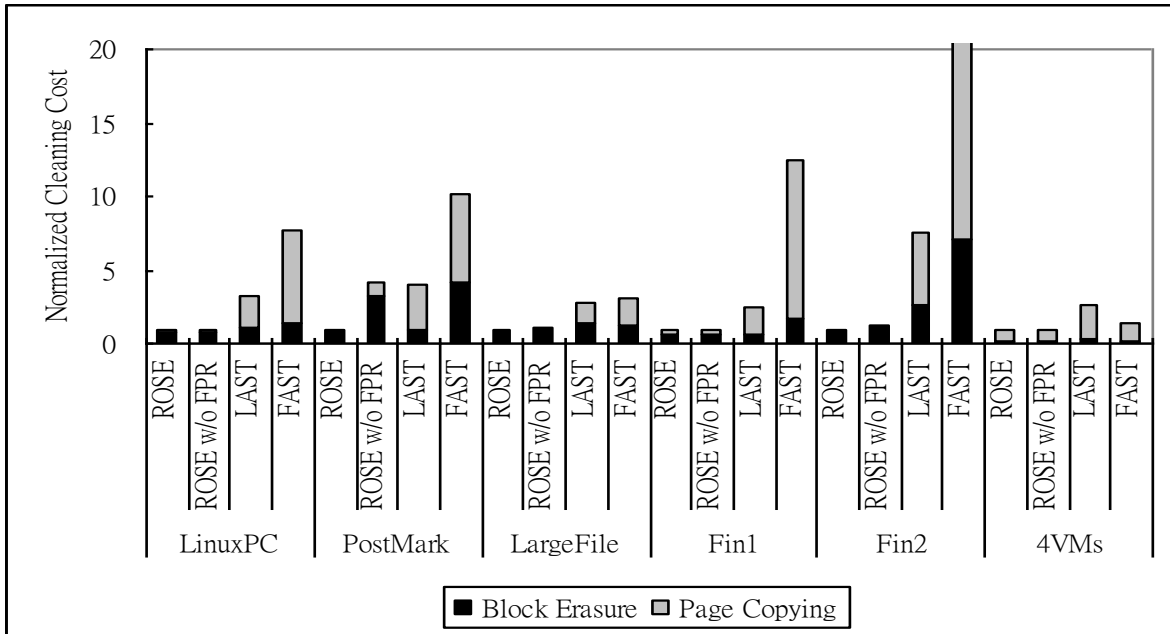
**TABLE 5.4: STATISTICS OF FPR**

<i>Traces</i>	<i>Free pages obtained by each swap (average)</i>	<i>Pages copied during each swap (average)</i>	<b>FPR ratios</b>
LinuxPC	40.89	3.04	0.03
PostMark	61.85	0.17	0.93
LargeFile	26.52	0.23	0.01
Fin1	0.11	0.38	0.04
Fin2	50.03	0.12	0.32
4VMs	33.18	0.30	0.01

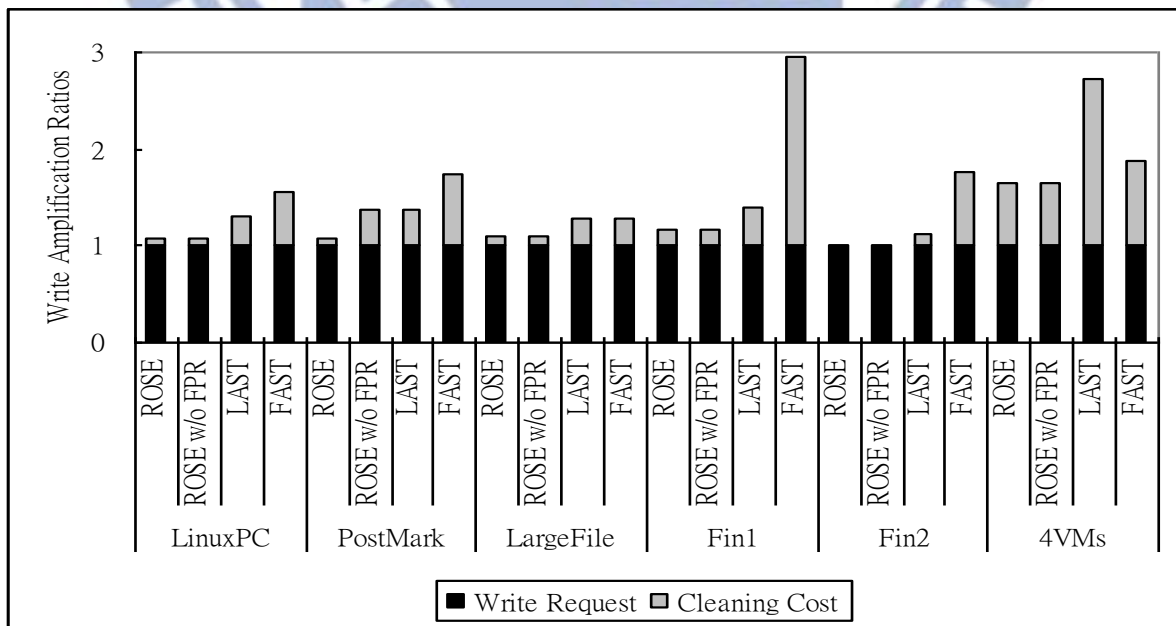
## 5.2.4 Overall Performance of ROSE

In this section, we compare the overall performance of FAST, LAST and ROSE. Figure 5.9 shows the cleaning cost normalized to that of ROSE, in which all the proposed techniques are enabled. We also show the cleaning cost of ROSE with FPR disabled for performance comparison. From the figure, ROSE outperforms FAST and LAST by 34% to 47 times and 2 to 6 times, respectively. Even with FPR disabled, ROSE still outperforms FAST and LAST significantly under almost all the traces. Figure 5.10 shows the write amplification ratio (WAR), which is defined in Section 2.1, of each FTL under each trace. As shown in the figure, ROSE achieves the lowest WAR among the FTLs under all

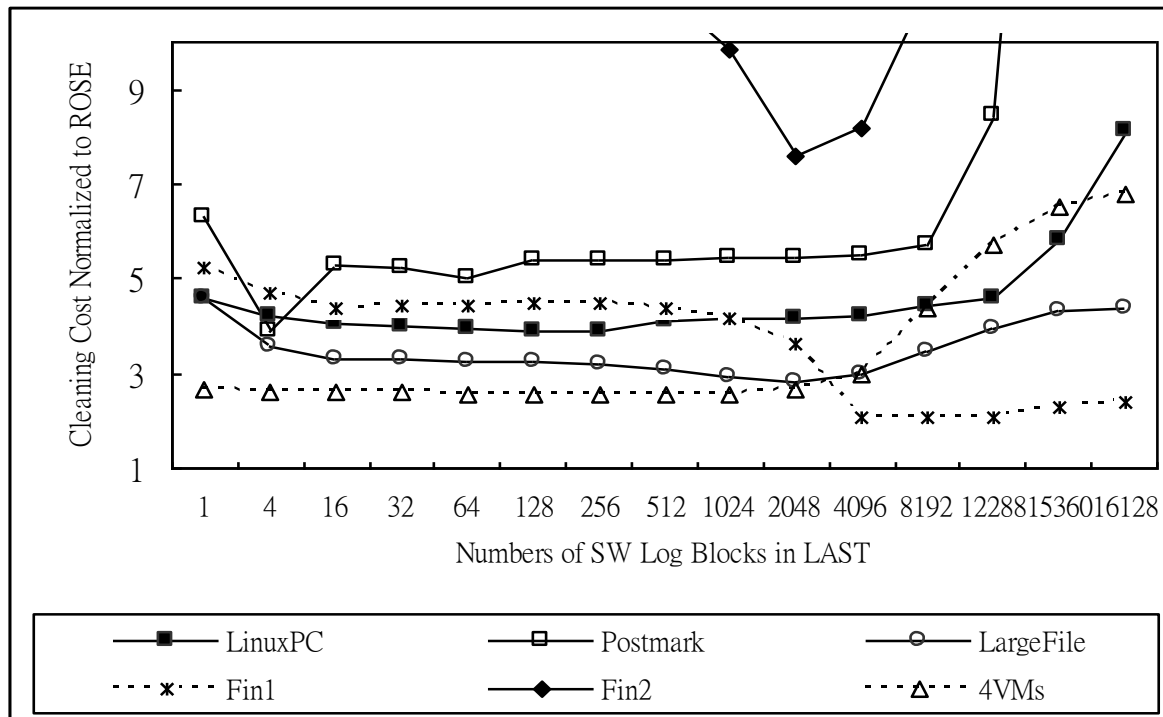
the traces. Specifically, it reduces the WAR by up to 1.1 and 1.8 when compared to LAST and FAST, respectively, leading to up to 39% and 61% reduction in the total write time.



**Figure 5.9: Cleaning cost of FAST, LAST and ROSE (normalized to the cleaning cost of ROSE)**



**Figure 5.10: Write amplification ratios of FAST, LAST and ROSE**



**Figure 5.11: Effect of the number of sequential write log blocks**

Figure 5.11 shows the cleaning cost of LAST with different numbers of SW log blocks. The results are normalized to the cleaning cost of ROSE. In the figure, all the values are larger than 1, meaning that ROSE always results in superior performance than LAST. Moreover, increasing the number of SW log blocks could help reducing the cleaning cost when there are few SW log blocks. However, when a large number of SW log blocks have already been used, further increasing the number of SW log blocks increases the cleaning cost. This is not surprising since little space is left for random writes if too many SW log blocks are used. Table 5.5 presents the cleaning cost of FAST, LAST, and

ROSE under different log area sizes, ranging from 1.5% to 3.5% of the storage size. From the table, the cleaning cost usually decreases with the growth of the log area size. Moreover, ROSE consistently outperforms the other two FTLs when the log area size is equal to or larger than 2% of the storage size.

**TABLE 5.5: CLEANING COST WITH DIFFERENT LOG AREA SIZES (SECONDS)**

<i>Traces</i>	<i>FTLs</i>	<b>Log area sizes (% of the storage size)</b>				
		<b>1.5%</b>	<b>2%</b>	<b>2.5%</b>	<b>3%</b>	<b>3.5%</b>
LinuxPC	ROSE	771	623	541	463	418
	LAST	2442	2404	2327	2256	2163
	FAST	4217	4167	4164	4077	4057
Postmark	ROSE	96	89	78	69	61
	LAST	358	350	344	340	334
	FAST	790	669	643	450	434
LargeFile	ROSE	401	394	388	381	375
	LAST	1127	1108	1101	1102	1107
	FAST	1227	1177	1199	1134	1116
Fin1	ROSE	830	573	446	410	381
	LAST	2139	1925	1671	1227	815
	FAST	5870	5638	5573	5543	5518
Fin2	ROSE	52	22	6	2	2
	LAST	88	56	43	35	28
	FAST	332	308	288	280	280
4VMs	ROSE	7644	6254	5029	3822	2801
	LAST	20749	16328	13322	10783	9361
	FAST	7278	6982	6763	6643	6528

Finally, Table 5.6 shows the result of wear-leveling in ROSE. In this



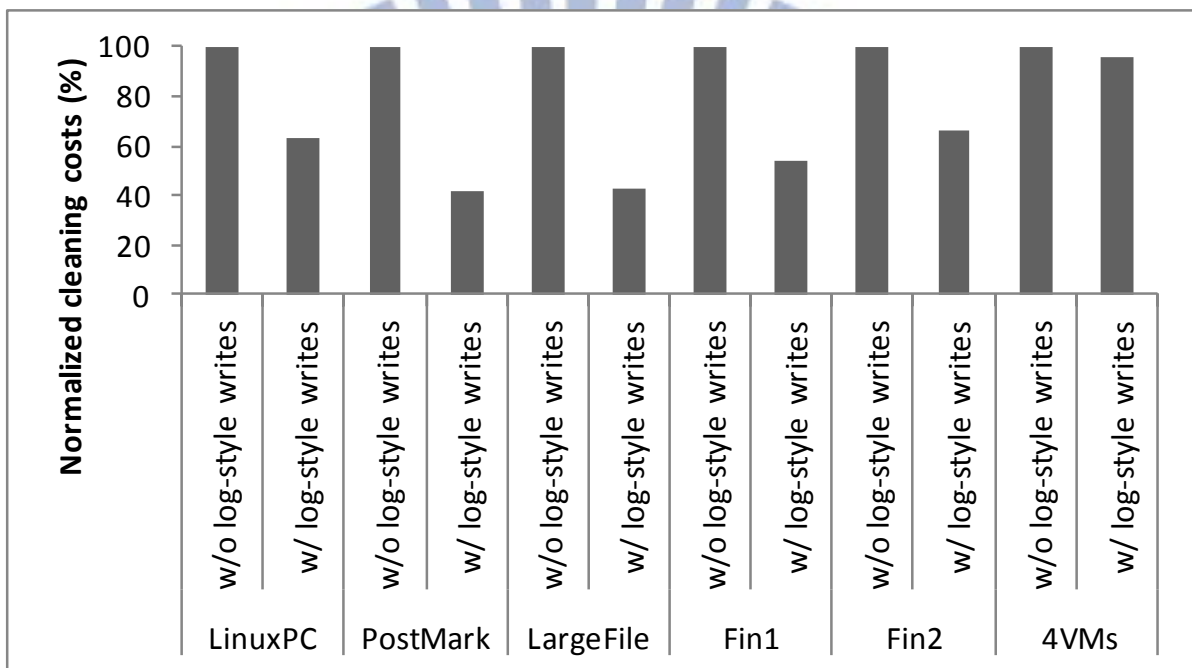
experiment, each trace is executed repeatedly until the average erase count of the blocks is larger than 20. From the table, the standard deviations of the erase counts are small for all the traces, showing that ROSE achieves wear-leveling with the support of the global wear-leveling technique. Moreover, the overhead of the global wear-leveling technique (i.e., the cost of swapping the data of hot and cold blocks) is insignificant compared to the overall cleaning cost (i.e., less than or equal to 2.01% of the cleaning cost under all the traces) since hot-cold swapping are not triggered frequently.

**TABLE 5.6: RESULT OF WEAR-LEVELING**

<i>Traces</i>	<i>Numbers of iterations</i>	<i>Ave. erase counts / Std. dev.</i>	<b>Cost of hot-cold swapping (% of the cleaning cost)</b>
LinuxPC	21	20.18/2.43	2.01
PostMark	225	20.01/0.28	0.02
LargeFile	56	20.13/2.19	1.13
Fin1	75	20.02/0.56	0.12
Fin2	591	20.00/1.30	0.41
4VMs	16	20.46/2.18	0.95

## 5.3 Performance Evaluation of HybridLog

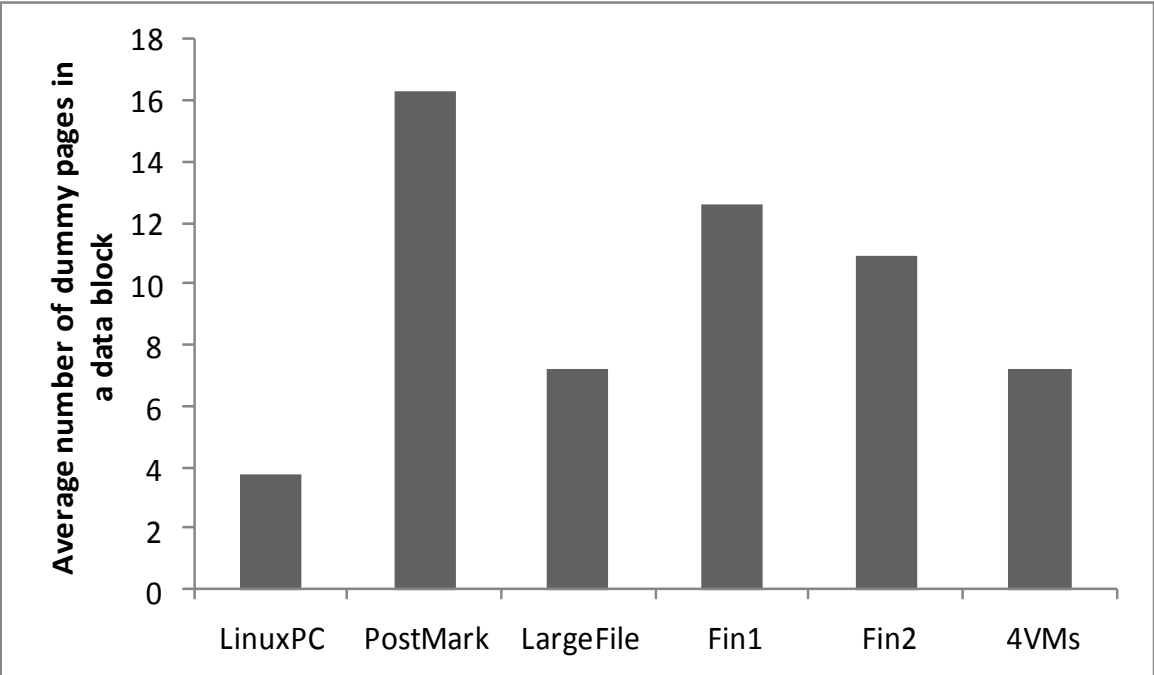
### 5.3.1 Effect of Log-Style Write



**Figure 5.12: Normalized cleaning cost w/ and w/o log-style writes**

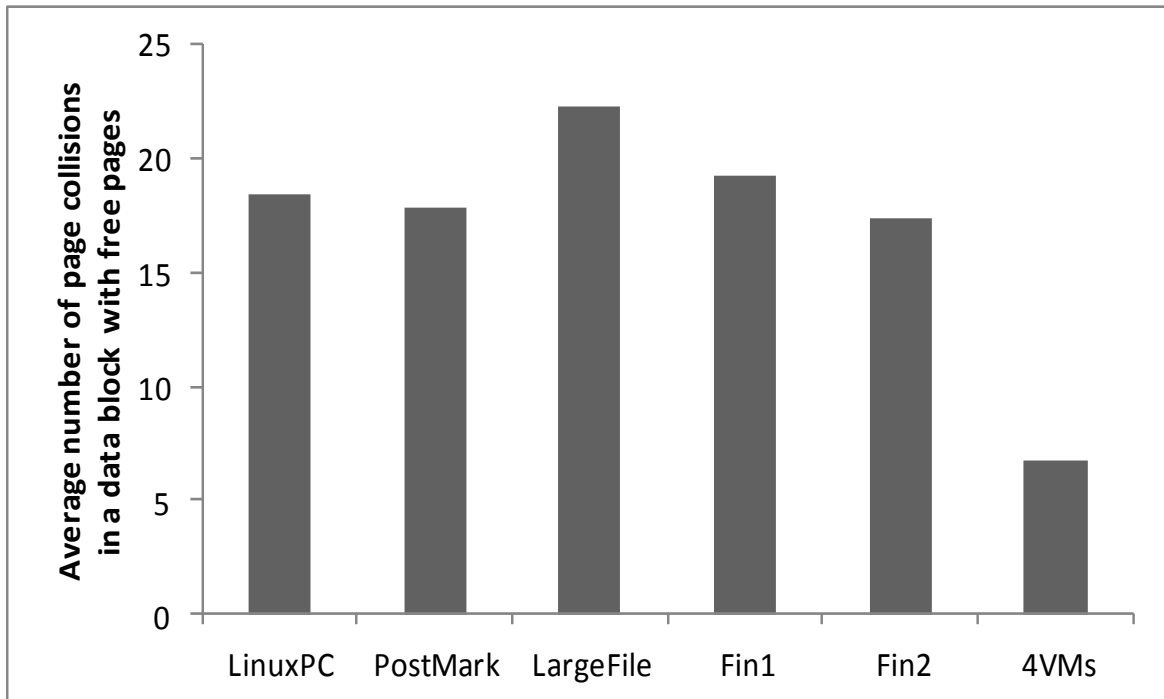
To evaluate the performance of log-style write, we measure the cleaning cost with and without the presence of log-style write in the HybridLog FTL. Since the values of the traces have different orders of magnitude, they are normalized to the cleaning cost without log-style write. As shown in Figure 5.12, using log-style write is effective in five of the six workloads, i.e., *LinuxPC*, *PostMark*, *LargeFile*, *Fin1* and *Fin2*. In these workloads, using log-style write

can reduce the cleaning cost by up to 58%.



**Figure 5.13: Average number of dummy pages in a data block**

The reason can be seen in Figure 5.13 and Figure 5.14. Figure 5.13 shows the average number of dummy pages that have been written when a data block becomes full. As mentioned before, dummy pages have to be written in each page padding operation to follow consecutive programming. From the figure, about 4 to 16 dummy pages in average were written in a data block. This wastes the flash memory space since these dummy pages do not accommodate any new user data. Moreover, further page writes to the space occupied by the dummy pages cause page collisions and thus have to be satisfied by the log area. With log-style write in HybridLog, dummy page writes can totally be eliminated.



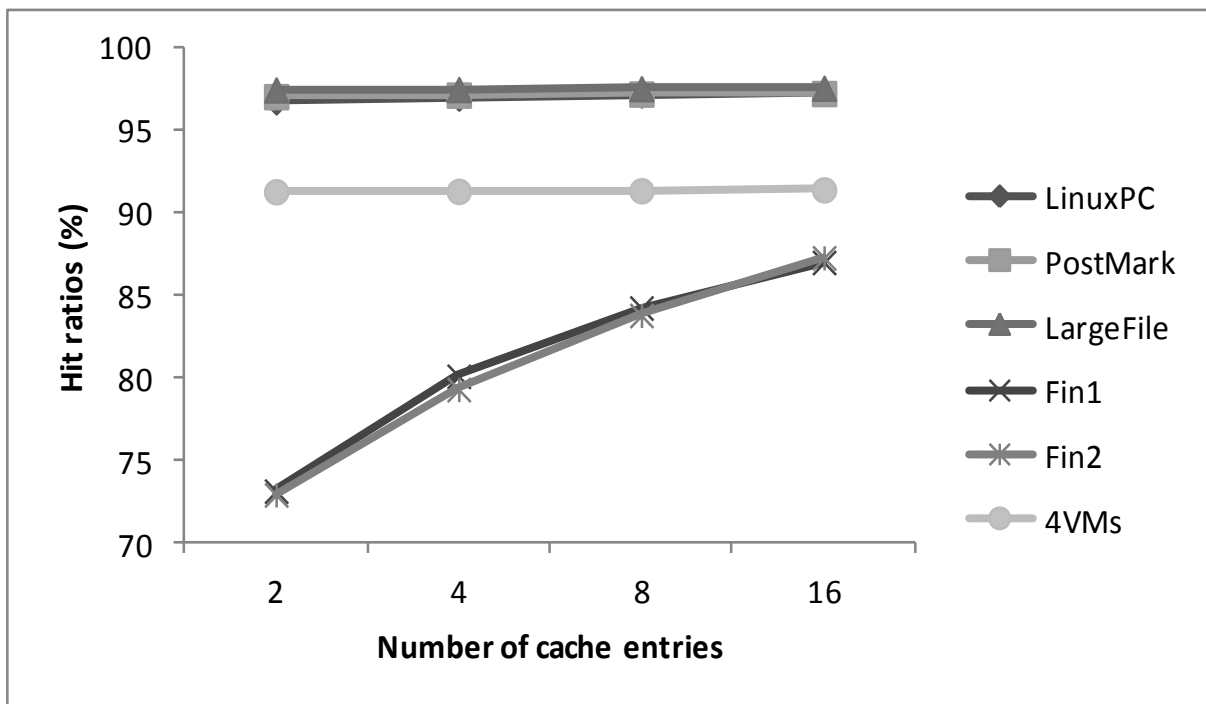
**Figure 5.14: Average number of pages collision in a data block with free pages**

Figure 5.14 shows the average number of page collisions in a data block with free pages. Without log-style write, the collided pages have to be written to the log area even in the case that the data blocks still have free space to accommodate the collided pages. With log-style write in HybridLog, the collided pages can be accommodated by data blocks if there is free space in the data blocks. This reduces the write traffic to the log area and thus leading to lower cleaning cost.

In summary, as mentioned in Section 4.2, log-style write allows every page of the data block to accommodate meaningful user data. Moreover, it reduces the write traffic to the small-sized log area and hence delays cleaning due to the

fullness of the log area, resulting in a lower cleaning cost.

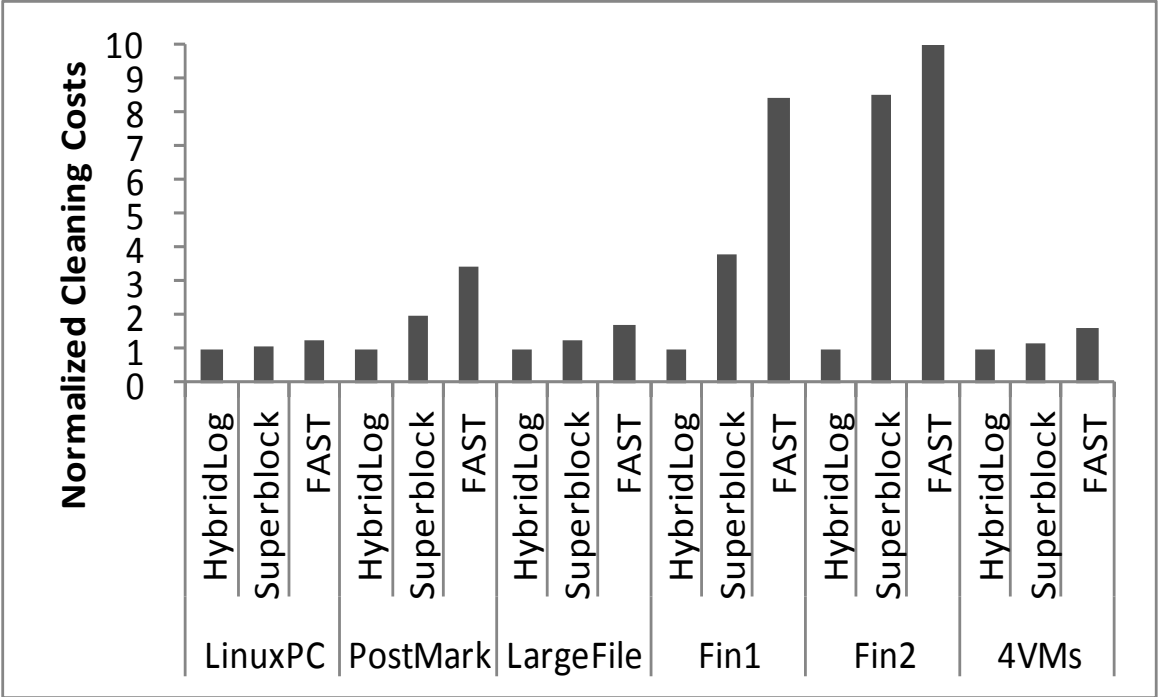
### 5.3.2 Cache Hit Ratio



**Figure 5.15: Cache hit ratios in HybridLog**

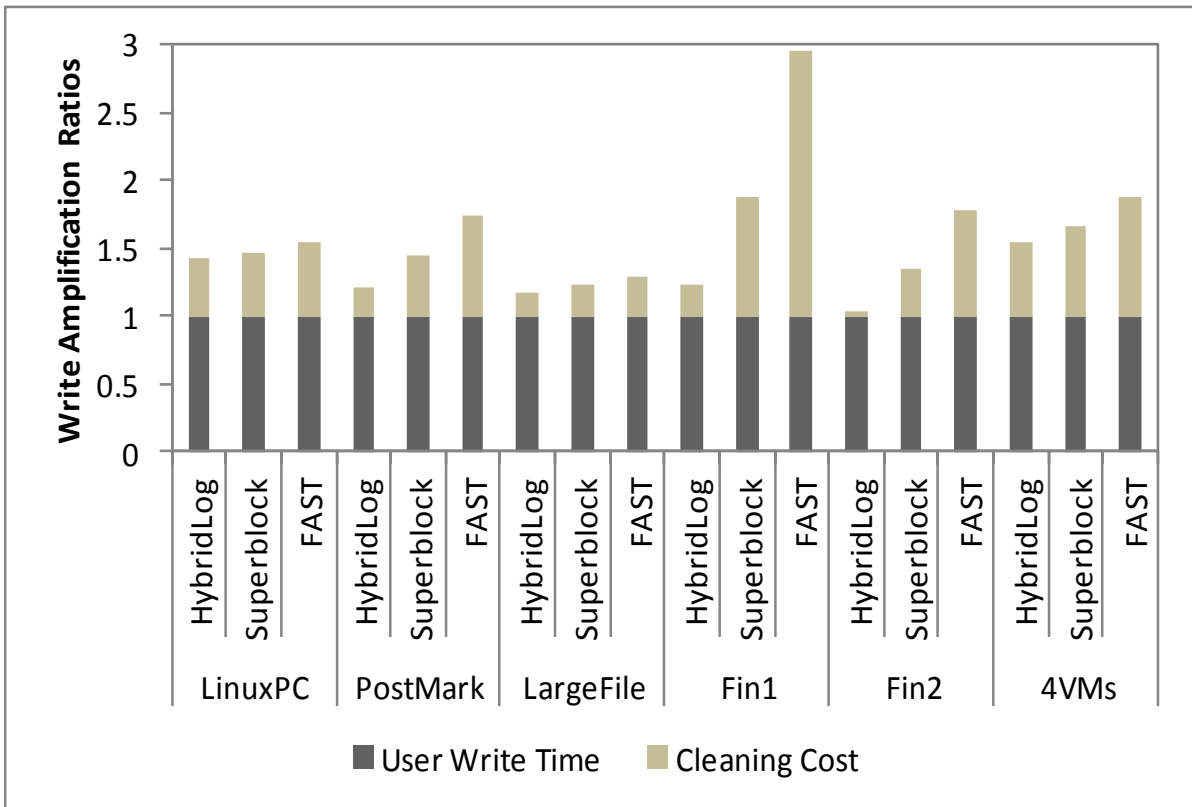
HybridLog caches recently used intra-block mapping in memory to reduce spare area reads. Figure 5.15 presents the cache hit ratios with various numbers of cache entries, ranging from 2 to 16. Not surprisingly, the hit ratio improves with increased cache size. As shown in the figure, very few cache entries are sufficient to achieve a high hit ratio due to temporal and spatial locality of the traces, which is common in many real workloads.

### 5.3.3 Overall Performance of HybridLog



**Figure 5.16: Normalized cleaning cost of Superblock, FAST, and HybridLog**

In this section, the overall performance of FAST, Superblock and HybridLog is compared. Figure 5.16 shows the cleaning cost of the three FTLs under each trace. The results are normalized to the cleaning cost of HybridLog. From the figure, HybridLog outperforms FAST and Superblock by 30% (under LinuxPC) to 17.8 times (under Fin2) and 10% (under LinuxPC) to 7.5 times (under Fin2), respectively.



**Figure 5.17: Write amplification ratios of Superblock, FAST, and HybridLog**

Figure 5.17 shows the write amplification ratio (WAR), as defined in (1), of each of the FTLs under each trace. As shown in the figure, HybridLog achieves the lowest WAR among the FTLs under all the traces. Specifically, it reduces the WAR by up to 1.73 and 0.65 (under Fin1) when compared to FAST and Superblock, respectively, leading to up to 58% and 35% (under Fin1) reduction in the total write time.

# Chapter 6

## Conclusions and Future Work

### 6.1 Conclusions

In this thesis, two HAT-based flash translation layers, ROSE and HybridLog, are proposed. ROSE integrates three novel techniques, namely EBW, MARO and FPR, to reduce the cleaning cost. Existing HAT-based FTLs handle sequential overwrites by predicting sequentiality. EBW takes a fundamentally different approach. It detects sequentiality instead of predicting it, eliminating the cleaning cost resulting from mispredictions that treat random or semi-sequential writes as sequential ones. The MARO cleaning policy selects a victim log block by considering the states of both the log blocks and their corresponding data blocks, improving the cleaning efficiency. In contrast to existing cleaning policies in HAT-based FTLs, both the ages and the merge costs of the log blocks are considered at the same time. Finally, the FPR technique reuses the free pages of obsolete blocks to buffer further page overwrites, which increases the space utilization and reduces the cleaning cost. Through trace-driven simulation, we have demonstrated the effectiveness of each proposed technique of ROSE. Moreover, the results also show that ROSE can outperform previous HAT-based FTLs by up to 47 times in terms of



cleaning cost and by up to 1.6 times in terms of flash write time.

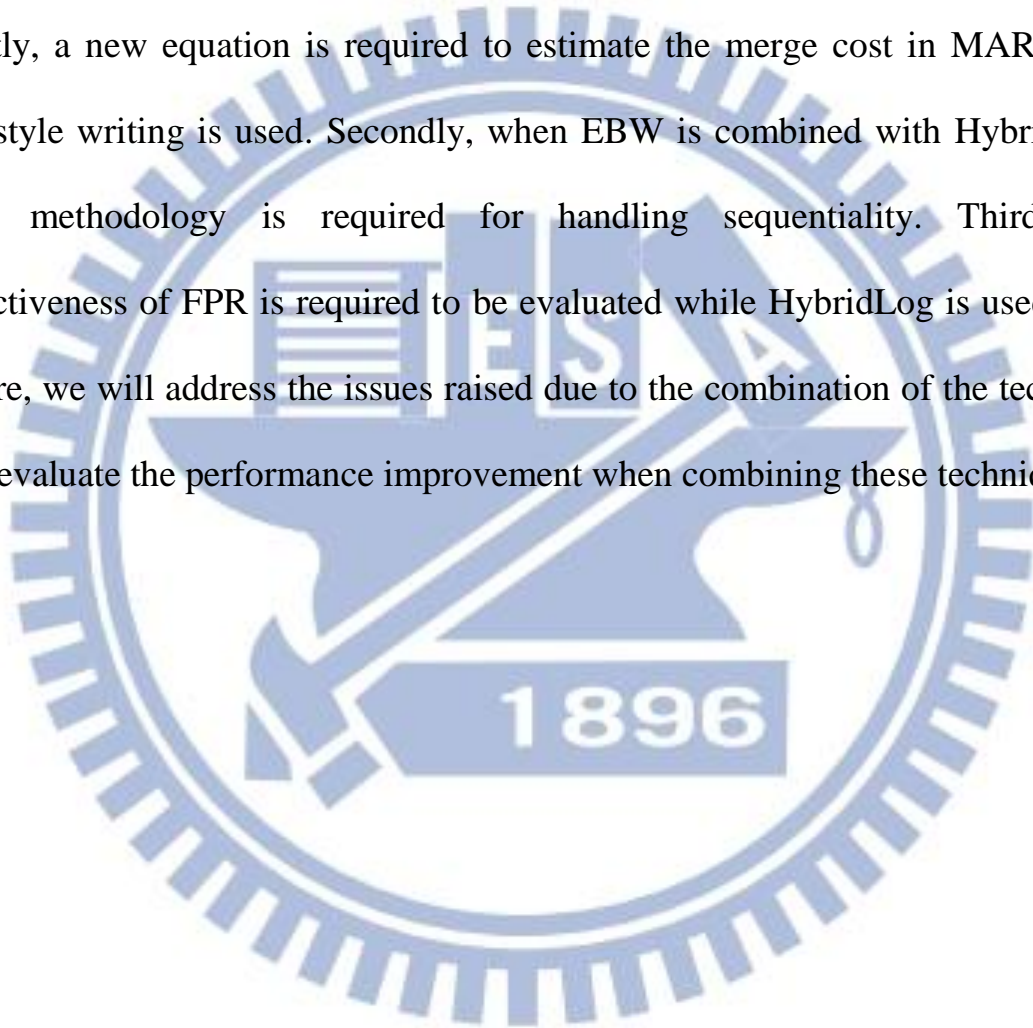
HybridLog is proposed to support modern NAND flash memory and to achieve low cleaning cost. To allow consecutive programming required by modern NAND flash memories, log-style write is used for all the blocks in the flash memory, including the data blocks. To support log-style write to all the blocks, intra-block mapping information is stored in the spare area of each written page. Since only a small space is required in the spare area for the mapping information, many modern SLC/MLC flash memories can be supported. In addition to allow consecutive programming, log-style write to data blocks also eliminate writes of dummy pages to the data blocks and reduce the write traffic to the small-sized log area due to page collisions, which are both helpful for reducing the cleaning cost. Through trace-driven simulation on six real or benchmark-based workloads, the effectiveness of log-style write and the superior performance of HybridLog compared to the other HAT-based FTLs have been demonstrated. Specifically, HybridLog outperforms existing HAT-based FTLs by up to 17.8 times in terms of cleaning cost and reduces the WAR by up to 1.73.

## 6.2 Future Work

Combining the techniques used in ROSE and HybridLog might reduce the cleaning costs further. For example, integrating log-style writing into ROSE

allows ROSE to support modern NAND flash memory efficiently. Combining EBW with HybridLog also allows the HybridLog to reduce the write traffic to the small-sized log area, since HybridLog provides the log-style write to accommodate the most up-to-date user data in all data pages.

However, there are still some challenges for combining these techniques. Firstly, a new equation is required to estimate the merge cost in MARO when log-style writing is used. Secondly, when EBW is combined with HybridLog, a new methodology is required for handling sequentiality. Thirdly, the effectiveness of FPR is required to be evaluated while HybridLog is used. In the future, we will address the issues raised due to the combination of the techniques and evaluate the performance improvement when combining these techniques.



# Bibliography

1. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J., Manasse, M., and Panigrahy, R., “Design Tradeoffs for SSD Performance,” in *USENIX, 2008*.
2. “A Simulator for Various FTL Schemes,” <http://csl.cse.psu.edu/?q=node/322>.
3. Ban, A., “Flash File System,” U.S. Pat. No. 5,404,485, 1995.
4. Ban A., and Hasharon, R., “Flash File System Optimized for Page-Mode Flash Technologies,” U.S. Pat. No. 5,937,425, 1999.
5. Bates, K., and McNutt, B., OLTP I/O Traces, available at <http://traces.cs.umass.edu/index.php/storage/storage>, 2007.
6. Blackwell, T., Harris, J., and Seltzer, M.I., “Heuristic Cleaning Algorithms in Log-Structured File Systems,” in *Proceedings of the USENIX 1995 Annual Technical Conference*, pages 277–288, 1995.
7. Chang, L.P., “Hybrid Solid-State Disks: Combining Heterogeneous NAND Flash in Large SSDs,” in *the 13th Asia and South Pacific Design Automation Conference*, 2008.
8. Chang, L.P., “On efficient wear-leveling for large-scale flash-memory storage systems,” in *the 22st ACM Symposium on Applied Computing*, March 2007.
9. Chang, L.P., and Kuo, T.W., “An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems,” in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.
10. Chang, L.P., and Kuo, T.W., “An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems,” *ACM Symposium on Applied Computing*, pages 862–868, Mar 2004.
11. Chang, L.P., and Kuo, T.W., “A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems,” in *the 8th International Conference on*

*Real-Time Computing Systems and Applications*, 2002.

12. Chang, Y.H., Hsieh, J.W., and Kuo, T.W., “Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design,” in *the 44th ACM/IEEE Design Automation Conference*, June 2007.
13. Chen, F., Koufaty, D.A., and Zhang, X., “ Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives,” in *Proceedings of the 2009 Joint SIGMETRICS/Performance Conference*, pages 181–192, 2009.
14. Chen, F., Luo, T., and Zhang, X., “CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives,” in *Proceedings of the 9th USENIX conference on File and storage technologies*, pages 6–6, 2011.
15. Chiang, M.L., and Chang, R.C., “Cleaning Policies in Mobile Computers Using Flash Memory,” *J. Systems and Software*, volume 48, number 3, pages 213-231, 1999.
16. Chiang, M.L., Paul, C.H., and Chang, R.C., “Manage flash memory in personal communicate devices,” in *Proceedings of International Symposium on Consumer Electronics*, 1997.
17. Chien, R.T., “Cyclic decoding procedure for the Bose-Chaudhuri-Hoc-quenghem codes,” *IEEE Transactions on Information Theory*, volume 10, number 4, pages 357-363, October 1964.
18. Cho, H., Shin, D., and Eom, Y.I., “KAST: K-associative sector translation for NAND flash memory in real-time systems,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 507–512, 2009.
19. Chu, Y.S., Hsieh, J.W., Chang, Y.H., and Kuo, T.W., “A set-based mapping strategy for flash-memory reliability enhancement,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 405–410, 2009.
20. Dees, B., “Native command queuing - advanced performance in desktop storage,”

- Potentials, IEEE*, volume 24, number 4, pages 4–7, October–November 2005.
21. Du, Y., Cai, M., and Dong J., “Adaptive Garbage Collection Mechanism for N-log Block Flash Memory Storage Systems,” in *Proceedings of the 16th International Conference on Artificial Reality and Telexistence*, 2006.
  22. Filebench File System Benchmark, available at <http://hub.opensolaris.org/bin/view/Community+Group+performance/filebench>, 2009.
  23. Gal, E., and Toledo, S., “Algorithms and data structures for flash memories,” *ACM Computer Survey*, volume 37, number 2, pages 138–163, 2005.
  24. Ganger, G.R., Worthington, B.L., and Patt, Y.N., “The DiskSim Simulation Environment Version 3.0 Reference Manual”, 2003.
  25. Gupta, A., Kim, Y., and Urgaonkar, B., “DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings,” in *Proceeding International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, March 2009.
  26. Hong, S., and Shin, D., “NAND Flash-Based Disk Cache Using SLC/MLC Combined Flash Memory,” in *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, pages 21–30, 2010.
  27. Hsieh, J.W., Kuo, T.W., and Chang L.P., “Efficient identification of hot data for flash memory storage systems,” *IEEE Transactions on Storage*, volume 2, pages 22–40, February 2006.
  28. Hu, J., Jiang, H., Tian, L., and Xu, L., “Pud-lru: An erase-efficient write buffer management algorithm for flash memory ssd,” in *Proceedings of the 18th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.
  29. Hu, X.Y., Eleftheriou, E., Haas, R., Iliadis, I., and Pletka, R., “Write Amplification

- Analysis in Flash-based Solid State Drives,” in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 101–109, 2009.
30. Intel Corporation, “FTL Logger Exchanging Data with FTL Systems”.
  31. Intel Corporation, “Software Concerns of Implementing a Resident Flash Disk”.
  32. Intel Corporation, “Understanding the Flash Translation Layer (FTL) Specification,” Application Note AP-684, December 1998.
  33. Jo, H., Kim, J.S., et al., “FAB: Flash-Aware Buffer Management Policy for Portable Media Players,” *IEEE Transactions on Consumer Electronics*, volume 52, number 2, pages 485-493, May 2006.
  34. Jung, D., Chae, Y., Jo, H., Kim, J., and Lee, J., “A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems,” in *Proceedings of the 2007 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2007.
  35. Jung, D., Kang, J.U., Jo, H., Kim, J.S., and Lee, J., “Superblock FTL: a superblock-based flash translation layer with a hybrid address translation scheme,” *ACM Transactions on Embedded Computer System*, volume 9, number 4, article number 40, March 2010.
  36. Kang, D., Jung, D., Kang, J.U., and Kim, J.S., “ $\mu$ -Tree: An Ordered Index Structure for NAND Flash Memory,” in *Proceeding Seventh ACM and IEEE International Conference Embedded Software*, pages 144-153, October 2007.
  37. Kang, J.U., Jo, H., Kim, J.S., and Lee, J., “A Superblock-based Flash Translation Layer for NAND Flash Memory, ” in *Proceeding Sixth ACM and IEEE International Conference Embedded Software*, pages 161-170, 2006.
  38. Kang, S., Park, S., Jung, H., Shim, H., and Cha, J., “Performance trade-offs in using nvram write buffer for flash memory-based storage devices,” *IEEE Transactions on*

- Computers*, volume 58, number 6, pages 744–758, 2009.
39. Katcher, J., “PostMark: A New File System Benchmark,” available at [http://rpmfind.net/linux/RPM/opensuse/factory/x86\\_64/postmark-1.51-19.42.x86\\_64.html](http://rpmfind.net/linux/RPM/opensuse/factory/x86_64/postmark-1.51-19.42.x86_64.html), 2009.
  40. Kawaguchi, A., Nishioka, S., and Motoda, H., “A Flash-Memory Based File System,” in *Proceedings of 1995 USENIX Winter Technical Conference*, pages 155-164, January 1995.
  41. Kgil, T., Roberts, D., and Mudge, T., “Improving nand flash based disk caches,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 327–338, 2008.
  42. Kim, H., and Ahn, S., “BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage,” in *Proceedings of Sixth USENIX Conference File and Storage Technologies*, pages 239-252, 2008.
  43. Kim, H.J., and Lee, S.G., “An Effective Flash Memory Manager for Reliable Flash Memory Space Management,” *IEICE Transactions on Information and Systems*, volume E85-D, number 6, pages 950–964, 2002.
  44. Kim, J.H., Jung, D., Kim J.S., and Huh, J., “A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs),” in *Proceedings of the 2009 MASCOTS Symposium*, pages 1–10, 2009.
  45. Kim, J., Kim, J.M., Noh, S.H., Min, S.L., and Cho, Y., “A Space-Efficient Flash Translation Layer for Compact-Flash Systems,” *IEEE Transactions on Consumer Electronics*, volume 48, number 2, pages 366-375, May 2002.
  46. Kim, Y., Tauras, B., Gupta, A., and Urgaonkar, B., “Flashsim: A simulator for NAND flash-based solid-state drives,” in *Proceedings of the 2009 First International Conference on Advances in System Simulation*, pages 125–131, 2009.

47. Koo, D., and Shin, D., "Adaptive log block mapping scheme for log buffer-based FTL (flash translation layer)," in *International Workshop on Software Support for Portable Storage*, 2009.
48. Kotz, D., Toh, S.B., and Radhakishnan, S., "A detailed simulation model of the hp 97560 disk drive," *Technical report pcs-tr94-220, dept. of computer science, darthmouth college. Technical report*, 1994.
49. Kwon, H., Kim, E., Choi, J., Lee, D., and Noh, S.H., "Janus-FTL: fi D. L the optimal point on the spectrum between page and block mapping schemes," in *Proceedings of the tenth ACM international conference on Embedded software*, pages 169–178, 2010.
50. Kwon, S.J., and Chung, T.S., "An Efficient and Advanced Space Management Technique for Flash Memory Using Reallocation Blocks," *IEEE Transactions on Consumer Electronics*, volume 54, pages 631-638, 2008.
51. Lee, H.S., Yun, H.S., and Lee, D.H., "HFTL: hybrid flash translation layer based on hot data identification for flash memory," *IEEE Transactions on Consumer Electronics*, volume 55, number 4, pages 2005-2011, November 2009.
52. Lee, S., Shin, D., Kim, Y.J., and Kim, J., "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *ACM SIGOPS Operating Systems Review*, volume 42, number 6, pages 36-42, October 2008.
53. Lee, S.W., Park, D.J., Chung, T.S., Lee, D.H., Park, S., and Song, H.J., "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, volume 6, number 3, article number 18, July 2007.
54. Lee, Y.G., Jung, D., Kang, D., and Kim, J.S., "μ-FTL: A Memory Efficient Flash Translation Layer Supporting Multiple Mapping Granularities," in *Proceedings of Eighth ACM and IEEE International Conference Embedded Software*, pages 21-30, October

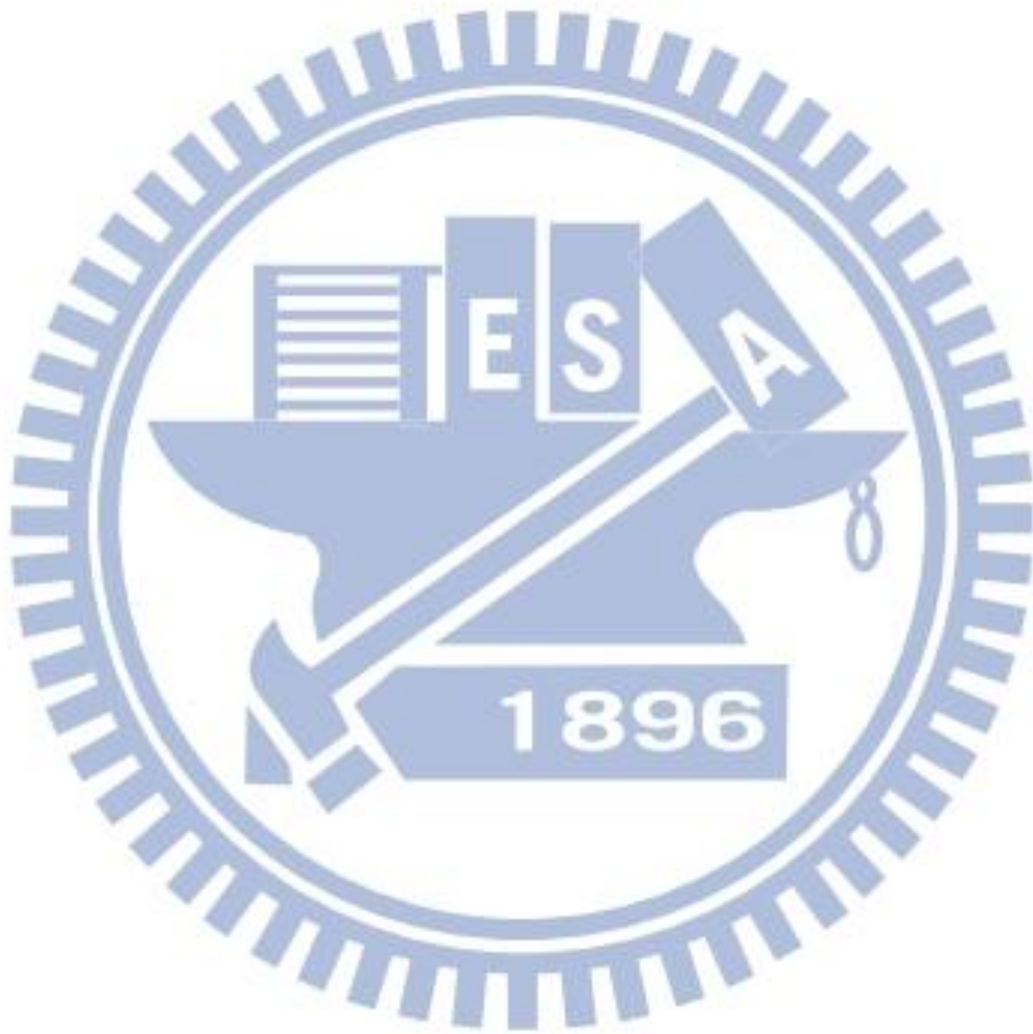


- 2008.
55. Lim, S., Lee, S., and Moon, B., “FASTER FTL for Enterprise-Class flash Memory SSDs,” *International Workshop on Storage Network Architecture and Parallel I/Os*, 2010.
  56. Lin, J.H., Chang, Y.H., Hsieh, J.W., Kuo, T.W., and Yang, C.C., “A NOR Emulation Strategy over NAND Flash Memory,” *the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2007.
  57. Lin, P.K., Chiao, M.L., and Chang, D.W., “Improving flash translation layer performance by supporting large superblocs,” *IEEE Transactions on Consumer Electronics*, volume 56, number 2, pages 642-650, May 2010.
  58. Liu, C.Y., Pan, Y.S., Chen, H.H., Wu, Y.C., and Chang, D.W., “Techniques for improving performance of the FAST (fully associative sector translation) flash translation layer,” *IEEE Transactions on Consumer Electronics*, volume 57, number 4, pages 1740-1748, November 2011.
  59. Liu, Z., Yue, L., Wei, P., Jin, P., and Xiang, X., “An Adaptive Block-Set based Management for Large-Scale Flash Memory,” in *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1621–1625, 2009.
  60. Ma, D., Feng, J., and Li, G., “LazyFTL: a page-level flash translation layer optimized for NAND flash memory,” in *Proceedings of the 2011 international conference on Management of data*, pages 1–12, 2011.
  61. M-Systems, “Flash-memory Translation Layer for NAND flash (NFTL),” 1998.
  62. M. T. Inc., “Small-block vs. large-block nand flash devices. Technical report (tn-29-07): Small-block vs. large-block NAND flash devices,” May 2007.
  63. ONFI Standard 3.2, *Open NAND Flash Interface Working Group*, Jan. 2013.
  64. Park, C., Cheon, W., Lee, Y., Jung, M.S., Cho, W., and Yoon, H., “A Re-configurable

- FTL (Flash Translation Layer) Architecture for NAND Flash based Applications,” *ACM Transactions on Embedded Computing Systems*, volume 7, number 4, July 2008.
65. Park, C., Lim, J., Kwon, K., Lee, J., and Min, S.L., “Compiler-assisted demand paging for embedded systems with flash memory,” in *the international conference on Embedded Software*, September 2004.
  66. Park, C., Seo, J., Seo, D., Kim, S., and Kim, B., “Cost-efficient memory architecture design of nand flash memory embedded systems,” in *Proceedings of the 21st International Conference on Computer Design*, pages 474–480, October 2003.
  67. Park, D., Debnath, B., and Du, D., “CFTL: A Convertible Flash Translation Layer with Consideration of Data Access Patterns,” *UMN CS Technical Report*, no. TR 09-023, 2009.
  68. Park, S., “K-Leveling: An Efficient Wear-Leveling Scheme for Flash Memory,” in *Proceedings of the 2005 US-Korea Conference on Science, Technology, and Entrepreneurship*, 2005.
  69. Park, S., Jung, D., Kang, J., Kim, J., and Lee, J., “CFLRU: a replacement algorithm for flash memory,” in *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241, 2006.
  70. Qin, Z., Wang, Y., Liu, D., and Shao, Z., “Demand-based block-level address mapping in large-scale nand flash storage systems,” in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 173–182, 2010.
  71. Rizvi, S.S., and Chung, T.S., “A Reliable Storage Management Framework for Flash based Embedded and Multimedia Systems experiencing Diverse Data Nature,” in *Proceedings of Ubiquitous Information Management and Communication*, Feb. 21~23, 2011.

72. Rizvi, S.S., Chung, T.S., “A Survey of Storage Management in Flash based Data Centric Sensor Devices in Wireless Sensor Networks,” in *Proceedings of Communication Systems, Networks and Applications*, volume 1, pages 1-4, 2010.
73. Samsung Electronics, “512M x 8 Bit / 256M x 16 Bit NAND Flash Memory,” Datasheet, available at [http://www.datasheetcatalog.org/datasheets/700/389215\\_DS.pdf](http://www.datasheetcatalog.org/datasheets/700/389215_DS.pdf), 2005.
74. Shim, G., Park, Y., and Park, K.H., “A hybrid flash translation layer with adaptive merge for SSDs,” in *IEEE Transactions on Storage*, volume 6, number 4, pages 15:1–15:27, June 2011.
75. Shin, J.Y., Xia, Z.L., Xu, N.Y., Gao, R., Cai, X.F., Maeng, S., and Hsu, F.H., “FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications,” in *Proceedings of the 23rd International Conference on Supercomputing*, pages 338–349, 2009.
76. T13 Technical Committee, ATA/ATAPI Command Set-2, 2010.
77. Torelli, P., “The Microsoft Flash File System,” *Dr. Dobb's Journal*, pages 62-72, 1995.
78. Toshiba, “1G x 8 Bit NAND Flash Memory (TC58NVG3D1DTG00),” Datasheet, 2007.
79. Wang, T.Z., Liu, D., Wang, Y., and Shao, Z., “FTL<sup>2</sup>: a hybrid flash translation layer with logging for write reduction in flash memory,” in *Proceedings of the 14<sup>th</sup> ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 91-100, 2013.
80. Wei, Q., Gong, B., Pathak, S., Veeravalli, B., Zeng, L., and Okada, K., “WAFTL: A workload adaptive flash translation layer with data partition,” *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, volume 0, pages 1–12, 2011.
81. “Wear Leveling in Single Level Cell NAND Flash Memories,” *STMicroelectronics Application Note(AN1822)*, 2006.
82. Wells, S., “Method for Wear Leveling in a Flash EEPROM Memory,” United States

- Patent, no. 5341339, 1994.
83. Won, S., Chung, E.Y., Kim, D., Chung, J., Han, B., and Lee, H., "Page overwriting method for performance improvement of NAND flash memories," *IEICE Electronics Express*, 2013.
  84. Wu, C., and Kuo, T., "An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems," in *Proceedings of International Conference Computer-Aided Design*, 2006.
  85. Wu, C.H., Lin, H.H., and Guo, T.W., "An Adaptive Flash Translation Layer for High-Performance Storage Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 29, number 6, pages 953-965, 2010.
  86. Wu, M., and Zwaenepoel W., "eNVy: A Non-Volatile, Main Memory Storage System," *Proceedings of Sixth International Conference Architectural Support for Programming Languages and Operating Systems*, pages 86-97, December 1994.
  87. Wu, P.L., Chang, Y.H., and Kuo, T.W., "A file-system-aware FTL design for flash-memory storage system," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*, pages 393-398, 2009.
  88. Xie, Q.Y., Liu, Q., Nie, H.S., Sun, Z.L., Zhou, L., and Song, R., "A Novel NAND Flash FTL for Mass Data Storage Devices Based on Hybrid Address Translation," in *Intelligent System Design and Engineering Applications*, 2013.
  89. Zhang, Q., Li, X.D., Wang, L.Z., Zhang, T., Wang, Y., and Shao, Z., "Optimizing translation information management in NAND flash memory storage systems," in *Design Automation Conference*, pages 326-331, 2013.
  90. Zhang, Y., Wu, Y., Li, D., and Shan, H., "Hot Data Detector based High-Performance NAND and PRAM Hybrid Storage," in *International Conference on Advanced Computer Science and Electronics Information*, pages 150-153, 2013.



# Vita

Mong-Ling Chiao was born on February 13, 1971. He received the BS degree in Business Mathematics from Soochow University in 1996 and the MS degree in Computer Science from National Chung Cheng University in 1998. He is currently a Phd student in Computer Science at National Chiao Tung University. He is also a firmware development manager responsible for NAND flash firmware development at Silicon Motion Technology Corporation. His research interests include flash memory storage systems, file systems, and embedded systems.

