FLASH MEMORY MANAGEMENT WITH COOPERATION, ADAPTATION AND ASSISTANCE

CHUNDONG WANG
(B.Sc., XI'AN JIAOTONG UNIVERSITY, CHINA)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Chundong Wang
November 14, 2013

Acknowledgements

First of all, my deepest gratitude goes to my supervisor, Professor Wong Weng Fai, for his persistent and attentive guidance throughout my Ph.D. candidature. Professor Wong always inspires me and encourages to do research. His professional supervision is of great value to my career in the future.

I would like to express my sincere thanks to my dissertation committee members, Professor Tulika Mitra, Professor Roland Yap Hock Chuan and Professor Tei-Wei Kuo. They have spent a lot of time in reviewing my dissertation, and given me insightful comments and suggestions.

I am grateful to teachers during my Ph.D. study. They did teach me not only knowledge but all skills for a researcher. I also would like to thank administrative staffs of the school and the university for their help in the past five years.

Many thanks are due to my fellows in the Embedded Systems Research Labs and SoC, including Edward Sim, Ju Lei, Anderi Hagiescu, Liang Yun, Huynh Phung Huynh, Sudipta Chattopadhyay, Liu Shanshan, Qi Dawei, Ding Huping, Chen Jie, Chen Liang, Pooja Roy, Wang Jianxing, Mamohan Manoharan, Thannimalai Somu Muthukaruppan, Zhong Guanwen, Ramapantulu Lavanya, Guo Xiangfa, Li Bo, Su Bolan and many others that are not listed. I want to express my gratitude to Professor Jürgen Teich in University of Erlangen-Nuremberg, Professor Qi Yong, Professor Song Qinbao and Dr. He Liang in Xi'an Jiaotong University, Dr. Yang Wentong in the National University Health System, and Assistant Professor Yeh Chi-Tsai in Shih Chien University. I also want to thank Wang Dong, Hai Zhen, Cheng Peng, Chen Peng, Hu Ping, Zhang Kaibin and Li Zhenggang. I highly appreciate their encouragement and support.

I would love to extend the warmest thanks to my parents. They always believe me and encourage me to pursue my dreams. Twelve years ago I left my hometown for study. I wish we could live together soon after my graduation.

Finally, I want to thank my wife, Jiang Lina. I might not be able to write this dissertation without her love and understanding. We met ten years ago in our high school. She is always being supportive to me and helping me through all the hard times. *This dissertation is dedicated to her.*

Contents

Abstract

NAND flash memory-based devices are ubiquitous for data storage in smart phones, personal computers and enterprise servers today. This can be attributed to the advantages of NAND flash memory over ferromagnetic material and volatile memory; in particular, they are lightweight, shock-resistance, energy-efficiency and non-volatility. However, NAND flash memory has inherent characteristics that are still serious concerns in its deployment. At the same time, the environments in which storage devices are used have become much more diverse in the past three decades since the invention of flash memory. Efficient and effective strategies to manage flash device are therefore necessary. This motivates us to innovate new approaches within this thesis.

The management of a NAND flash device is traditionally done by an embedded software called the *flash translation layer* (FTL). The FTL is developed in a modular design with each module being responsible for one aspect of flash management. For example, address mapping maps logical addresses of file systems to physical addresses of flash memory; wear leveling attempts to commit all flash blocks to age at a similar rate, and RAM buffer management aims to make the best use of the RAM buffer inside a flash device.

Our first idea is to have the modules of the FTL cooperate with one another. Modules are likely to have different and possibly independent perspectives with regards to flash management. Therefore, a module of the FTL may benefit from the knowledge of another. Based on this idea we have developed OWL. It is a wear leveling algorithm that works within hybrid address mapping. The latter

classifies allocation requests when allocating blocks for data storage. Cooperation between them goes beyond simply exchanging information. Instead, a part of the wear leveling module of OWL is co-developed with the hybrid mapping module so as to incorporate the latter's information and consideration upon deciding which block to be allocated.

Workload adaptation is our second idea. Flash-based storage devices serve workloads to store and access data. The ability of adapting to a given workload is essential due to the diversity of workloads. Address mapping and RAM buffer management are two functionalities of the FTL that relate to data access. We have first designed a hybrid mapping scheme named ADAPT. ADAPT achieves the goal of workload adaptation through separating and handling respective sequential and random requests. TreeFTL is another scheme we have devised to manage the RAM buffer of a flash device. TreeFTL caches metadata of address mapping and real data pages in the RAM space using a tree-like structure. To minimize the overheads of context switch between workloads, TreeFTL has a lightweight mechanism for evicting the LRU victims to make space.

Our third idea is to enlist the help of the operating system (OS). Traditionally the FTL is self-contained and the OS is oblivious of storage devices. As the OS has a global perspective of data and files, we would like to use the OS's knowledge to assist the FTL to manage flash device. The result of this collaboration is a scheme we called SAW, of which the OS analyzes files to figure out quantitative hints for the FTL to perform wear leveling. Correspondingly the FTL customizes its block organization to utilize the hints received from the OS. Hints are packed along within data segments and delivered to the FTL. The FTL unpacks each segment, interprets the hint and conducts block allocation accordingly.

Experiments have been conducted to evaluate our proposals. Results confirm that our approaches in this thesis could gain significant improvements on device lifetime and access performance, respectively, with insignificant overheads.

List of Publications

1. Chundong Wang and Weng-Fai Wong. Observational wear leveling: an efficient algorithm for flash memory management. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 235–242, San Francisco, California, USA, 2012. ACM.

2. Chundong Wang and Weng-Fai Wong. Extending the lifetime of NAND flash memory by salvaging bad blocks. In *15th Design, Automation, and Test in Europe (DATE 2012) conference*, pages 260–263, Dresden, Germany. March 2012.

3. Chundong Wang and Weng-Fai Wong. ADAPT: Efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies*, MSST '12, Pacific Grove, California, USA, April 2012.

4. Chundong Wang and Weng-Fai Wong. TreeFTL: Efficient RAM Management for High Performance of NAND Flash-based Storage Systems. In *Proceedings of the 16th Design, Automation and Test in Europe Conference*, DATE '13, pages 374-379, Grenoble, France. March 2013.

5. Chundong Wang and Weng-Fai Wong. SAW: System-assisted wear leveling on the write endurance of NAND flash devices. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 164:1-164:9, Austin, Texas, USA, 2013. ACM.

List of Tables

List of Figures

Chapter 1

Introduction

The advent of flash memory has changed the persistent data storage of computer systems. NAND flash memory's non-volatility, lightweight, shock-resistance and scalability make it a promising candidate for the secondary storage in both embedded systems and general-purpose computing systems. However, the ever-increasing utilization of NAND flash memory comes with its challenges. On the one hand, the environments in which NAND flash memory is used today vary significantly. For example, the access pattern of a smart-phone is very different from that of an enterprise server. On the other hand, NAND flash memory has been evolving to be denser and weaker than before. Also, the products made of NAND flash memory are getting diverse; they can be either emulated to be block devices or just exposed as raw flash devices. In all, these challenges necessitate revising existent strategies for managing NAND flash-based device. This thesis will hence present novel approaches on the management of NAND flash memory. Several management algorithms, which target either longer device lifetime or higher access performance, have been developed accordingly in order to achieve satisfactory effectiveness and efficiency.

1.1 Flash Memory Management

1.1.1 NAND Flash Memory

NAND flash memory is preferred in hand-held products like smart-phones, digital cameras and tablet computers, because of its lightweight and resistance to damage during movements [50]. Simultaneously, flash-based solid state drives (SSDs) are starting to replace traditional ferro-magnetic hard disk drives (HDDs) [1, 78]. Both personal computers (PCs) and enterprise servers have been utilizing flash-based SSDs for secondary storage. For example, the MacBook Air laptops

of Apple inc. are mature in marketplace. In 2008 Google announced a plan to use Intel SSD storage in its servers [20]. Later in the autumn of 2009, MySpace migrated its data from HDDs to SSDs produced by Fusion-io [72].

A NAND flash device consists of multiple flash memory chips. In a NAND flash chip there are hundreds of thousands of flash cells. Each flash cell has a single transistor with an extra metal strip, which is called the floating gate between the control gate and the oxide tunnel [5, 27, 89, 7]. To store data into a cell has to program it, which means to place a very high voltage to drive electrons to approach the floating gate. However, electrons will stay there unless a reverse voltage is applied to pull them off the floating gate. Such a process is referred to as an erase operation. Note that an erase operation takes a much longer time than a program operation. So it is unacceptable to update "in place" as the time caused by an additional erase operation is too costly. Herein lies the first key issue of NAND flash memory, which is, data have to be updated in an *out-of-place* way: data to be updated are first written into a clean page and the original page of the data is invalidated to be dirty. Another issue is the units of the said program and erase operations for flash memory. Because of the fabrication, the unit for a program operation of NAND flash memory is a page, and the unit for an erase operation is a block. Generally a page consists of thousands of flash cells, and a block comprises scores of pages. The out-of-place updating and the access unit constraints are the main concerns for the improvement of access performance of NAND flash memory-based devices.

The third issue of NAND flash memory also comes from the flash cell structure. The oxide layer of the cell, the one that isolates the electrons of the floating gate, is alternatively strained by continual program and erase operations for storing data [82]. In a long run, the oxide layer would be punctured after too many P/E cycles [79]. Then the cell cannot store data any longer. A page that has a permanently defective cell is deemed to be "worn-out". It in turn makes the block it is in worn out. A worn-out bad block is supposed to be kept away from regular use [39, 67]. Worse, a flash chip that has excessive worn-out blocks has to be discarded. Such an issue is referenced as the *write endurance* of NAND flash memory. It adversely impacts the lifetime of NAND flash devices.

1.1.2 Flash Memory Management

The characteristics of NAND flash memory, including access unit constraints, out-of-place updating and write endurance, are the foundation of all strategies for flash memory management. There are three goals for the flash memory management. First, the utilization of flash blocks and pages should be as high

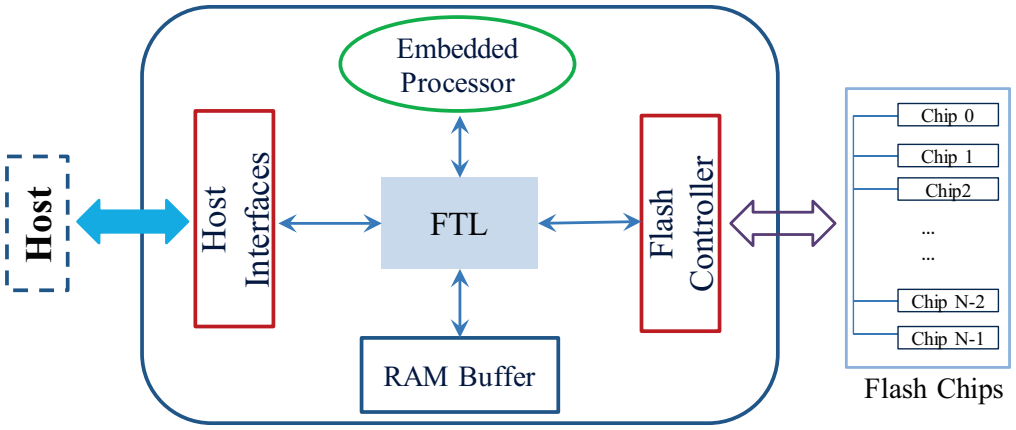

Figure 1.1: A Logical Structure of NAND Flash Devices

as possible. Second, the performance of data access must be optimal. Third, the lifetime of flash device has to be entailed without too much performance degradation [44, 95].

Figure 1.1 shows the logical structure of a common NAND flash device. It has an interface like USB or SATA that connects to the host system. Inside the flash device an embedded processor is equipped for computation. The RAM cache, also referenced as RAM buffer in some literatures, is used to buffer data and metadata. The flash controller conducts write, read and erase operations on flash chips. The FTL, which is abbreviated for the *flash translation layer*, is the embedded firmware that is responsible for the management of a flash device.

The functionalities of flash memory management include address mapping, wear leveling, bad block management (BBM), RAM buffer management and garbage collection, as is sketched in Figure 1.2. Address mapping is also known as address translation. We will use them interchangeably in this thesis. Address mapping is to map logical addresses given by the host file system to physical addresses in the form of flash block and page. Owing to the constraints of access units as well as out-of-place updating, address mapping of flash device is not that straightforward. Wear leveling is a technique targeting the issue write endurance of flash memory to avoid premature retirement of flash blocks. It aims to even out erase operations across all flash blocks. So it is used to ensure that flash blocks are worn at the same rate. Though, blocks may still go worn-out, and BBM is employed to trace them. RAM buffer is an important component of NAND flash devices. SRAM or DRAM has much shorter latency than NAND flash memory, and to utilize a RAM cache for buffering may favorably affect access performance of NAND flash devices. Garbage collection, also known as the reclamation, is caused by out-of-place updating that leaves invalid, obsolete

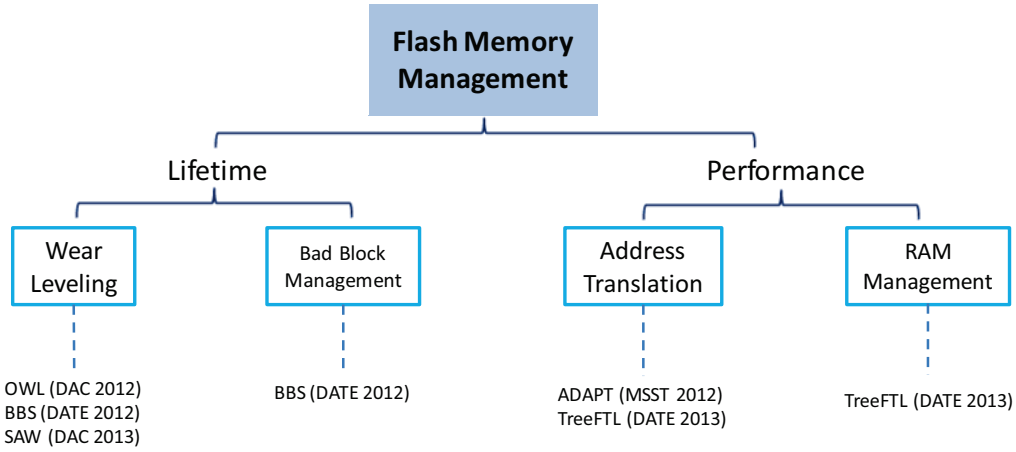

Figure 1.2: The Flash Memory Management

data behind. Such dirty data have to be demolished. The blocks and pages they take up can be vacated and cleaned for further use.

All the above functionalities of flash management are performed by one entity, i.e., the mentioned FTL. The FTL may be presented or named in different ways [68]. Here we reference them uniformly as the FTL for the ease of discussion. The FTL is designed in a modular way; each module of the FTL works on one functionality of flash management. Though, how to develop a module deserves special attention as it is not trivial to hold both the effectiveness and the efficiency simultaneously in hand.

1.2 Problem Formulation and Motivation

The ever-increasing utilization of NAND flash memory indicates the bright future of flash devices. As the dollar/capacity offered by flash-based storage devices is continuously decreasing, the utilization would be further boosted. However, the concomitant challenges are ignorable. The dropping of price for NAND flash memory is partially caused by the Multi-Level Cell (MLC) technique to produce flash memory. Briefly speaking, a traditional flash cell can store only one bit per cell, which is called Single-Level cell (SLC) flash. Using MLC technique, two [61] or more [54] bits now can be stored just within one single cell. Since flash memory can be manufactured to be much denser with MLC technique, the reduction of production cost is not beyond expectation. However, the reduction of price is not free of charge on other aspects. Empirical evidence of worsening lifetime and reliability, as well as access performance, of MLC flash memory has been reported [27]. Though, MLC flash is still considered to be the mainstream in

marketplace [28], and most low-end and middle-level SSDs are made of MLC flash chips [15]. The two-fold MLC flash and its prevalence dictate that the embedded software to manage a flash device, i.e., the said FTL, should be fittingly designed to provide satisfying device lifetime and access performance.

Besides the issue of the development of NAND flash memory, which is derived from the innate characteristics of flash itself, the situations where flash device is being environed turn to be a concern also. Different workloads differently impose on the storage device. As access performance and write endurance of flash device are strongly correlated to the workload in service, to be adaptive to workload is widely advocated by researchers and practitioners [1, 15, 17, 45, 64, 78, 111]. A common way to speculate the access behavior of a workload is to assess the ratio of sequential to random requests. Sequential requests are ones that access a large number of pages. Random requests selectively access a handful of pages among a wide range. Flash-based device is believed to be favored by workloads with a high demand for random access requests [78] as flash memory need not rotate the actuator to locate the desired position like ferromagnetic hard disk. Nevertheless, random writes in a large storage space may lead to excessively long response latency, owing to write amplification caused by inevitable garbage collection as well as wear leveling [15, 33]. Worse, because of out-of-place updating, the various workloads of access requests result in various layouts of data across flash blocks. This may not be a big deal for hard disk, or byte-addressable SRAM and DRAM as they support in-place rewriting; for NAND flash memory, however, to recycle used space badly impacts access performance and device lifetime. Therefore, it is desirable for a flash device to have a good understanding of workloads for serving them.

In all, both the flash memory itself and its utilization motivate us to rethink of **how** to manage flash device. On the one hand, the management of flash device must highly regard the specifics of NAND flash memory. The aforementioned address mapping, for example, is not merely to map addresses; to allocate flash pages and blocks is one of its duties. The allocation of blocks and pages must abide by access constraints and erase-before-program issue of NAND flash memory. As for wear leveling, it is just employed to target the issue of write endurance of flash.

On the other hand, the management of flash device ought to be self-adaptive to various workloads. Existing strategies of previous works, however, have limitations on the adaptation. For example, FAST [60] is a classical FTL that was proposed for mapping addresses. It judiciously utilizes the access units of flash memory as well as out-of-place updating in managing blocks and pages to

accommodate data, but it lacks on the ability of handling sequential requests. The successor of FAST, the FASTer FTL [64], emphasizes on workloads found in OLTP systems. But OLTP system just represents one type of workloads.

The third perspective on flash device is to view it in a systemic way. Flash device is used for secondary storage in a computer system. It is not irrelevant to other components of the integration. Two implications lie herein. Firstly, flash device serves the upper-level OS to store and access data; in other words, it conducts communications with OS. So it is able to obtain substantial information from the OS for the purpose of managing flash device. TRIM command [21], which engages the modern OS in informing flash device of reclaiming space in advance, shows the feasibility of notification from the OS to the FTL. Although the TRIM command is simple, more complicated exchange is implied to be possible. Secondly, the management of flash device can be enhanced using the ideas reflected in other parts of the computer system. For example, the mentioned FAST FTL uses the idea of CPU cache for address mapping. The page management of virtual memory [97], as well as the virtual RAM drive constructed by a part of main memory [2], shares similar points with flash device as well. However, as flash itself differs from DRAM-based main memory, they cannot be directly applied to flash device. Though, their ideas are still referential to us.

1.3 Thesis Statement and Overview

Given the challenges described above, the aim of this thesis, is to propose novel strategies for flash management which, on the one hand must take into consideration the idiosyncratic characteristics of NAND flash memory, and, on the other hand should be effective and efficient for a variety of workloads. With these in mind, we have taken three approaches to the problem. Since the FTL is the main agent in charge of managing a flash device, it is natural to start by exploring the internals of the FTL. Thus in the first approach we proposed new modes of the cooperation between modules of the FTL. A module is responsible for one functionality and it has its particular perspective with regards to flash management. The cooperation we proposed is not simply exchanging of messages in between. Rather it is the co-development of modules; a part of one module is embedded into another so as to gain immediate information on the nature of the ongoing accesses. By doing so it is expected that one module can benefit from the sharing with another one.

As flash device needs to be able to handle various workloads, our second attempt is on the workload adaptation of FTL modules. In other words, we intend

to construct workload-adaptive modules. As a workload is nothing more than a series of consecutive access requests, the access behavior of a running workload can be learnt accordingly. The learning in turn helps the FTL handle future requests. In the end the management algorithm is able to adapt to different workloads.

The third approach we have explored is on the collaboration between the OS that sits in the upper level and the FTL that is in the lower-level storage device. The OS has good knowledge of applications, files and data, which is not available to the FTL. On the other side, the FTL autonomously manages the flash device in a manner that is transparent to the OS. So we involve the OS in the process of flash management. With the assistance of the OS, the FTL should profit from this involvement.

The main contributions of this thesis, also main ideas of this thesis, are as follows.

- Inter-module cooperation-based management for flash device is investigated. An algorithm for wear leveling, namely *Observational Wear Leveling* (OWL) [105] is proposed. The wear leveling module of OWL is co-developed within the address mapping module. By doing so, OWL can succinctly classify data and accommodate them accordingly.

- Schemes for workload-adaptive address mapping and RAM buffer management have been proposed. ADAPT [103] is for address mapping and it is able to serve workloads that have variant mixes of sequential and random requests. TreeFTL [107], which manages the RAM buffer of flash device, can dynamically adapt to workloads as it has a self-adjustive structure maintained in the buffer.

- OS-assisted flash management has been studied. An algorithm named *OS-Assisted Wear leveling* (SAW) [106] was devised. The wear leveling of SAW relies on the OS's hints. The OS is responsible for the analysis over a massive number of files with a model, and the FTL performs wear leveling as it is notified. According to the idea of SAW, a prototype has been established upon open-source systems.

The effectiveness as well as efficiency of these approaches have been verified to be evident and significant by our experiments. We believe that our proposals are positive contributions to the field of flash memory management. We also hope that our explorations will help practitioners improve existing designs. Besides the widespread presence of flash device in mobile systems like smartphones,

netbooks and tablet computers, it is also clear that flash memory will play an important role in the next generation of secondary storage for general-purpose computing systems. To summarize, we believe our proposals to be described in following chapters of this thesis will improve the utilization of flash-based storage devices in the near future.

1.4 Organization of the Chapters

In this thesis, the three said approaches with several novel schemes would be described. This chapter has introduced an overview of NAND flash memory, flash-based device and the motivation for novel flash management strategies. Chapter 2 will give a detailed background of NAND flash memory. Chapter 3 surveys flash device and state-of-the-art schemes that were proposed for flash memory management. They are for different functionalities and the essence of their designs would be discussed. Chapter 4 is what we did to verify the effect of the module-cooperative approach. It presents the Observational Wear Leveling (OWL). For OWL, the module of address mapping assists the module of wear leveling to allocate flash blocks to data. In other words, address mapping classifies data and wear leveling accommodates them subsequently. Through cooperation the wear evenness is significantly improved with ignorable performance overheads. Chapter 5 and Chapter 6 are our attempts to develop workload-adaptive modules for flash management. Chapter 5 presents ADAPT. As mentioned, ADAPT is able to be adaptive to workloads that are variously mixed with random accesses and sequential accesses. Chapter 6 proposes an algorithm named TreeFTL [107] for RAM buffer management. TreeFTL is succinctly sensitive to running workloads. It adapts to workloads by dynamically partitioning the RAM space for buffering data and mapping addresses. The performance improvement has been reported through the employment of TreeFTL and ADAPT, respectively. Chapter 7 is about the OS-Assisted Wear leveling (SAW). For SAW, the OS is not unaware of flash memory management any longer. Instead, the FTL conducts wear leveling with hints provided by the OS. The hints are generated online through a model over a large number of files. The wear evenness is consequently improved due to the participation of the OS. Chapter 8 will conclude this thesis and possible future works would be briefly presented.

Chapter 2

Background

This chapter gives an overview of NAND flash memory as well as tactics preferred for flash memory management. It first details physical characteristics of NAND flash memory, including issues about flash cells, out-of-place updating and write endurance. Following these are aspects of flash memory management, including the modules of wear leveling, address mapping, RAM buffer management and bad block management, etc.

2.1 NAND Flash Memory

NAND flash memory was invented by Masuoka et al. [71] of Toshiba. Its full name could be NAND flash Electrically Erasable Programmable Read-Only Memory (EEPROM or E^2PROM) [89]. All the characteristics of NAND flash memory, as well as the modules of flash management firmware, are based on the structure of a NAND flash cell.

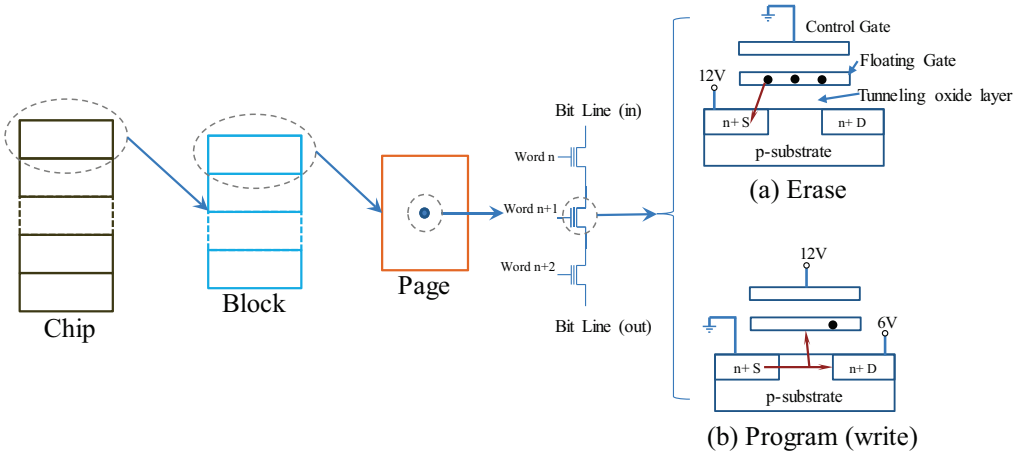

Figure 2.1: Structures and Operations of NAND Flash Memory

Flash Cell, Page and Block Figure 2.1 shows a sketch of the structure of a flash cell, with erase and program operations alongside. A flash cell is a transistor with an extra floating gate. Flash memory makes use of charge stored on the floating gate to accomplish the non-volatile storage [7]. The floating gate is a metal strip between the control gate and the tunnelling oxide layer of the transistor. It is sandwiched with oxide insulators, which enables the cell to retain charge for a long period of time even if the circuit power supply is cut off. To program or erase a flash cell is just to drive electrons. When the erase operation is conducted, under the voltage the electrons at the floating gate will be ejected to the source by tunnelling. The cell after an erase operation is in the '1' state. To program a cell to be '0' state, a reversed voltage must be applied to the control gate, and then electrons are driven to approach the floating gate.

SLC flash and MLC flash There are two types of NAND flash memory. One is single-level cell (SLC) flash memory of which each cell stores one bit. On the other hand, a cell of multi-level cell (MLC) flash is able to store two bits or more. Note that for SLC flash memory whether the bit is '1' or '0' is decided through sensing the voltage. The range of the voltage is divided into two halves with a threshold. If the voltage sensed is higher than the threshold, it is deemed to be '1'. Otherwise it is '0'. For MLC flash, more thresholds are inserted to set up more divisions over the voltage range. For example, if the range of the voltage is divided into four quarters, the cell can represent '00', '01', '10' and '11'; commonly two bits are stored in an MLC flash cell [26]. Products that have three bits stored in a cell are available in marketplace today. However, the increase of density is at the cost of the worsening endurance for a flash cell.

Out-of-place updating To do in-place updating is not reasonable for NAND flash memory. It is due to the physical characteristics of the flash cell. As is mentioned, electrons are trapped until an erase operation is conducted to pull them away. Considering the access units of NAND flash memory, to update data requires that a page should be rewritten. A flash page cannot be individually erased unless the whole block it is in is erased. Put in another way, if we tried to do in-place updating on a single page, we would have to rewrite all pages in a block after an erase operation. In this way the overhead caused by a write operation would be too significant due to many writes plus one erase operation. Out-of-place updating is yet acceptable. Every time data in a page are to be updated, an erased page will be allocated to accommodate them; the original page will be invalidated then.

Write endurance The issue of write endurance is another problem of NAND flash memory, which is also ascribed to the physical characteristics of flash

cells. It is obvious that both program operation and erase operation alternatively strain the oxide layer of a cell through applying voltages to drive electrons. After undergoing too many program/erase (P/E) flips (the reversals of voltage), finally the oxide layer cannot isolate the floating gate any longer. The limitation for MLC NAND flash memory is much tighter than SLC flash. For the former, it is about 10,000 cycles for a page; for the latter, it is about 100,000 cycles. As is said, the range of the voltage for NAND flash memory is divided into more parts. To program the bits for writing requires much more elaborate techniques. The finer adjustment adversely impacts the physical tolerance of the flash cell. This explains why MLC flash devices have a much shorter lifetime. For SLC flash devices, though it has a longer lifespan, the upper bound of P/E cycles is still not so satisfying for use.

2.2 Modules of Flash Memory Management

The said flash translation layer (FTL) is the one that is responsible for the management of flash device. It can be found in flash-based block devices, such as SSDs or USB sticks. In an MTD device made of raw flash [98], it is presented in another form. As their functions are identical, we will reference them uniformly as the FTLs for the ease of discussion.

The FTL emulates flash devices like traditional block-interface devices to hide special characteristics of NAND flash memory. Main functionalities of flash management, including wear leveling, address translation, bad block management, RAM buffer management and garbage collection, are represented by respective modules of the FTL. We will first give an overview of wear leveling and address translation, as they are two basic modules for flash memory management.

Wear Leveling Wear leveling targets the issue of write endurance of flash memory. As is mentioned, limited program/erase flips exist for a flash page. However, previous algorithms of wear leveling mostly focus on erase operations as the physical limitation is mainly caused during the erasing procedure [89]. On the other hand, to reduce program/erase flips at the page-level is not reasonable as the unit of erase operations is a block. Besides, the coarser granularity of erasures can ease the module of wear leveling. Hence, it is preferred for wear leveling to spread erase operations over flash blocks.

Wear leveling's common tactic is to classify data and put them into suitable aged blocks. To do so a data structure called the *block aging table* (BAT) is needed [40]. It is used to record the age of each block. The age here refers to the erase count of a flash block. The more the erase count, the older the flash

block. As for data, they would be identified to be either hot or cold. Hot data are ones that are frequently updated. Otherwise, they are cold data. This is an inaccurate and rough classification on data. Because cold data are seldom or never rewritten after storage, they are preferred to being put into elder blocks. In this way elder block can avoid being erased soon. On the other hand, given a younger block that is used to accommodate hot data, as the data are likely to be invalidated soon, it would be erased soon for reclamation. Therefore, the wear evenness over flash blocks is gradually achieved.

Traditionally algorithms of wear leveling are classified into two categories. It can be either *dynamic* or *static* [10, 40]. Dynamic wear leveling generally selects the youngest free block for new data. Static wear leveling may vacate the block currently occupied by cold data for use. The latter is more prevalent today because all blocks are under consideration. Another perspective to classify wear leveling schemes is on how the module of wear leveling is triggered: an algorithm can be deemed to be proactive, passive or hybrid [105]. Proactive wear leveling aims to put data in suitable aged blocks actively. Upon allocation requests, the access frequency of the data has been estimated, and a block would be found and allocated accordingly. The overhead to do estimation is inevitable. Passive wear leveling swaps data between blocks when the wear evenness over blocks has been worsened beyond a certain limit. Hence, the evenness has to be continually detected at runtime. Hybrid wear leveling has both features.

Address Mapping Address mapping, also known as address translation, is to translate logical addresses given by file systems to physical addresses in the form of flash block and page [103, 118, 107]. Page mapping and block mapping are two basic mapping schemes. Figure 2.2 sketches them.

Given a logical address, the FTL looks up in the mapping table to find the corresponding physical block number in the case of block mapping, or physical block number and page number in the case of page mapping. Page mapping is flexible to relocate data among pages. However, the overhead due to the fine granularity cannot be ignored. Specifically the size of the mapping table is troublesome. For a 64GB SSD with 2KB per page, there would be more than 32 million entries in the table. If 4 bytes are used for an entry, the table will be 128MB. It is difficult to maintain such a large table in RAM buffer for reference.

On the other side, block mapping works at the block-level. It has a much smaller mapping table, but it lacks flexibility owing to its coarse granularity. For a logical page, it can only reside within the same physical page of different blocks under block mapping. Therefore, to rewrite a page will cause block-level copying because data in neighbouring pages have to be migrated to next physical block

alongside. It is arduous to move so many data at one time for one single rewrite.

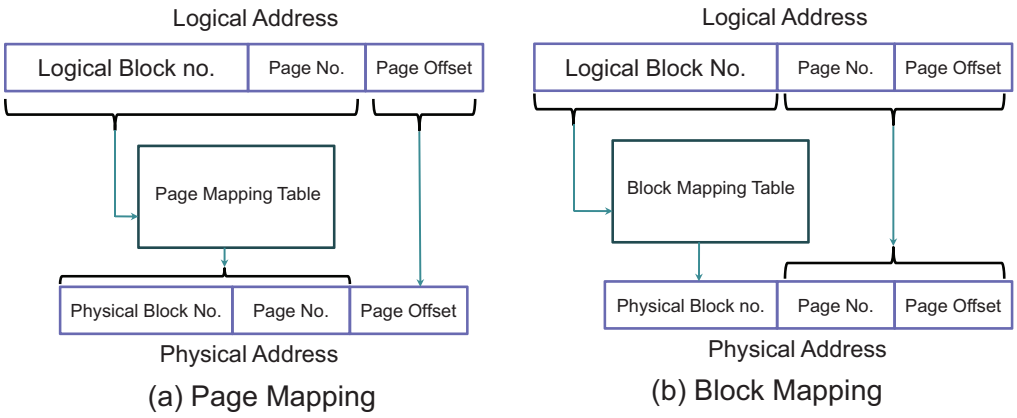

Figure 2.2: Page Mapping and Block Mapping

Hybrid mapping combines page mapping and block mapping. It separates all physical blocks into the *data space*, *log space* and *free block pool*. Each logical block is mapped to a block in data space using block-level mapping. As block mapping is not flexible, the log space is maintained to temporarily hold updates in page mapping. Updates are first absorbed by log pages. They will be merged to data space afterwards. Details of hybrid mapping will be shown in Chapter 3.

Bad Block Management (BBM) BBM can be viewed as an extension of wear leveling. It is used to trace bad blocks that contain permanently defective cells. Note that some bad blocks are already present when the flash device is shipped [39]; they are referred to as *initial* bad blocks. In the beginning, initial bad blocks are marked and recorded in a *Bad Block Table* (BBT) [37] by manufacturers. The worn-out block is another type of bad blocks that come out at runtime. A flash cell is likely to go defective after it undergoes excessive P/E cycles. If a cell wears out, the page it is in, as well as the block, will be identified to be *worn-out*. Worn-out bad blocks are recorded in the BBT also. In tradition, bad blocks are supposed to be kept away from regular use.

RAM Buffer Management RAM buffer is an important resource of NAND flash devices. The RAM buffer is made of SRAM [29, 86], DRAM [43, 49, 94, 99] or non-volatile RAM [47, 66, 83]. Although flash memory can be accessed at a much higher speed than magnetic hard disks, the gap between the requirement of host system and the performance of flash device is still wide. Moreover, considering the said out-of-place updating, a buffer to cache updated data is very necessary for a flash-based device.

Besides the metadata related to flash management, entries of the address mapping table and data pages are also cached in the RAM space. In this way,

RAM buffer management serves the module of address mapping. Previously, the RAM space is used for one purpose, either address mapping or data buffering. Recently how to manage the RAM space for both uses has been explored.

Garbage Collection Garbage collection, also known as the reclamation [25], is usually designed within the wear leveling and/or address mapping. It is due to the out-of-place updating during address remapping. Invalid dirty data may be scattered across blocks after a period of execution [12, 15, 65]. When there are no flash blocks left for use, ones that have invalid data will be reclaimed. Yet valid data might exist in the block also. Therefore, for a victim block, the module of garbage collection needs to bypass invalid data, and move valid data to another clean block [33]. Then the victim block can be erased for future use. Besides affecting resource utilization, the scheduling of reclamation may have an impact on the access performance too.

2.3 The Background of the Era

The strategies to manage flash device were simple when they were primarily utilized. The capacity of a flash device was in a small magnitude three decades ago. To assure wear evenness or conduct garbage collection in a 128MB flash drive at that time is much easier than a 1TB SSD today. The situations in which flash memory was equipped were not complicated also. It was mainly used in USB drives or digital cameras. Access behaviors observed in these portable computing systems are usually discontinuous and bulky. Such simple access patterns are not difficult for the FTL to handle.

Things have changed a lot in the past thirty years. The presence of smartphones and tablet computers, as well as the upgrade of enterprise servers, requires that the secondary storage should be supported by a lightweight, shock-resistant and energy-efficient material. Undoubtedly NAND flash is a promising candidate. Thanks to the development of manufacturing and techniques like MLC, the flash device now can be produced in a huge capacity at a lower price. However, the ever-increasing utilization and expansion make the flash device confront unprecedented obstacles. The challenges met by flash devices that are exposed to various workloads are real and tough. How to manage flash device effectively without loss of efficiency in different systems deserves thorough investigation; otherwise the further use of flash device will be hindered. Researchers and practitioners are pondering, as solutions to the above problems are about to enhance the utilization of flash devices. On this ground, next chapters will show our proposals to mange flash device for both effectiveness and efficiency.

Chapter 3

Literature Review

Before the descriptions of our approaches, we will first present flash device and its past and potential. Then an overview of existing designs about flash memory management would be shown. Related works will be categorized according to the aspects of flash memory management, including previous schemes for wear leveling, address mapping and RAM buffer management. The strategies relevant to the design of management modules would be discussed also.

3.1 Flash Device and Its Potential

The evolution of flash memory entails it to be a promising candidate for the secondary storage of computer systems. The presence of flash device, however, is not unique. Generally speaking, there are two types of flash device. One is the raw flash device, which can be seen everywhere today as it is used in smartphones. The raw flash device directly exposes the physical characteristics of flash memory to the system, and the MTD hardware driver [98] helps the system write and read data. Flash memory management, though, is performed either by flash file systems or extra software layers. Note that file systems like Ext4 or NTFS cannot work immediately on raw flash devices. Flash file systems are ones that have been developed specifically for raw flash, including JFFS2 [112], YAFFS and YAFFS2 [70], as well as UBIFS [36]. These file systems cooperate with MTD drivers for data storage and access. They differ from Ext4 or NTFS in that they take into consideration characteristics of flash like erase-before-program issue and write endurance. So besides functionalities of common file systems, they also integrate modules relevant to flash management. JFFS2, YAFFS and YAFFS2 manage the flash device by themselves. UBIFS has a specific software layer called the UBI [23]. UBI can be viewed as a customized FTL for UBIFS. UBI has modules for address mapping and wear leveling while the garbage collection

is performed by UBIFS.

Another form of flash memory is to encapsulate flash chips into a drive that has a block input/output interface such as SSDs, USB thumb drives and micro-SD cards. Here the *block* does not means a flash block; the former is a sequence of bytes with a fixed length, used for data access and transmission, and the latter is the unit of erase operation of flash memory. In this thesis we will use the *block-interface device* to stand for *block device* to distinguish. Factually a basic use of FTL is to hide specifics of flash and emulate a flash device to be a block-interface device. By doing so, the flash device is able to be compatible with existing systems.

With the assistance of the FTL, file systems like Ext4 or NTFS can access data from block-interface flash-based device. It is not necessary for file systems to care about flash management as JFFS2 and YAFFS2 do. The FTL will be responsible for all management functionalities instead. As SSDs are springing in marketplace, much attentions have been paid to its inroad into enterprise servers and personal computers. Agrawal et al. [1] investigated the design tradoffs for SSD performance. They revealed that the access performance and the device lifetime of SSDs are highly workload-sensitive. They also argued that the layout of data is critical to both load balancing and wear leveling.

Later Narayanan et al. [78] gave an analysis on whether it is worth migrating the secondary storage of enterprise servers from ferromagnetic hard disks to SSDs. Their emphasis is on the cost versus capacity of SSDs. They addressed that the price of SSDs has to be decreased much more in order to replace HDDs. At the same time, Chen et al. [15] did experiments on low-end, middle-level and high-end SSDs to get insightful understanding upon performance issues of SSDs. Through measurements they found that the management of flash device ought to be more efficient for workloads. Other investigations for data-intensive workloads with flash memory were conducted as well [8]. Grupp et al. [27] did empirical estimates over flash memory to predict the future of SSDs. Their results point out that the density gain due to MLC techniques adversely impacts both performance and reliability of flash memory, which implicitly highlights the importance of the management firmware.

Besides real measures performed to flash products, the simulation of flash device is also attractive. For example, nandsim is a useful tool to simulate a raw flash device. It has been included in the Linux kernel [76]. Agrawal et al. [1] extended the DiskSim simulator to simulate an idealized SSD. Kim et al. [53] proposed FlashSim simulator, which is trace-driven and object-oriented. FlashSim allows researchers to implement their own FTLs for evaluation.

3.2 Algorithms of Flash Management

In this section the classical algorithms on facades of flash memory management are presented. Fundamental and classical schemes would be presented in details while others are briefly described.

3.2.1 Schemes for Wear Leveling

Table 3.1 shows four algorithms that were recently proposed for wear leveling. They all fall into the category of static wear leveling, although how they perform wear leveling significantly varies. Among these algorithms, the dual-pool scheme [9], BET [14] and lazy wear leveling [10] are activated only when the level of wear unevenness reaches some thresholds. So they perform wear leveling in a passive way.

Table 3.1: A Summary of the Latest Wear Leveling Algorithms

Algorithm	Type	Block Organization	Address Mapping
Dual-pool [9]	Passive	Hot pool and cold pool: a block with valid data is in either pool, where blocks are prioritized upon their erase counts.	Not constrained
BET [14]	Passive	Block sets and BET: A set has one block or several consecutive blocks to correspond a bit in the *block erasing table* (BET).	Not constrained
Rejuvenator [74]	Proactive + Passive	Multiple block lists: blocks that have the same erase count are grouped in a list.	Page mapping + Hybrid mapping
Lazy wear leveling [10]	Passive	Common way: free block pool, valid block pool, etc.	Hybrid mapping

In dual-pool algorithm, hot data and cold data stay in the hot pool and the cold pool, respectively. When the difference on the erase count between the head of the hot pool and the rear of the cold pool exceeds a predefined threshold, the two blocks will swap their places. For each pool, it may also be adjusted by exchanging data between blocks to adapt to dynamic workloads.

The block erasing table, abbreviated as the BET, is a key structure of the algorithm developed by Chang et al. [14]. We shall use this acronym to reference their algorithm. For BET, blocks are first divided into sets, and a set may have one block or more. The BET consists of bits; each bit represents a block set. When a predefined interval begins, all bits in the BET are initialized to be '0'. If one block of a block set is erased within the interval, its associated bit in the BET will be set to '1'. The total number of erasures in the interval is recorded. If the count of erase operations over the number of erased blocks exceeds a predefined

threshold, BET will repeatedly pick un-erased blocks of the last interval, and perform data transfers, after which it will erase them until the wear skewness is smoothed out.

Jung et al. [44] proposed a group-based wear leveling algorithm which is similar to BET, as it records the summary information for a group of logically consecutive blocks. By doing so the memory footprints can be reduced. The main tactic of this group-based algorithm is on data swapping between flash blocks. It also considers the performance degradation due to inevitable wear leveling actions.

Lazy wear leveling [10] is a recently proposed scheme. It is performed in the merge procedure of hybrid mapping. As is mentioned in Chapter 2, the hybrid mapping maintains the block mapping between logical blocks and data blocks while the page mapping is used to temporarily hold updated data with log blocks. The merge is a procedure during which valid data of a victim log block are merged with valid data from corresponding data blocks into newly-allocated blocks. Prior to lazy wear leveling, a data block that is involved during merge, say D, will be immediately erased. In lazy wear leveling, however, if D's erase count is higher than the average by a threshold Δ which can be tuned online, besides erasing D, the FTL will find a data block with cold data, say C, transfer C's data to D, erase C, and return C as a free block for future use.

In summary, the dual-pool scheme responds to the widening gap between two blocks' erasure counts, the BET scheme is activated when the erasures are unevenly distributed beyond an extent, and lazy wear leveling works when the block to be reclaimed is much older than the average. These reasons explain why we deem them to be "passive".

Rejuvenator [74] has both proactive and passive mechanisms. It allocates hot or cold data to young or old blocks respectively in a proactive way. It records recent access frequencies of logical pages, and identifies the temperature of pages accordingly. It also groups blocks that have the same erase count in a list. A list is in the *lower numbered lists* if its erase count is smaller than a dynamic threshold; or it is in *higher numbered lists*. When new write requests arrive, based on the recorded access information, cold data are put into younger blocks of the lower numbered lists using page mapping, and hot data are placed in elder blocks of the higher numbered lists in hybrid mapping. Between the smallest and biggest erase counts is a window. If the number of free blocks in either partition drops below two thresholds (T_L and T_H) respectively, data will be moved out from the lowest list to upper lists, and the window is then adjusted. This is how Rejuvenator performs passive wear leveling.

Recently the reason of write endurance has been investigated in terms of bit error rate of flash cells, and algorithms have been designed accordingly [79, 117]. For the ERA algorithm proposed by Yang et al. [117], the metric to spread erase operations inside a flash device is imposed on error rates of blocks. Yet the spreading is based on data migration between flash blocks. Besides, analytic models for wear leveling of flash memory [96] were also constructed; they are referential to designers.

3.2.2 Schemes for Address Mapping

Address mapping should be the most fundamental function of the FTL. Without it the flash device could not be usable at all. Page mapping [3] and block mapping [4] were devised based on the access units of page and block respectively. They are primary and simple. For an early flash device with a small capacity, they are sufficiently effective. However, with the advent of flash devices at a large capacity, the algorithms of [3] and [4] are not satisfying any longer as page mapping suffers from the large spatial overheads of address table while block mapping is inflexible at updating data [113, 103]. On this ground hybrid mapping that combines page mapping and block mapping was proposed.

The first attempt of hybrid mapping is BAST [52]. Its successor FAST [60] introduced more flexible associativity. FAST was in turn succeeded by FASTer [64] that exploited temporal locality for further performance improvement.

It is mentioned that hybrid mapping maintains data blocks using block-level mapping as well as a fixed number of log blocks in page-level mapping. Updates are first put into a log page instead of allocating a new data block. Hence, the log space formed by log blocks acts like a cache of processors [31] to data blocks. In BAST, there is a fixed one-one mapping between data blocks and log blocks. This inflexibility results in a poor utilization of log space. FAST, on the other hand, adopts a fully associative mapping between log space and all data blocks: a log block is no longer designated to one data block but shared by all. Thus, in terms of cache associativity, BAST maintains a direct mapped cache and FAST is fully associative. More complicated N-way associative schemes of log blocks have also been devised. Physical blocks are grouped together, and they are associated to a set of log blocks; the size of the set may be dynamically changed at runtime [80, 55]. Mapping schemes, like the superblock [46], LAST [62], KAST [18] and WAFTL [111], are also in the category of hybrid mapping but emphasize on garbage collection, multitasking and real-time systems, respectively. Besides, RNFTL [109] improves the utilization of flash blocks through reusing clean pages in blocks to be merged.

Mapping schemes that are conducted on other granularities have been proposed also [113, 63]. Generally they are derived from the above three categories. One is a set-based mapping strategy [19]. Each set contains multiple blocks. Logical sets are mapped to physical sets with another table used to store the mapping of logical block to physical block in a set. Lately another scheme is based on the concept of working set [116]. Additionally, Janus-FTL [56], as its name suggests, attempts to strike a balance between page mapping and block mapping at runtime.

Typically, the log space of hybrid mapping is over-provisioned to be 3% of all space [59, 64]. It is usually partitioned into a sequential area for sequential writes and a random area for random writes. FAST assigns one log block as its sequential area while LAST maintains a fixed number of blocks. They also have methods to identify whether a request is sequential or random.

It is natural to process access requests for hybrid mapping. When a write request arrives, the FTL first checks whether the page in the mapped data block is clean. If not, a log page will be allocated to accommodate the data. The old copy will be invalidated. The relationship between the logical page and the log page is recorded in the log page mapping table. When no clean page is left in the log space, a victim log block will be picked out and *merged* with corresponding data blocks. After merging, the victim is erased and returned to the free block pool. Another clean block will be allocated to replenish the log space. Figure 3.1 is adopted from [62]. In Figure 3.1 a square is a page and a rectangle of four squares represents a flash block. The number in each square is the logical page number that it maps to. Data in a shaded page are invalid. In Figure 3.1(c), logical page 2 is mapped to data block D_2 but cannot be rewritten directly. A page in log block L_3 has to be allocated. Successive updates can be handled by more log pages, and mapping entries are changed accordingly. In Figure 3.1(c), three log pages in L_2 and L_3 are used for logical page 4. If all pages of log blocks are exhausted, a merge procedure must be performed to make space.

Figure 3.1 shows three types of merge in FAST. *Switch merge* and *partial merge* have lower overheads, and are expected in the sequential area. For a switch merge (shown in Figure 3.1(a)), the log block contains contiguous valid data from the same logical block. It can therefore be simply switched to data space. In a partial merge, the log block will also replace its relevant data block but some valid data in current data block have to be transferred to it first, as shown in Figure 3.1(b). *Full merge* is more complicated. FAST is fully associative and each log block is shared by all data blocks. Thus, a full merge is costly because each page with valid data in the log block must be (potentially)

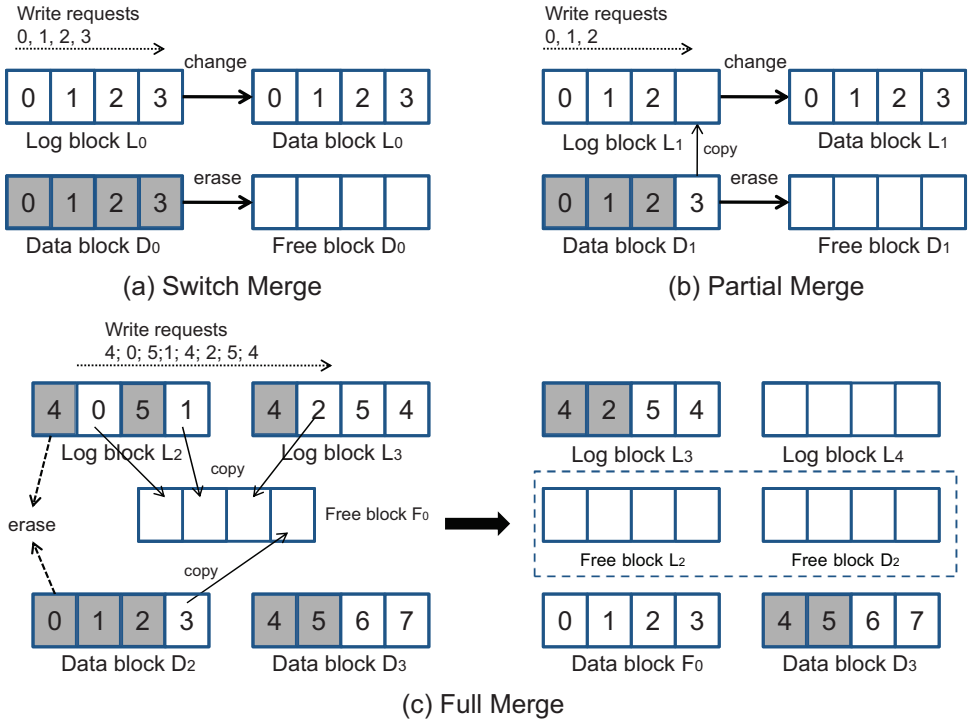

Figure 3.1: Three types of merge(adopted from Lee et al. [62])

merged with a different data block. This requires many writes and erasures. FAST and FASTer organize the random area in a FIFO queue (that they called "round-robin"), and the victim log block for the full merge would be the one at the head of the random area.

Recently, content-aware FTLs that attempt to reduce duplicate writes have been proposed too. Examples include CAFTL [17] and CA-SSD [30]. ΔFTL [114] also considers content locality; if a similar copy comes for an existing data segment, only the difference will be stored by ΔFTL. In all, they can potentially benefit from the content detection and reduction.

3.2.3 Schemes for RAM Buffer Management

To manage the RAM buffer is an important responsibility of FTLs. Metadata and data that are under request both pass through the RAM buffer, so the RAM buffer is the most suitable one to reflect access behaviors of workloads. FTLs use RAM space to hold mapping entries. DFTL [29] loads entries from translation pages on demand. Besides single entries, CDFTL [86] selectively caches translation pages in a two-level structure, as is shown in Figure 3.2. Mapping entries form the first level, the *cached mapping table* (CMT). Evicted entries from the

CMT are first absorbed by cached translation pages in the second level. The second-level exploits the spatial locality in workloads since neighbouring logical addresses in a same translation page are likely to be accessed. DAC [85] is similar to CDFTL on caching mapping entries but the former works at block-level for large-scale flash storage systems.

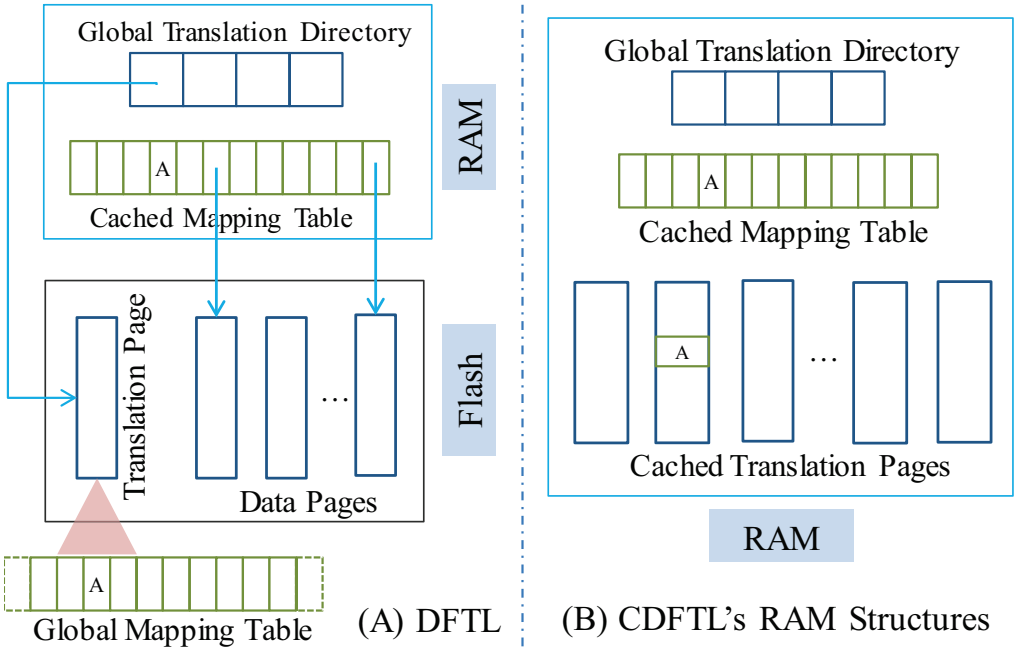

Figure 3.2: Page-level Mapping: DFTL and CDFTL

Data buffering, especially for write requests, is another use of RAM space. A flash page is the buffering unit due to NAND flash memory's access constraints. BPLRU [51] utilizes a *padding* strategy within hybrid mapping. Unlike RAM buffer management that only writes data to flash memory upon evictions to free up space, BPLRU may read data from flash memory to pad a log block and flush all data of a block back. Padding is expected to avoid arduous merge procedures. However, reading data pages also costs time. A scheme named *l-buffer* [13] has been proposed to trade off padding for merging, and vice versa. Beside locally caching inside an individual device, buffering data for multiple flash devices have also been investigated. FlashCoop [110] is exemplary to show how to make use of remote RAM buffer of SSDs that are from neighbouring servers for data buffering.

APS [94] and JTL [35] are two recent proposals that use the RAM cache for mapping and buffering jointly in a flash device. APS reserves two small areas of RAM as "ghost caches". One is maintained to keep metadata of evicted mapping entries, while the other maintains the metadata of evicted data pages.

They are used to compute the expense caused by not enlarging the cache for mapping and buffering, respectively. Write or read misses in actual cache may hit in ghost cache. A cost-benefit model is built on these hit statistics to estimate the benefits of enlarging either partition. Because APS's estimation is based on values of the past interval, there are delays in adjusting to runtime workload. Moreover, APS uses the least recently used (LRU) algorithm at page-level or entry-level to find a victim for evictions in respective partition. The overhead of frequent LRU selections can be significant since tens of thousands of data pages and mapping entries exist in the RAM.

JTL statically partitions the RAM space into two halves, one for mapping, and the other for buffering. JTL uses a multi-level structure to manage mapping entries. For the level n ($n \geq 0$), it has 2^n entries. The number of levels is determined by the size of the RAM partition dedicated to mapping, and the size of a single entry. All levels are divided into two groups. As RAM cache is halved for buffering data pages, the mapping entries for these buffered pages form Group 0 and take up positions from level 0 to m. Remaining levels fall into Group 1, and their entries correspond to data pages that are still stored in flash. The entry in the top level corresponds to the most recently used data page. The entry of the newly-accessed page will drive the current entry in the top level to move down. One entry at level 1 may need to move to level 2 if no vacancy exists. More moves may follow in next levels. The victim to be moved in each level is randomly selected as entries in the same level is deemed to have similar access recency. When an entry reaches level $m + 1$, its cached data page in RAM will be flushed to flash memory. By doing so, JTL can keep the recently used mapping entries and data pages cached in RAM.

3.3 Strategies Behind Flash Management

3.3.1 Module-Cooperative Flash Management

Module cooperation is based on the hypothesis that modules can help each other within flash memory management. At the beginning of utilizing flash devices, the cooperation between modules were not necessary. Three decades have passed, and unprecedented obstacles come out to hinder the further use of flash-based storage devices. The module-cooperative approach turns to be the first feasible and possible way to seek for improvements.

There have been some schemes proposed to make one module cooperate with another one. Let us take BPLRU [51] and l-buffer [13] that are for RAM buffer management for example. They both involve hybrid address mapping in. It

is mentioned that data of a logical block under hybrid mapping are scattered in log pages and a data block. When the RAM space is used up, instead of flushing cached pages of a logical block, BPLRU reads some pages from the same logical block from flash memory and pads them to form a block, which entails a sequential write operation to flash device. By doing so BPLRU aims to avoid the expensive merge [51]. l-buffer extends the padding of BPLRU by balancing padding and merging. Hence, either BPLRU or l-buffer just services the writing of hybrid address translation. The interaction between them is not very meaningful.

The cooperation between address translation and wear leveling also exists. Lazy wear leveling [10] mentioned above is a good example. It works within hybrid mapping. To be more specific, it checks the victim block during a merge, and decides whether to utilize it or find another instead. The cooperation is also straightforward, as no interplay is introduced into either side.

Factually, the cooperation between modules can be more meaningful and significative. In this thesis, Chapter 4 will present the effectiveness of wear leveling resulted from the deep cooperation between modules of address mapping and wear leveling for flash memory management.

3.3.2 Workload-adaptive Flash Management

The requirement of being workload-adaptive is due to the variety of access patterns of workloads flash memory is serving [11]. WAFTL [111] was claimed to be workload-adaptive. It is for address mapping. It combines the said two basic mapping schemes, but differs from hybrid mapping in its management on flash blocks. It has a page-mapping *buffer zone* like the log space to hold updates, and data blocks are partitioned into Block-level Mapping Blocks (BMB) and Page-level Mapping Blocks (PMB). When the buffer zone is full, a *data migration* procedure will be called to transfer the data out. WAFTL adapts to workloads by sending buffered data to either BMB or PMB upon their access frequencies: highly accessed data will be sent to PMB and others will be put in BMB. Unlike merging a log block, data migration will flush all data in buffer zone and completely reconstruct the space. It is costly to move so many data at a time.

There are also proposals on RAM buffer management that attempts to be adaptive to workloads. The adaptation is achieved through the adjustment of the partitions for address mapping and data buffering, though the way to adjust partitioning is not simple. APS [47] maintains ghost caches for two partitions to emulate the misses and hits in every interval in order to set the future partitioning. However, such complicated mechanism and the feedback way make it

heavyweight to adapt to online workloads, not to mention the delay to respond to the context switch of workloads.

In this thesis, two intelligent schemes based on the workload adaptation will be shown. Their tactics are easily implemented and the effects are yet evident.

3.3.3 OS-involved Flash Management

The FTL for flash memory management is traditionally designed to be self-contained [6]. The host OS communicates with the flash device through interfaces like USB or SATA, and is generally oblivious of the management of flash memory. The OS sends requests to the FTL, and waits for replies in a client-server manner, treating the flash device as a black box.

The involvement of the OS into flash management is attractive. There are schemes that were devised to take file systems into account. MFTL [115] interposes a filter between the file system and the FTL to separate metadata and real data of files. Metadata are essential information to manage data of files, like the filename, access time and access type. Generally metadata are small and frequently udpated. MFTL pays special attention to them. It was implemented within ext2 and ext3 file systems, and performance improvement was reported. FSAF [75] focuses only on deleted data in FAT32. It is similar to the TRIM command of modern OS [21, 42]. FSAF detects the deletion by utilizing its knowledge about the format of FAT32 in storage devices. Meta-Cure [108] is similar to MFTL. It adds a filter between file system and FTL to enhance the reliability of "critical data" to avoid being damaged. Critical data in [108] are ones that are vital for the file system and flash management. The loss of critical data may bring in disastrous damage to the storage system. Though, Meta-Cure does not change the file system; it is transparent to the FTL. Neither is Hystor [16] which manages both SSDs and HDDs as one single block-interface entity and avoids undesirable significant changes to existing file systems.

In all these works, either the OS is unaware of FTL's workings, or vice-versa. The FTLs just focus on either data to be deleted, metadata or cirtical data. Our proposal in this thesis, however, is completely different, as it is a collaborative model. The OS itself participates in the process of management. The flash management is expected to exploit the OS's knowledge of data and files for profits. More details can be found in Chapter 7.

Chapter 4

OWL: Cooperative Wear Leveling

This chapter will present the algorithm we developed in the first step of this thesis to explore the inter-module cooperation inside the FTL. Its name is *Observational Wear Leveling*, abbreviated as OWL. It has cooperation between address mapping and wear leveling. The cooperation here is not simply exchanging messages between modules. Instead, a sub-module of wear leveling is inset into the hybrid mapping module, so that the latter is able to provide the immediate information to the former for wear leveling. Specifically speaking, OWL allocates suitable aged flash blocks to data during the process of address mapping. Block allocation requests are raised in different scenarios for hybrid mapping; OWL handles them case by case. In order to facilitate the module of wear leveling, the way to organize blocks is also customized. Through the organic deep cooperation between wear leveling and hybrid address mapping, the wear evenness is significantly improved, which hence confirms our hypothesis on the potential gains obtained from the module cooperation. The mechanism of OWL, as well as the experimental evaluation, will be detailed in this chapter.

4.1 Overview

As is mentioned in Chapter 2, wear leveling and address translation are two basic functionalities of flash memory management. That is one reason why we seek their cooperation. Wear leveling is employed to spread erase operations as evenly as possible to ensure the lifetime of NAND flash device. From the analysis of the latest algorithms on wear leveling in Chapter 3, we can see most of them are induced when the wear evenness has been worsened to some extent. So we

deem them to be *block-centric* wear leveling algorithms, or *passive* wear leveling. Being block-centric means that their emphasis is on the flash blocks, while being passive means that they are activated by the worsening wear evenness. Moreover, previous wear leveling schemes ignore a key point of wear leveling. Let us first raise a question, "what causes the wear unevenness among flash blocks?" The answer is the data. Evidently different data have different access frequencies. They are accommodated into flash blocks. As a result, updating data differently impacts the wear status of blocks. Take, for example, an extreme case, assuming that all data were read-only. After the first write they would remain unchanged, and no skewness would appear among blocks. This example is unusual. However, it directs us to a new way to perform wear leveling. That is, how to devise a *data-centric* algorithm for wear leveling. The proposal in the chapter, the said OWL, has been developed in this data-centric way. OWL can also be viewed to be *proactive*, as it estimates and accommodates data into suitable aged blocks to avoid unnecessary arduous data movements.

The essence of OWL is to assess data's access likelihood and put them in suitable aged blocks, which is founded on the co-development of the wear leveling module and the address mapping module. One reason why we explore the cooperation between them has been given above: they are basic modules for flash memory management. Another reason is that it is during the process of mapping logical address to physical address that flash blocks are allocated to data. Moreover, there are reasons why **hybrid address mapping** is particularly opted. First, hybrid address mapping has been widely utilized for flash devices. The prevalence of hybrid mapping makes it a good candidate for investigation. Second, hybrid address mapping provides good perspectives on the spot of block allocation; the scenarios to allocate blocks are inherently classified, which facilitates the wear leveling module to estimate and rank update frequencies for data. Third, hybrid mapping has a shortcut to separate hot data and cold data, because the log space interposes as a filter to sift different data.

Let us give an overview of OWL. It is said that OWL attempts to evenly spread erase operations during the process of hybrid address mapping. To do so, OWL maintains a ***B**lock **A**ccess **T**able* (BAT) that observes and records the history of recent logical block accesses. The BAT is used to perform ***L**ocality-based **B**lock **A**llocation* (LBA) in the merge process of hybrid mapping. OWL also identifies cold or very hot data, and transfers them if necessary to prevent young blocks from being occupied for too long time. The cold or very hot data emerge as hybrid address mapping leaves them behind. The blocks they take up are consequently identified with the cooperation of hybrid mapping. The

sub-module of OWL, namely the scan and transfer (ST) scheme, will find them and vacate young blocks they take up for future use. From our experiments, with minimal spatial and temporal overheads, OWL can improve wear evenness by as much as 29.9% and 43.2% compared to two state-of-the-art wear leveling algorithms, respectively.

4.2 Challenge and Motivation

The emphasis of all wear leveling algorithms is laid on how to efficiently manage a number of flash blocks with different ages in order to make them wear at the same low rate. The key idea of OWL also lies in the management of blocks. The particularity is that OWL attempts to take advantage of a given address mapping scheme; herein it is the hybrid address mapping. The first concern is what the module of wear leveling must customize to suit for the cooperation with address mapping. One issue is the customization of organizing blocks. As OWL intends to allocate flash blocks case by case, an efficient block organization is necessary. As a matter of fact, not only wear leveling but all functionalities of flash memory management are affected by the way to organize flash blocks.

Assuming an efficient block organization is available, how to allocate blocks must be seriously considered as during processing block allocation data and flash block meet. Data have different access frequencies while flash blocks have different wear records. The strategy to allocate blocks surely impacts the future wear evenness. Traditionally there are two straight manners to conduct block allocations: the first-in-first-out (FIFO) and the youngest block first [9]. One FTL roughly picks either way to allocate flash blocks. A good opportunity to perform wear leveling, however, has been missed by previous schemes as the wear skewness starts from the allocation of different aged blocks to different data.

An efficient customized organization of flash blocks for OWL would be detailed. On the other hand, which block to be allocated depends on the scenario where the allocation request is raised. For hybrid mapping there are three scenarios to allocate blocks. First, allocations are necessitated for data that newly arrive. Such data are stored for the first time. Second, data to be merged have to be accommodated. Third, the victim log block has been reclaimed after merging, and the log space must be replenished by one clean flash block. The latter two cases are from the procedure of merge. Table 4.1 shows the proportions of allocations for newly-arrived data and data to be merged under the FAST FTL [60]. Without loss of generality, traces from [84], [101] and [77] were used for analysis. It is obvious that the ratio for data to be merged is much higher

Table 4.1: Block Allocation Ratios in FAST

Trace	New Allocation	Merge Allocation
SPC1	3.90%	96.10%
TPC-C	33.76%	67.24%
MSR-hm_0	4.87%	95.13%
MSR-mds_0	13.02%	86.98%
MSR-prn_0	16.07%	83.93%
MSR-prxy_0	7.07%	92.93%
MSR-rsrch_0	18.42%	81.58%
MSR-stg_0	7.30%	92.70%
MSR-ts_0	8.29%	91.71%
MSR-web_0	6.75%	93.25%

than that for newly-arrived data, so OWL targets the allocation requests raised for the former.

Note that the log space is used to hold the updated copies of data. It is maintained in an over-provisioned, fixed capacity [1, 60]. Therefore, some of the data in the log blocks may have to be evicted by merging to free up space. However, the eviction does not mean that such data would never be updated any longer. In terms of temporal locality, some of them may be accessed soon while others may become cold. At this moment if we can predict which data are likely to go cold, and allocate elder blocks to them, then future movements for cold data will be avoided. Moreover, allocating young blocks to data that are still hot also enhances wear evenness.

4.3 OWL's Block Organization

As is mentioned, an efficient block organization promisingly facilitates the block allocation and other functions of flash management. Even so, the block organization has not attracted as much attention as it deserves. The ways to organize flash blocks of existing FTLs were mostly simple and straightforward. For example, DFTL maintains a *free block pool* of clean blocks for address translation [29]. For hybrid address mapping flash blocks are just divided into three groups, and not too many particular rules have been given except that log blocks are preferably managed in a FIFO queue [60, 64].

As wear leveling needs to locate the block it asks for as soon as possible, it usually has its own special manner on organizing flash blocks. Previous works have been shown in the Table 3.1 in Chapter 3. Take Dual-pool algorithm [9]

for instance. Its name suggests that it relies on two pools of blocks to perform wear leveling. The recently-proposed Rejuvenator [74] maintains multiple lists of blocks that have the same erase count for grouping.

As for the said OWL in this chapter, all flash blocks, excluding log blocks that are managed in a FIFO queue as usual, are grouped in two pools, which are the *free block pool* and the *valid block pool*. This is a common organization in FTL designs [29]. In OWL we modify it slightly. The free block pool is sorted according to the erase count of each block. Its data structure can be organized like a min-heap for implementation, or other complicated ones that may consume less RAM space [9]. However, one issue of the min-heap is that it has no strict order for its nodes, which violates our intention. But the pool of flash blocks is different from a small-scale, unique-key set of nodes. Since many flash blocks are likely to share the same erase count, they can be approximately put into neighboring levels of the heap structure. A strict order is not necessary so as to minimize the temporal and spatial overheads caused by the maintenance of the structure. In all, what we manage for OWL is a general sorted multiple-level structure that keeps flash blocks in an inexact sequence. Using such a min-heap-like structure, if the number of blocks is n, it will take $O(log(n))$ to enqueue an erased block into the pool. Besides the free block pool, blocks in the valid block pool are ordered by their arrival time. It is almost like an ordinary FIFO queue, except that a valid block in the middle of the pool may, at an appropriate time, be moved to the head. The valid block pool groups all data blocks. It can be managed in a linear structure, for which the cost of insertion and removal is $O(1)$ and $O(n)$, respectively. Note that the structures are temporarily stored in byte-addressable RAM cache [52, 66]; even though one single insertion or removal is insignificant, the cost still needs to be minimized considering cumulative impacts in a long run. This also advocates that data structures for block organization and other management issues deserve careful consideration as they would be continuously used.

4.4 Locality-based Block Allocation

As pointed out earlier, block allocation requests may be issued for new log blocks, arrivals of new data, or data to be merged. Traditional wear leveling algorithms mostly employ one policy, either FIFO or the youngest block first [9, 104]. In OWL, they are not uniformly handled as usual. In particular, allocations for log block and new data are done using the youngest block first policy. This

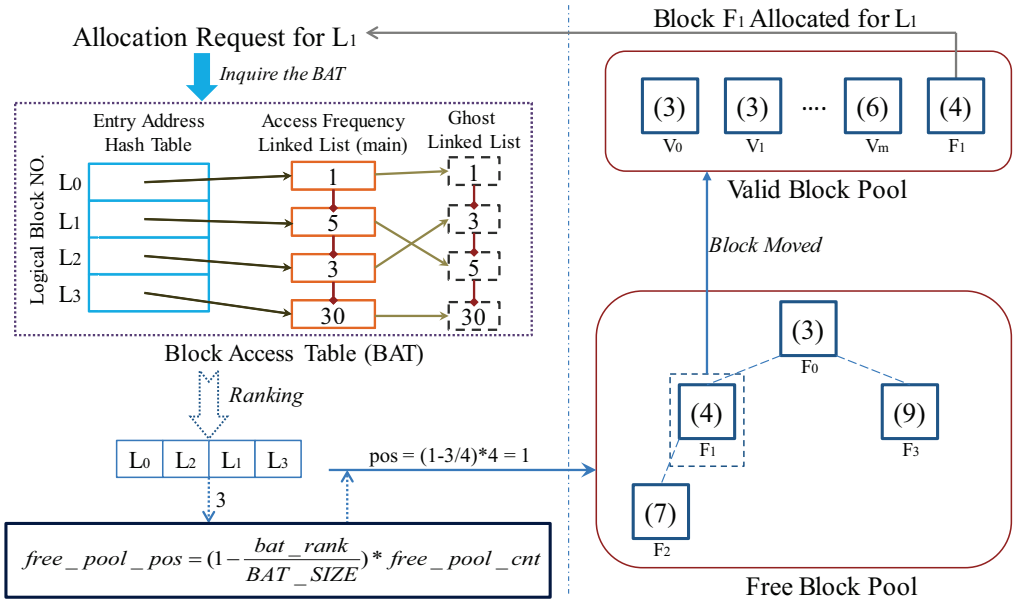

Figure 4.1: Locality-based Block Allocation with BAT

is easy to implement using OWL's free block pool organization as it is just a matter of fetching the top of the min-heap structure. Requests from merge, however, are serviced by the allocation of a suitable aged block that is selected in a predictive way according to the data's recent write history.

The recent history of writes to logical blocks is recorded in the BAT in the form of write frequencies. Hence, the BAT is a runtime record of the temporal locality of writes. The BAT comprises three components: a hash table for rapid looking-up, and the main linked list to hold logical blocks' access frequencies, and a ghost linked list. A sketch of the BAT is shown in Figure 4.1. The hash table maps a logical block number to a linked list entry. In the main linked list, an entry can be quickly appended or moved to the end of the list (being the most recently used). On the arrival of a write request with a logical address, the hash table will be checked. If the logical block number does not exist, an entry will be created and appended to the end of the main linked list, and the hash table mapping is set up accordingly. Otherwise, the relevant entry will be updated and moved to the end. If the RAM space allocated for the main linked list is exhausted, the least-recently-used (LRU) entry, i.e., the one in the front, will be deleted to make space for the new arrival. Hence, the BAT keeps the latest information of the temporal locality of recent writes. It will be used by the FTL for the servicing of block allocation requests. The ghost linked list and the overheads to maintain the BAT will be given in next subsections.

Here we will present the LBA algorithm used in the merge procedure. As

mentioned, during a merge, free blocks are needed to accept data from the log page selected as the eviction victim, and its related data blocks. These data were not recently used. However, the situation may change in the near future. LBA aims to put the data to be merged into blocks of suitable ages in a predictive way. In particular, LBA tries to make younger blocks hold hot data, while using elder ones for the cold.

Algorithm 1: Locality-based Block Allocation

```
   Input  : logical_blk_no, logical block number in request
   Output: free_blk_no, allocated free block number
 1 begin
 2     bat_rank := CalcBATRank (logical_blk_no);
 3     free_pool_pos := (1 − bat_rank / BAT_SIZE) ∗ free_pool_cnt;
 4     blk_pt := GetFreePoolHead (void);
 5     cnt := 0;
 6     while (cnt < free_pool_pos) do
 7         cnt + +;
 8         blk_pt := GetNextFreeBlk (blk_pt);
 9     end
10     free_blk_no := blk_pt;
11     return free_blk_no;
12 end
```

Algorithm 1 presents the skeleton of LBA. It is called in the merge procedure with the logical block number as a parameter and returns the block number of a free physical block. At line 2, the FTL first calculates the "rank" of the logical block in the BAT. In brief, the rank of a logical block is the count of blocks in the BAT that have lower access frequencies than it. At line 3, the FTL computes a position in the free block pool using a heuristic formula. From line 4 to line 11, LBA will find the block at that position in the free block pool, and return it to the merge procedure.

The idea behind Algorithm 1 is as follows. First, the rank of the given logical block is calculated using the recent write history recorded in the BAT. If the logical block is highly accessed, its access frequency will be higher than many others. Then its rank in the BAT will be high too. The LBA puts this rank in the formula at line 3 to get the position in the free block pool, and looks for a free block accordingly. The free block pool is a min-heap sorted with the blocks' erase counts. Hence, LBA can easily locate the one with the suitable age in $O(log(n))$ time.

Computing the rank of a logical block is not straightforward. Since the BAT stores the frequencies of recently referenced blocks, an intuitive way to rank a

logical block L is $\frac{BAT[L].freq}{\sum_{l \in BAT} BAT[l].freq}$. However, this is incorrect. In the most recent interval, some blocks may be highly accessed. These hot blocks will have very high frequencies, and they can dominate the total sum. The above fraction will show a bias towards these blocks, and the ranks of other blocks will be inaccurate. Worse, physical blocks cannot be fairly utilized because hot data are unlikely to be merged soon but always occupy younger blocks. In OWL, we first sort the blocks according to their frequencies. The rank is obtained after sorting. Our experiments show that this is a better measure.

Figure 4.1 gives an example of LBA scheme. There are 4 entries in the BAT, and 4 blocks in the free block pool. The number in the brackets of each block is its erase count. When a request is raised for logical block L_1, the FTL will examine the BAT, and perform sorting. The rank of L_1 is 3, and according to the formula, its position in the free block pool is calculated to be 1. With this number, the FTL finds physical block F_1, and moves it to the valid block pool. Finally, the FTL will return the block number F_1.

A possible issue of the LBA might be ranking logical blocks through sorting their recent write frequencies. As a sort procedure may take a long time, the overhead is unacceptable. Based on the BAT, we have devised a lightweight method. It is achieved with the aid of a ghost linked list whose nodes also stand for access frequencies of logical blocks. But the ghost linked list does not store frequencies by itself but utilizes those of the main linked list. So it can be viewed as a shadow of the main linked list. A sample of two linked lists are shown in Figure 4.1. We do not perform sorting upon each allocation request. Instead we relocate the node in the ghost linked list after each write. Upon a write operation to a logical block, its frequency in the main linked list will be updated. When the increment to the block is completed, the block's write frequency has consequently changed. As the frequency definitely increases, we will push the node in the ghost linked list forward to approach the rear of the ghost linked list. In this process we compare the frequency of the moving node to the frequency of each node we are about to bypass. If the frequency of the moving node is no more than the one to be bypassed, the pushing process will terminate. The moving node will be inserted here, as the current position is just fit for it. Initially there are no nodes in the ghost linked list as well. An incoming one would be enqueued to be the head. Note that for the main linked list the incoming one is appended to the rear to represent the MRU item. Each time we need to calculate the degree, we will just find the position number counting from the head of the ghost linked list. In this way the ponderous sortings are avoided, as is shown in Figure 4.1.

The temporal and spatial overheads of the BAT are fairly small. It is maintained in the RAM space with the block and log page mapping tables. The access latency is much smaller than that of flash. It is not necessary to store the BAT in flash because temporal locality is always changing. The spatial overhead is also low. For each logical block, 4 bytes are used for the entry of the hash table, and another 4 bytes record the access frequency in the main linked list. To maintain the ghost linked list, 4 extra bytes are allocated for each logical block. So the BAT will ask for 3KB for 256 logical blocks to support the LBA scheme of OWL. Experiments show that a 3KB BAT is sufficient for OWL to function.

From Table 3.1 we can see Rejunvator also has proactive way to do wear leveling. OWL differs from Rejuvenator in three important ways. Firstly, Rejuvenator focuses on the block allocation upon the arrival of new write requests; OWL works in the merge procedure that issues much more allocation requests (as shown in Table 4.1). Secondly, Rejuvenator uses page mapping for hot data and hybrid mapping for cold data. This is fairly complicated. OWL utilizes hybrid mapping only, and hence eases the design. Thirdly, while they both utilize a structure to record reference counts of logical addresses at runtime, Rejuvenator maintains access information at the granularity of pages. OWL's BAT works at the block-level. With the same amount of RAM space, OWL can store longer historical accesses.

4.5 Scan and Transfer Scheme

LBA works in the merge process, and it may miss two types of data. One is data that are seldom, or possibly never updated after being stored. They have no up-to-date copies in log space. In other words, their data blocks are not related to any log page. Another is ones that are very hot. If data are highly updated, their old copies in log pages will be quickly invalidated. So they can avoid being merged. Evidently blocks occupied by these data are unlikely to be erased. Thus, we use a proactive scheme named *scan and transfer* (ST) to find these data, and efficiently place them in elder blocks.

Many methods have been proposed for hot/cold data identification [74, 104]. Note that here valid blocks are chronologically appended to the valid block pool, and blocks at the head have been there for the longest time. ST exploits the organization of the valid block pool, and periodically scans a small portion through the pool to find a block containing one of the above two types of data. To do so, ST employs two variables, λ and δ. Briefly, ST scans ($\delta \cdot 100\%$) of the valid block pool after every λ write requests.

In its scanning, ST identities a young block with cold data using the block's erase count and mapping status. In our implementation, we deem a block to be "young" if its erase count is smaller than half of the average erase count of all blocks, which is more strict than group-based wear leveling [44] and lazy wear leveling [10] that set such a standard to be the average erase count. If a young block is not associated to any log pages, it will be picked. After the scanning, more than one candidate may be found. To minimize the performance overhead, ST will transfer one block's data each time. Let functions $T(b)$ and $Q(b)$ represent block b's residence time in the pool and the quantity of valid pages to be transferred, respectively. The victim should be the one that has stayed for the longest time, and has the least data. Let

$$v(b) = \frac{T(b)}{Q(b)}. \tag{4.1}$$

The block that has the largest $v(b)$ can be selected as the victim. Given the valid pool's organization, $T(b)$ can be replaced by $\frac{1}{P(b)}$ where $P(b)$ is block b's *position number* in the pool. For example, the head of the pool has a position number 1. Then Equation 4.1 will be

$$v(b) = \frac{1}{P(b) \cdot Q(b)}. \tag{4.2}$$

There are several issues to use Equation 4.2, however. Firstly, $P(b)$ can be easily obtained, but to maintain $Q(b)$ for each block requires a large amount of RAM space. Secondly, in Equation 4.2, $Q(b)$ has the same weight as $P(b)$. Since ST transfers one block after every λ requests, a larger $Q(b)$ is acceptable, but a block with a big $P(b)$ might be mistakenly identified as cold. Thirdly, computing $v(b)$ for all the candidates may consume too much time.

Based on Equation 4.2, ST can be done in a simplified yet efficient way. Besides λ and δ, ST employs a pointer `pt` and a counter k. Initially, `pt` points at the first block that is associated to log pages, and k is set to zero. ST will check each block's erase count and mapping status through scanning δ blocks of the pool. If a block satisfies the condition mentioned above, i.e. is young and not associated to any log page, it will be selected, and inserted before `pt`. After scanning, data of the first selected block will be transferred and k will count by one. Before next scan, if the block that `pt` points at is to be merged, `pt` will be replaced at the next block that is associated to some log pages, and k will be reset to zero. In the next scan, if blocks found in previous scans exist, ST cancels scanning and just performs data transfer on the first one of these

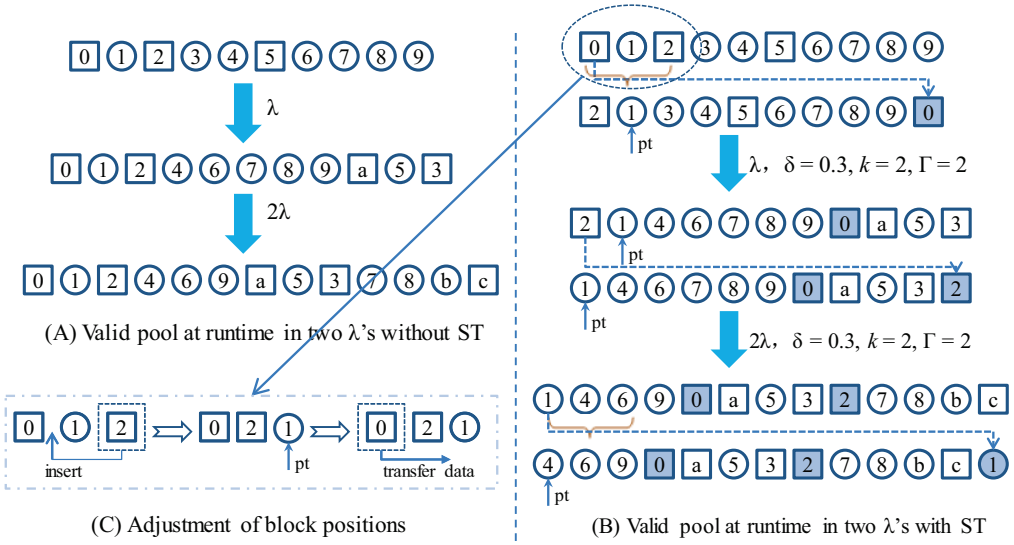

Figure 4.2: An Example of ST Scheme

blocks. If after scanning no candidate is found, ST will check k. If k is bigger than a threshold Γ, the block that `pt` points to has been there for at least $(\Gamma \cdot \lambda)$ requests, and avoided being merged. The data that block holds could be very hot. So ST will select and transfer it; `pt` and k will be reset accordingly. If $k < \Gamma$, ST just returns. Note that the ST scheme is similar to the Clock algorithm for approximate LRU page replacement of virtual memory [97]. One different point is that ST does not track how often a block is accessed by itself. Instead, it takes advantage of the information provided by the module of hybrid address mapping. Another important difference is that, Clock algorithm only picks out the one that is the least recently used while ST may also choose the one that is the most recently used (a flash block with very hot data).

Obviously ST prefers blocks of cold data to blocks of hot data because the latter still might be merged. It uses `pt` and k heuristically to identify a block with very hot data. Figure 4.2 shows an example of ST at runtime. Figure 4.2(A) is the pool's being in two λ requests without ST. Squares are blocks that are not associated to any log page, and circles are ones that are. The number inside is the logical block number mapped to each block. In Figure 4.2(B), ST transfers data in logical block 0, 2 and 1 to elder blocks. Figure 4.2(C) shows a case that a selected block is inserted before `pt`.

4.6 Experimental Evaluation

4.6.1 Experimental Methodology

There are three ways to do experiments in order to measure the effectiveness of wear leveling algorithms. They all aim to ensure that all blocks are covered in assessing wear evenness. The first way is what lazy wear leveling did [10]. They configured a 20.5GB SSD (0.5GB was over-provisioned for log space of hybrid mapping) in their simulator, and "replayed the input workload one hundred times". The second was used in the Rejuvenator paper [74]. Their SSD in simulation had 32GB, but they "restrict the active region" for write requests and "the remaining blocks did not participate in the I/O operations". The third way is what we did. For each workload, we assigned a reasonable capacity so that all blocks have the chance to be involved in wear leveling. The capacities for workloads we used are shown in Table 4.2. Note that the over-provisioning rate for log space is 3% which is the same as previous works [29].

Table 4.2: Capacities for Traces

Trace	Capacity
SPC1	2.06GB
TPC-C	3.09GB
MSR-hm_0	4.12GB
MSR-mds_0	2.06GB
MSR-prn_0	6.18GB
MSR-prxy_0	4.12GB
MSR-rsrch_0	2.06GB
MSR-stg_0	4.12GB
MSR-ts_0	2.06GB
MSR-web_0	2.06GB

All the experiments were conducted using the FlashSim simulator [53] in a Linux 64-bit system with GCC 4.6. The address mapping used was FAST [60] that has been modified with our block organizations. We implemented BET, lazy wear leveling and Rejuvenator as comparisons to OWL. In the following texts, `baseline` refers to a configuration that has no wear leveling, `lazy` is the one with lazy wear leveling, `OWL` refers to our proposed OWL algorithm, and `OWL-nc` has all of OWL except the ST module.

The traces we used came from three sources. They are shown in Table 4.1 and Table 4.2. SPC1 and SPC2 were downloaded from [84]. TPC-C is a typical online transaction processing (OLTP) workload from [101]. All others were from

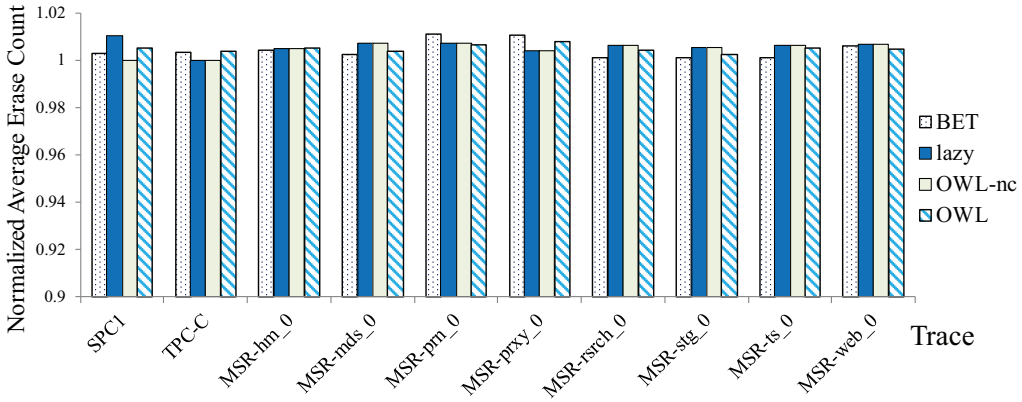

Figure 4.3: Average Erase Counts of Each Trace

Microsoft's data centers [77]. They represent various environments, and the numbers of write requests vary from 1 to 12 million. Note that each write request in the trace may consist of multiple write operations. *Caveat lector*: these traces were recorded at different machines whose configurations were never clearly documented. Therefore, we used a different configuration for each trace, as is shown in Table 4.2, in order to assess wear evenness caused by wear leveling algorithms.

We studied three metrics. The average erase count, and its standard deviation are used to measure the effectiveness of the wear leveling algorithms. The overhead is measured by the elapsed time needed to finish processing the trace. All three metrics have to be assessed together in order to obtain a qualitative judgement about the efficacy of the algorithms.

For `BET`, we configured each block set to be a single block. This is the best case for BET in terms of wear leveling. The threshold Δ of lazy wear leveling was initialized to be 2. It is adaptively tuned online according to [10]. For `OWL`, the default values of λ, δ and Γ are 1000 requests, 0.4% and 50. All flash parameters, like the latencies of write and erase operations, were obtained from [34].

4.6.2 Effectiveness of OWL

Figure 4.3-4.5 are results on average erase count, standard deviation, and elapsed time for each trace, normalized against `baseline`. Note that the optimal wear leveling algorithm could achieve the absolute wear evenness without different among flash blocks, which means that the standard deviation is zero. Figure 4.3 shows `OWL` can reduce the number of erasures in many cases, while Figure 4.4 shows that the standard deviation decreases, by as much as 29.9% and 43.2% compared to `BET` and `lazy` with MSR-prxy_0 respectively. These lead us to

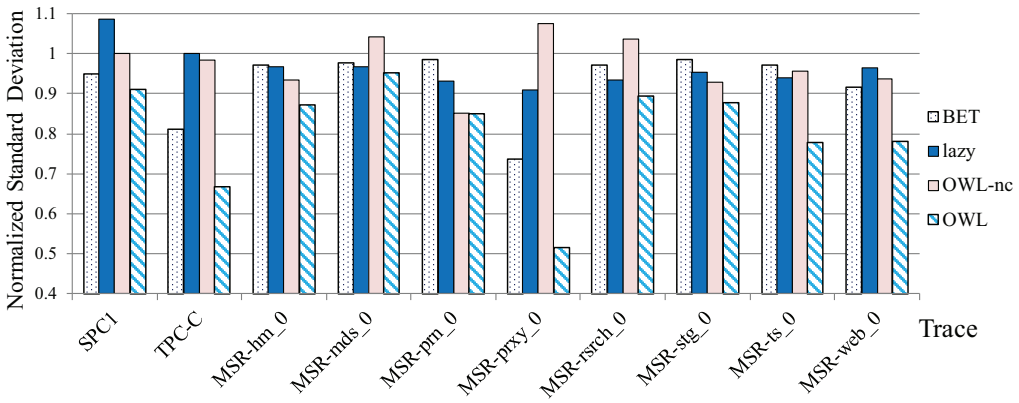

Figure 4.4: Standard Deviation of Erase Counts

conclude that `OWL` performs better than `BET` and `lazy` in evening out erasures. Figure 4.5 shows the elapsed time on processing each trace. `OWL` is at most 1.1% slower than the `baseline` in the case of MSR-prn_0.

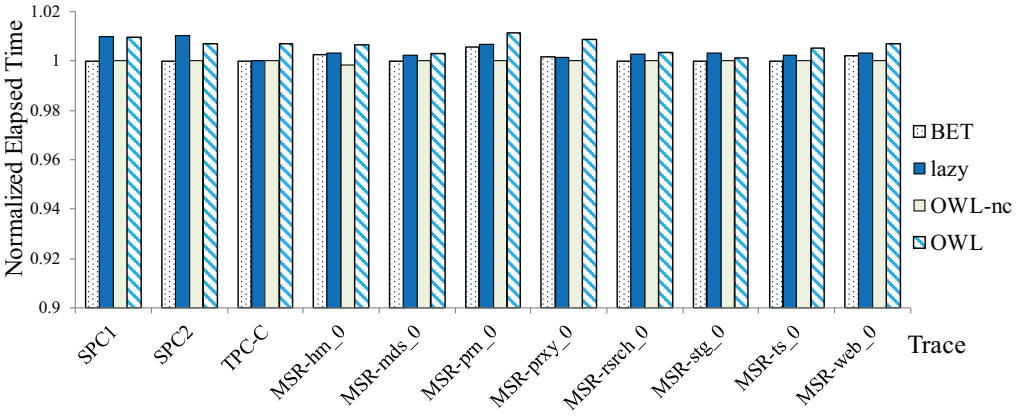

Figure 4.5: Elapsed Time with Four Algorithms

As mentioned earlier, the three metrics should be considered together. Take for example TPC-C. It has 7.7 million requests in the workload. From Figure 4.3, we can see `OWL` has a similar number of erasures as `BET` and `lazy`. However, as shown in Figure 4.4 the difference in standard deviation is significant. This implies `OWL` achieves better wear evenness with roughly the same erasures.

There are traces in which `OWL` did not do too well also. Figure 4.3 shows that `OWL` has slightly more erasures than `lazy` for MSR-prxy_0. We analyzed MSR-prxy_0, and found it quite different from other traces. Normally, one would expect a write request to access a number of pages. MSR-prxy_0, however, has a large number of very small write requests, with 77.8% of the requests accessing only one page. Since the BAT works at block-level, such a situation is difficult

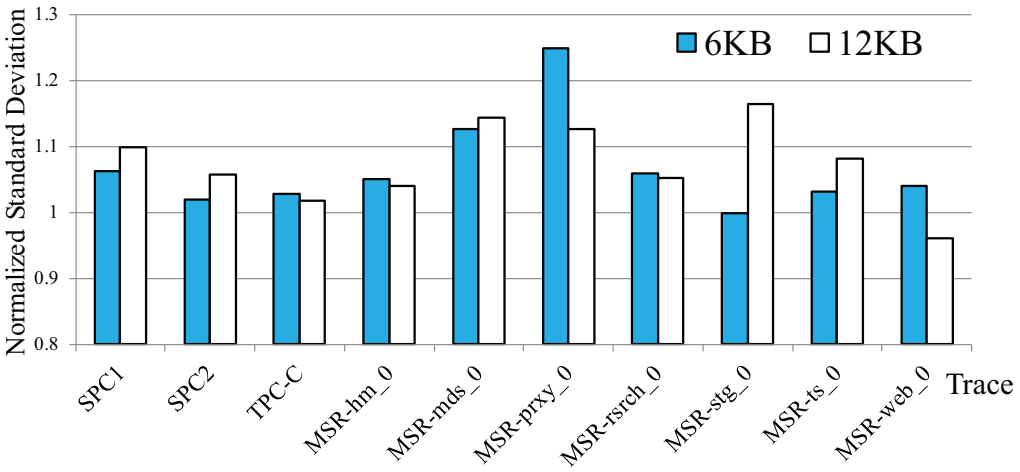

Figure 4.6: The Effects of Different BAT Size

for the BAT to record access information accurately. This in turn affected LBA's allocations. Even so, `OWL` was still able to use the ST module to perform wear leveling. This is why `OWL` has a little more erasures, but the best evenness.

We have also implemented Rejuvenator. However, there were several stumbling blocks. Specifically, two thresholds (T_L and T_H) were not given in their paper. Also, it was said initially all blocks will have a zero erase count, and all will be in the lower numbered lists. However, when and how to migrate from such initial state to the two partitions of the lower and higher numbered lists were not described in [74]. These parameters and process are important for Rejuvenator. Nonetheless, we tried to simulated it but the results are not comparable to those for `BET`, `lazy` and `OWL`. Take TPC-C trace for example. It should be easy to identify hot and cold data based on the access information of TPC-C workload. Our simulation of Rejuvenator has a similar erase count as `OWL` but its standard deviation over all blocks is 44.3% more than `OWL`. It is worse for other traces.

4.6.3 Effects of BAT Size

The BAT is used to support LBA in the merge procedure. The default size in our experiments is 3KB, allowing for 256 records. We also tried varying the size to 6KB and 12KB. The standard deviations of these normalized against the 3KB configuration are presented in Figure 4.6. From it we can see in general a larger BAT results in more unevenness. The BAT records the latest write frequencies, and one with a larger capacity is more likely to store outdated information. This will mislead LBA. In terms of overhead, besides saving space, a smaller BAT can also have a lower access time.

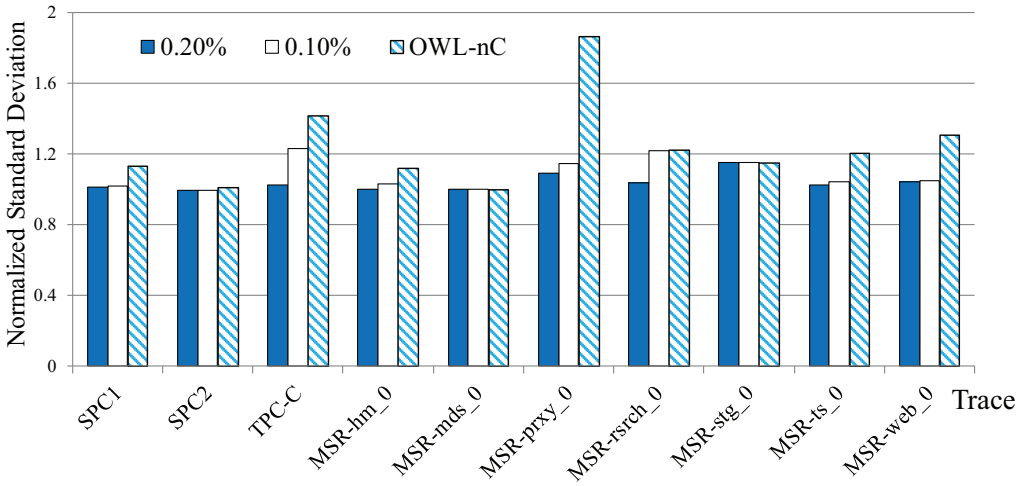

Figure 4.7: The Effects of ST with Various δ (A)

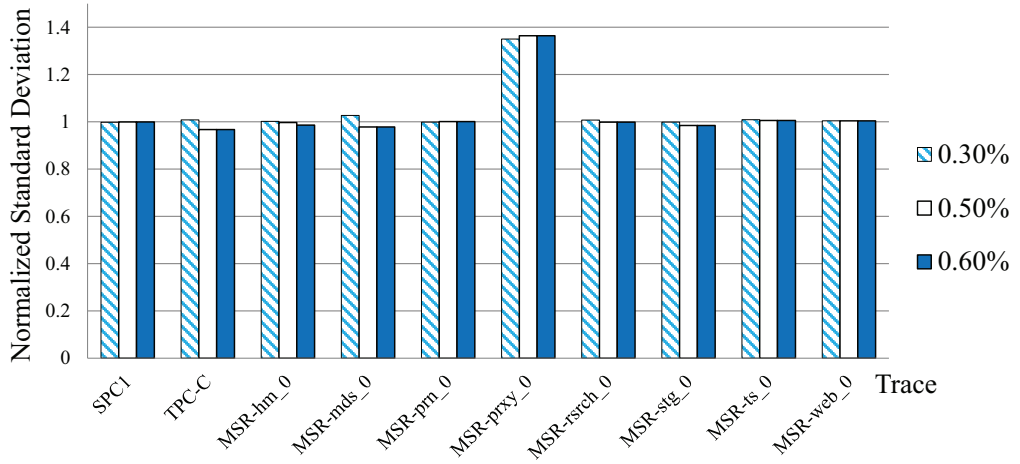

Figure 4.8: The Effects of ST with Various δ (B)

4.6.4 Effectiveness of ST

δ, λ and Γ are three parameters of ST module. We experimented with different values of them to study ST's functions. The results of various δ are shown in Figure 4.7 and Figure 4.8.

In our default setting, OWL will go through 0.4% of the valid block pool. We also experimented with δ being 0.1%, 0.2%, 0.3%, 0.5% and 0.6%, and normalized their results against those for 0.4%. Figure 4.7 shows that in general the wear evenness will worsen when a lower proportion of blocks is checked (TPC-C and MSR-rsrch_0). The worst case occurs in `OWL-nc` that does not have ST. From Figure 4.7, processing the MSR-prxy_0 will suffer the most from the removal of ST module. Processing less blocks means that ST is less aggressive on moving

cold data. This will result in cold or very hot data occupying their blocks longer, preventing these blocks from being utilized. On the other hand, a less aggressive movement would also mean less performance overhead.

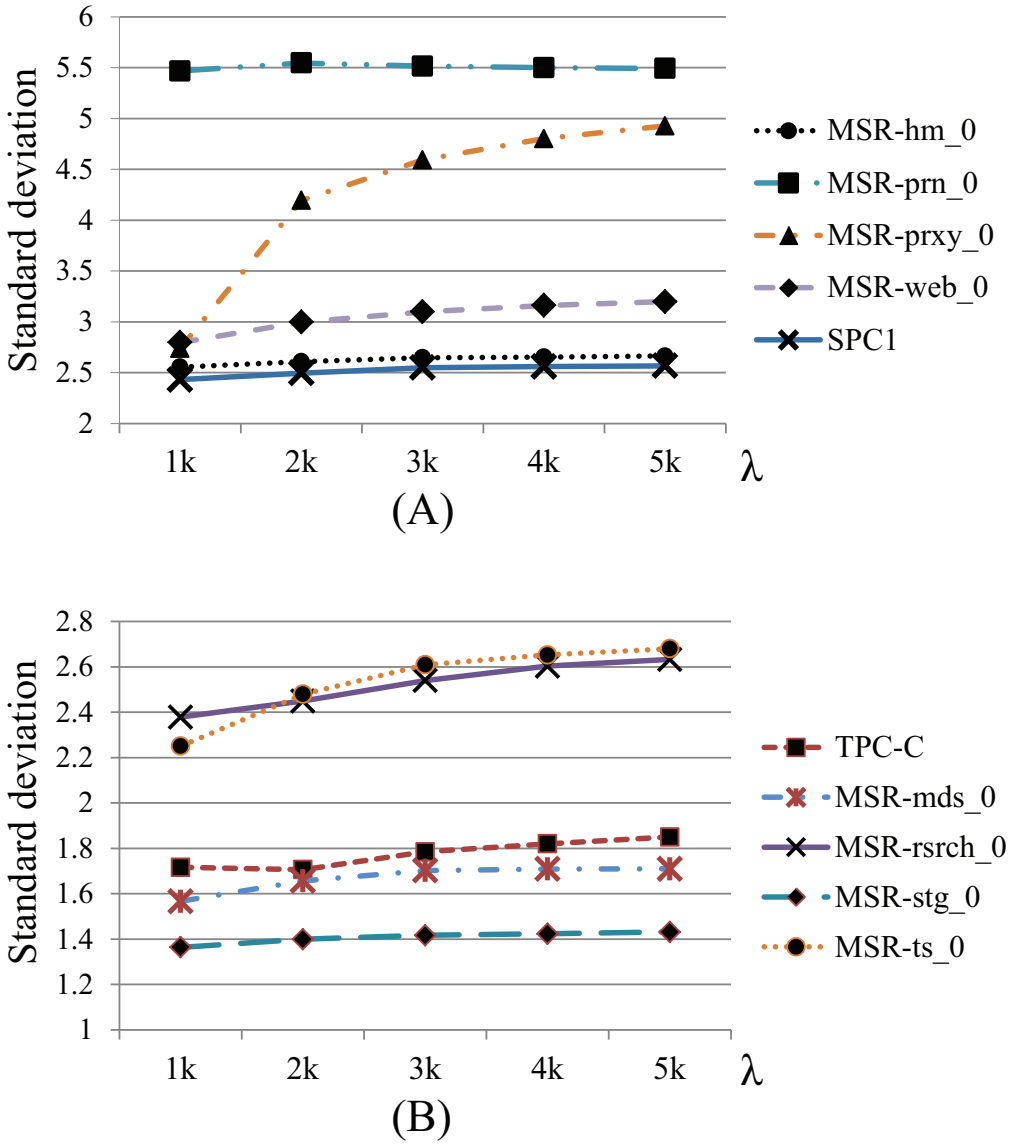

Figure 4.9: The Effects of λ length

Note that a larger δ value will cause ST to scan more blocks in the valid block pool. However, the effect is also dependent on the workload. From the three figures, we can see with most traces the impact of various δ is not significant. This is due to the access patterns of these workloads being quite uniform. But

for MSR-prxy_0 again, it is obvious in Figure 4.8 that results of $\delta = 0.4\%$ can be viewed as optimal. That is to say, scanning more blocks will incorrectly classify the data, and transfers based on such erroneous identification will only worsen the wear evenness. On the other hand, scanning less blocks may miss blocks that should be transferred.

We also did experiments to measure the effects of different λ. ST will be activated every λ interval. The default value of λ is 1000 requests. Figure 4.9 shows the standard deviations in wear evenness with λ being set at 2000(2k), 3000(3k), 4000(4k) and 5000(5k) requests. From the results we can conclude the effect of λ depends on specific workload. For MSR-prn_0 or MSR-stg_0, the interval length has no significant impact on wear evenness. For others, however, a longer λ will worsen the evenness. This is because ST will be less aggressive on a longer λ. With the same δ, ST will miss blocks that ought to be transferred. Still, for MSR-prxy_0, a more frequently executed ST module can greatly enhance wear evenness.

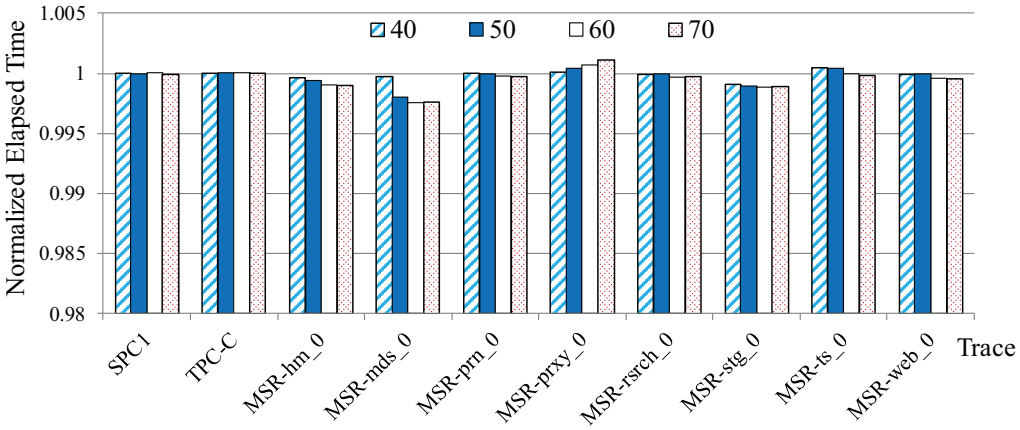

Figure 4.10: Normalized Elapsed Time with Various Γ

Figure 4.10, 4.11 and 4.12 show the results upon various values of Γ. Evidently Γ only has a marginal impact on wear evenness in most cases. Note that Γ is the threshold for identifying very hot data. If a block stays in the valid block pool for more than $(\Gamma \cdot \lambda)$ requests, its data are most likely to be very hot. The default value of Γ in previous experiments was 50. We conducted more experiments with Γ being 30, 40, 60 and 70 ($\lambda = 1000$ and $\delta = 0.4\%$). Figure 4.10 shows that the elapsed time did not change much with various Γ (results of $\Gamma = 30$ are used to normalize other settings). Neither did average erase count in Figure 4.11 (results of $\Gamma = 30$ are used to normalize other settings).

In most cases, the standard deviation of erase counts was not affected as shown in Figure 4.12. However, with a longer interval, MSR-prxy_0 would result

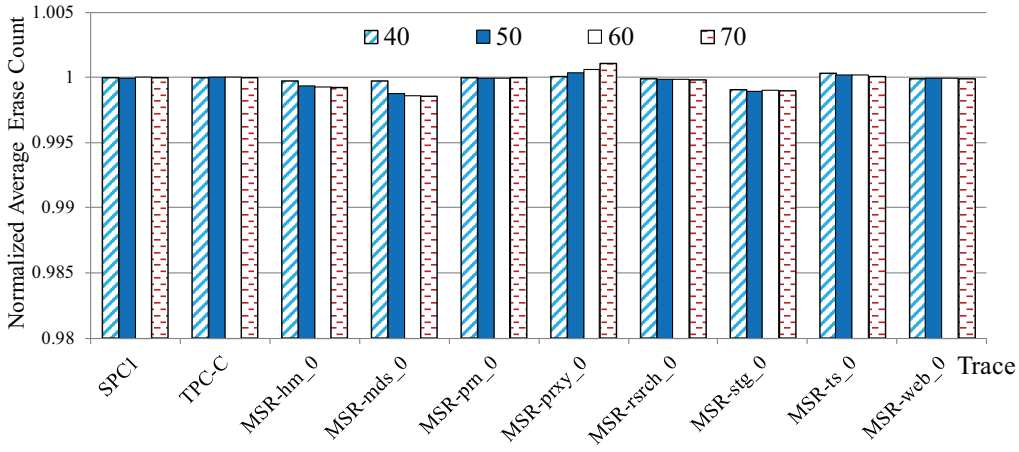

Figure 4.11: Normalized Average Erase Count with Various Γ

in slightly more erasures for better evenness, while MSR-mds_0 suffered from wear unevenness with slightly less erasures. For the former trace, a bigger Γ could result in more data blocks being identified as very hot. The characteristics of MSR-prxy_0 have been described before. Because small requests were frequently issued, a longer interval (in more requests) would be more suitable and could help to accurately filter out very hot data. Thus ST modules had lower standard deviation with the increase of Γ. This also confirms our argument in Section 4.6.2 and 4.6.4 that ST has played an important role in processing MSR-prxy_0. For MSR-mds_0, which is taken from a media server with a majority of big requests, a longer interval may miss blocks of very hot data with the same δ depth to scan, and less erasures would be performed by the ST module in transferring the data. This explains why as the interval was lengthened, the erase counts decreased and the wear evenness worsened for MSR-mds_0, as shown in Figure 4.11 and 4.12(C).

4.7 Summary

In this chapter an innovative algorithm for wear leveling is presented, i.e., the Observational Wear Leveling (OWL). OWL is based on the cooperation between the module of wear leveling and the module of address mapping. Hybrid address mapping facilitates the module of wear leveling to satisfy allocation requests for blocks. The cooperation is not through exchanging messages in between. Instead we have done it in a deep way: a part of the wear leveling module is embedded into the address mapping module, so that the former can get the immediate information to allocate flash blocks. Technically speaking, OWL records the

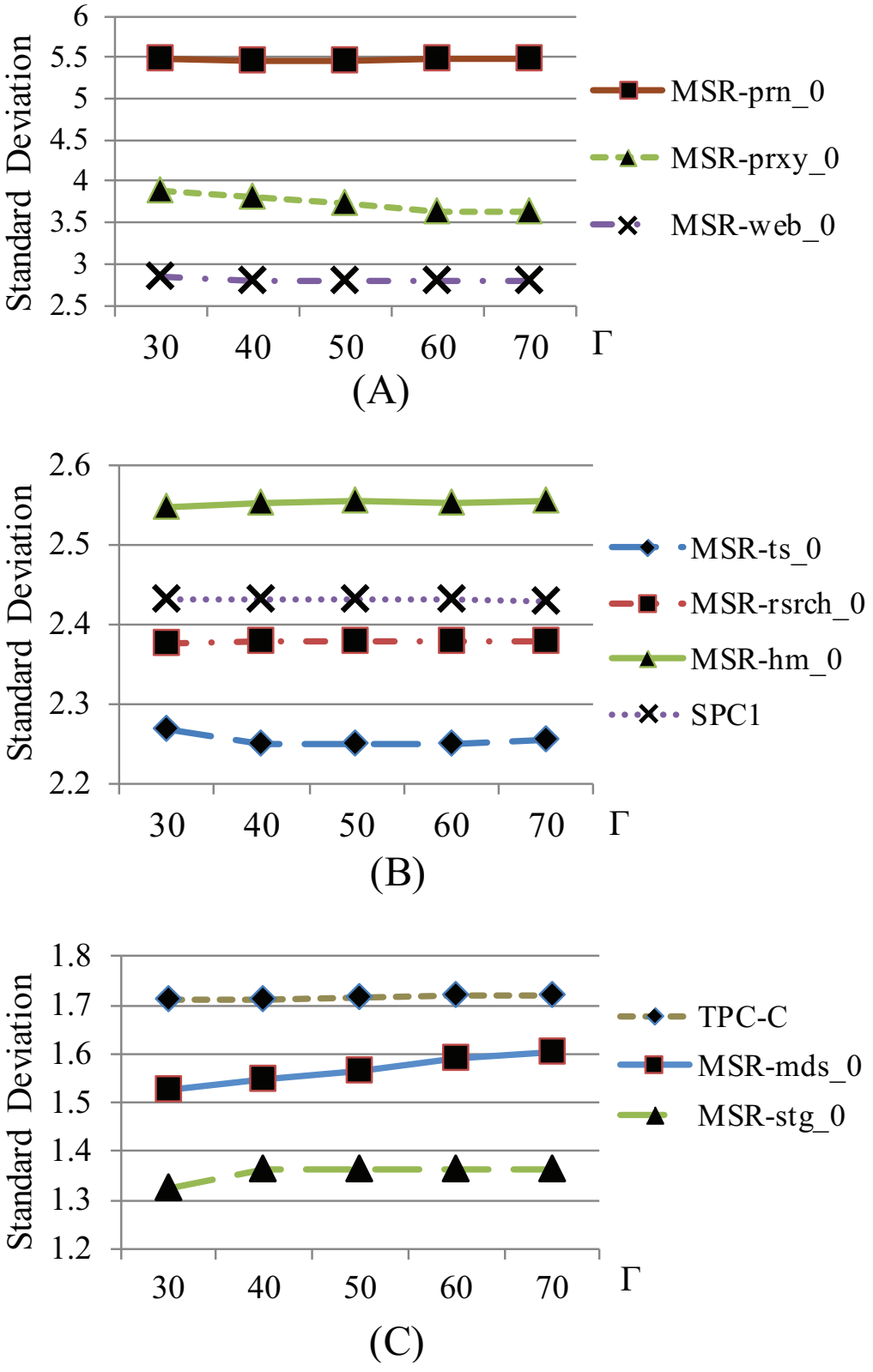

Figure 4.12: Standard Deviation with Various Γ

temporal locality of write activities at runtime, and allocates blocks judiciously in the merge procedure of hybrid address mapping. To do that, it makes use of a block access table (BAT) to make decisions. In order to further even out erasures, OWL employs a scanning and transfer mechanism to identify and move cold or very hot data. Experimental results show that OWL outperforms a state-of-the-art wear leveling algorithm on the evenness of erasures by as much as 43.2% with about 1.1% performance degradation, and a space overhead of 3KB. The evaluation therefore confirms our hypothesis of the module cooperation for flash memory management.

Chapter 5

ADAPT: Workload-Adaptive Hybrid Address Mapping

The second step of this thesis is emphasized on the adaptation to workloads. In this chapter, a scheme for the address mapping module will be presented. We name it ADAPT. With it we attempt to suit the FTL to dynamic access behavior of a workload. ADAPT falls into the category of hybrid address mapping. It has the traditional block divisions like the fixed log space, the block-mapped data space and the free block pool. It also activates merge to make space in the log space. However, ADAPT has been developed to online adapt to workloads. Essentially, it adjusts the partitioning of the log space in response to sequential and random write requests issued during runtime. By observing online access behaviors of workloads, ADAPT also avoids premature merges by predicting the likelihood of future references.

5.1 Overview

Address mapping is a basic functionality of the FTL. The combination of basic page-level mapping and block-level mapping makes the hybrid mapping more prevalent in market today. However, environments in which hybrid mapping is employed vary greatly. Primary hybrid mapping FTLs, like BAST and FAST, target embedded systems. With the widespread use of SSDs in enterprise servers, workload characteristics of general-purpose computing systems have to be considered. For instance, FAST pays attention to random writes using only one log block for sequential writes [60]. FAST's successor FASTer [64], however, focuses on online transaction processing (OLTP) systems. Workloads of many application domains, such as finance and commerce, are OLTP in nature. Typical

OLTP workloads are dominated by small and random I/O requests. A high-level access skewness exists on a handful of pages with other pages rarely touched.

Table 5.1: I/O Request Size of Various Workloads

Trace	Small	Medium	Large
TPC-C_20	99.17%	0.83%	0.00%
SPC1	86.58%	10.63%	2.79%
MSR-hm_0	76.70%	13.72%	9.58%
MSR-mds_0	72.35%	19.79%	7.86%
MSR-prn_0	79.46%	8.88%	11.66%
MSR-prxy_0	87.91%	6.82%	5.27%
MSR-rsrch_0	68.22%	25.04%	6.74%
MSR-stg_0	72.33%	18.62%	9.05%
MSR-ts_0	67.81%	25.87%	6.32%
MSR-web_0	67.50%	23.85%	8.65%

Besides OLTP there are many other important types of I/O workloads. For instance, mail and media servers serve contents that may be fairly large. These types of workloads differ from OLTP in that accesses are less skewed and generally more data need to be accessed in a request; sequential and random write requests may mix in different ratios and form dynamic access patterns, which requires FTLs to adapt to them efficiently for high access performance. The hybrid mapping FTLs need to be enhanced to target the variety of workloads.

Table 5.1 shows that the variation in I/O request sizes is significant. Traces from [84], [101] and [77] were used again. Here we define a *small* request as one that is 4KB (2 pages with 2KB per page), or less. This same definition was used by previous works [62, 64]. A *medium* request is one whose size is smaller than 16KB (8 pages), and any request that is larger is classified as *large*. For preliminary analysis, we roughly deem large requests to be sequential, which agrees with LAST [62]. TPC-C_20 in Table 5.1 is a typical OLTP workload which hardly has sequential writes but is almost full of random requests in all 7.7 million write records. MSR-prxy_0, one that was taken in a proxy server and also has a large amount of small writes, contains a lot of large requests. For non-OLTP workloads in Table 5.1, sequential writes compose about 3% to 12% of all requests. If these requests are handled, for example, with one log block as in FAST, there will be high capacity misses that can badly degrade the performance. For small random requests, since they are frequent and interpose with sequential

writes, how to satisfy them is always attractive in the development of FTLs. One key insight of our design is that the FTL should use an intelligent strategy to deal with workloads that are mixed with sequential and random writes.

Based on this observation, we have developed a hybrid mapping scheme. We have named it ADAPT. One reason of this name is that ADAPT highlights the proposal's ability of being workload adaptive. Another reason is that ADAPT can stand for two sub-modules of the scheme: ***A****ggregated* ***D****ata movement* ***A****nd* ***P****redictive* ***T****ransfer*. Traditional hybrid mapping schemes partition the log space into sequential area and random area to receive respective updates. However, the partitioning of previous algorithms is fixed. On the other hand, the access behaviors of workloads are changing. Therefore, our ADAPT applies a policy to dynamically adjust the partitioning at runtime. Besides, the merge procedure is optimized to avoid premature merge. As is covered in Chapter 3, during a merge multiple writes and two erasures have to be performed. ADAPT utilizes two mechanisms called the predictive transfer and the aggregated move to avert the arduous merging works. Experimental results show that ADAPT is as much as 35.4%, 44.2% and 23.5% faster than a state-of-the-art hybrid mapping scheme, a prevalent page-level mapping scheme, and a latest workload-adaptive mapping scheme, respectively, with a small increase in RAM space requirement.

5.2 Online Adaptive Partitioning of the Log Space

How to efficiently handle sequential writes and random writes is an important issue in FTL design. As mentioned before, the log space is partitioned into the sequential and random areas. Hybrid mapping schemes always expect sequential writes to cause switch or partial merge. FAST utilizes one log block as the sequential area [60] while LAST considers multi-tasking environments and employs a fixed number of log blocks to handle sequential requests [62]. On the one hand, using one log block tends to result in block thrashing. On the other hand, since the system workload changes from time to time, it is not optimal to reserve a fixed number of blocks also.

Before presenting our design, let us first revisit the issue of identifying sequential write requests. FAST uses two conditions to direct a write request to the sequential log block: (1) if its page number is zero within the logical block (data that the log block holds at present will be merged), or (2) the logical block number of the write request is the same to that of the sequential log block and the pages to be written can be simply appended in the log block. The first condition is likely to incorrectly label a random request beginning from page zero

to be a sequential one, and may result in frequent merges. For LAST, besides using more blocks for sequential writes to avoid such merges, it also takes into account the size of a write request: if a request writes data to a number of pages, it will be a sequential request. LAST was implemented in PCs of Windows XP operating system, so its threshold was set to be 4KB (2 pages) [62]. For ADAPT, however, we do not use an absolute number of pages accessed in a request to determine whether it is sequential or random. Instead, we will adaptively change the threshold. How this is done will be described below.

We shall now present our area partitioning scheme of ADAPT. Unlike FAST or LAST, the sizes of the sequential and random area are adjusted dynamically. The key idea is that, at runtime, if performance suffers from having insufficient sequential log blocks, blocks will be transferred from the random area to the sequential area, and vice versa. To do these, ADAPT maintains two variables in a time interval. The first is the *switch and partial merge ratio*,

$$\delta = \frac{count\ of\ switch\ and\ partial\ merges}{count\ of\ sequential\ log\ block\ allocation}.$$

This is the count of switch and partial merges over all block allocations from the sequential area. Another one is the *full merge ratio*,

$$\varphi = \frac{count\ of\ merged\ pages\ in\ full\ merges}{count\ of\ full\ merge}.$$

This is the average number of merged pages in the full merges occurring in the random area.

δ and φ represent the situations of recent write requests in a period inside the sequential and random areas, respectively. One reason of using δ and φ is that they reflect the effect of partitioning in an intuitive way. Another reason is to measure δ and φ is not difficult at runtime, so the overhead can be minimized.

δ varies from 0 to 1. A larger δ means a higher hit rate of block allocation in sequential area. Evidently enlarging the capacity of sequential area is likely to be profitable. If $\delta > \Delta$, we will do so. Δ is the exponential moving average of δ over past intervals, and how to obtain Δ would be shown. On the other hand, a smaller δ implies more requests were incorrectly treated to be sequential, and hence the need for sequential log blocks is not high. φ is an integer between 0 and the number of pages in a block, typically 64 [38]. A larger φ means that on average a full merge has to process more valid pages. So having more blocks in the random area may alleviate the pressure. We will enlarge the random area when $\varphi \geq \Phi$ where Φ is another threshold for φ. Φ is also the exponential moving average of φ over past intervals. A smaller φ implies random requests are handled

well by the current random area size, and possibly some blocks can be transferred to sequential area. By measuring δ and φ, ADAPT can adjust the utilization of blocks in both areas. To avoid significant fluctuation on performance, ADAPT will transfer one block every time between two areas. If δ suggests increasing the sequential area, ADAPT will select a victim block in the random area, merge it with its relevant data blocks and reclaim it. A clean block will be allocated to be a sequential log block then. The random area can be adjusted likewise.

Let us present how the values for thresholds Δ and Φ are set. One more parameter, κ is first introduced. $0 \leq \kappa \leq 1$. δ would be used for elaborating the calculation. As δ is measured period by period, we can have an average value of δ over past intervals. It can be immediately used for the threshold. Say, if δ of the current interval is much more than the average, the sequential area would better be enlarged; otherwise whether to enlarge the random area will be check. This sounds reasonable, but the average value may inaccurately indicate the situation as the farthest intervals have equal impact due to the common way to calculate the average. That is why we utilize κ to control the impact of recent intervals. For the interval n, the average Δ would be computed as

$$\Delta_n = \kappa \cdot \delta_{n-1} + (1 - \kappa) \cdot \Delta_{n-1}, \tag{5.1}$$

where $n \geq 1$. Δ_1 stands for the first interval and it is initialized to be zero.

If $\kappa = 0$, Δ would be meaningless as over time Δ would be set to the initial value, i.e., zero. If $\kappa = 1$, $\Delta_n = \delta_{n-1}$; δ measured in the last interval will be used as the threshold, and other past intervals are ignored. If $0 < \kappa < 1$, however, using Equation 5.1 we can have

$$\begin{aligned}
\Delta_n &= \kappa \cdot \delta_{n-1} + (1 - \kappa) \cdot \Delta_{n-1} \\
&= \kappa \cdot \delta_{n-1} + (1 - \kappa) \cdot [\kappa \cdot \delta_{n-2} + (1 - \kappa) \cdot \Delta_{n-2}] \\
&= ... \\
&= \kappa \cdot \delta_{n-1} + (1 - \kappa) \cdot \kappa \cdot \delta_{n-2} + ... \\
&\quad + (1 - \kappa)^i \cdot \kappa \cdot \delta_{n-i-1} + (1 - \kappa)^{i+1} \cdot \kappa \cdot \delta_{n-i} \\
&\quad + ... + (1 - \kappa)^{n-1} \cdot \Delta_1.
\end{aligned} \tag{5.2}$$

So Δ_n will have an average value of δ from past intervals. Equation 5.2 explains why we call Δ the *exponential moving average*. The method to compute an exponential moving average has been utilized in the design of operating systems [97]. It is very useful, and we will make use of it again in Chapter 7. Here the default value of κ is 0.9. A relevant discussion on κ will be raised in Section 5.6.

The calculation of Φ is similar. If φ is larger than Φ, the random area is advised to be enlarged. In ADAPT, the enlargement of the sequential area has a higher priority than that of the random area. That is to say, ADAPT will consider δ before φ. There are several reasons for this. Firstly, switch and partial merges are less expensive. Secondly, sequential log blocks are managed using block mapping, which consumes less RAM space. Thirdly, the utilization of random log blocks can be optimized with ADAPT's predictive transfer and aggregated movement components, which will be covered in next few subsections.

Unlike LAST's predefined 4KB threshold, the threshold of ADAPT to direct a request to the sequential or random area is also adaptive. In a recent interval, a very small δ, say less than 0.1, means that the sequential area was not very effective. This will cause the threshold to be changed. From our observation on enterprise workloads, over a long period of time, sequential writes tend to access a similar number of pages, either a handful (around 2 pages) or a large number (about 32 pages). Thus, if the threshold is very low, ADAPT will increase it to a large value. Otherwise, the threshold will be decreased. This simple adjustment is quite easy to implement. From our experiments, however, we saw that a latency might be needed to gradually adapt to a specific workload.

Algorithm 2 shows main steps in adjusting the log space. The adjustment is activated every `INTERVAL` requests (line 2 to 5). The impact of the interval length will be discussed in Section 5.6. We check δ first at line 8. If it is not positive, we will check φ at line 14. The partitions are then adjusted as described above. A victim block is picked from one area and merged with its data blocks (line 9 to 10 or line 15 to 16). A free block will be allocated from the free block pool to replenish the other area (line 11 to 12 or line 17 to 18). The way to select a victim in the random area is the same as a common merge procedure. ADAPT organizes the random area as a FIFO queue like FAST and FASTer, and the head will be the victim each time (line 9). For the sequential area, however, it is better to find a block that will make a switch merge or partial merge, which is computed by the function at line 15.

Note that our adaptation is different from previous works [80, 55]. They adapt by changing the degree of associativity between the random log space and the data space. In other words, a log block may be changed from being shared by many data blocks to being bound to one. ADAPT's adaptivity focuses on the partitioning of log space to service either type of write request efficiently. ADAPT also differs from WAFTL whose adaptation is in the transfer of data from the buffer zone to either the page or block mapping areas of the data space.

Algorithm 2: Adjustment of Log Space in ADAPT

```
1  begin
2      reqst_count++;
3      if (reqst_count < INTERVAL ) then
4          return;
5      end
6      else
7          reqst_count := 0;
8          if (δ > Δ) then
9              victim := GetHeadofRandArea(void);
10             Merge(victim, RW);
11             free_blk := AllocFreeBlock(void);
12             AddtoSeqArea(free_blk);
13         end
14         else if (φ ≥ Φ) then
15             victim := GetVictimofSeqArea(void);
16             Merge(victim, SW);
17             free_blk := AllocFreeBlock(void);
18             AddtoRandArea(free_blk);
19         end
20         return;
21     end
22 end
```

5.3 Predictive Transfers

The second chance scheme is the main feature of FASTer. FASTer gives valid data in the victim log block a second chance thereby preventing premature merges. With the second chance scheme, valid data from the head block of the random area will be moved to the block at the rear of the queue. FASTer performs well for OLTP systems because they frequently access little data from some very hot logical pages and not too many data would be left in the victim log block for movement. For other classes of workloads, however, such movements seem wasteful. While the second chance scheme can reduce the number of erasures, it may significantly increase the amount of data copying. Table 5.2 shows typical latencies of write and erasure of NAND flash [38]. It can be deduced that moving five pages will reverse the gain of an avoidance of an erasure. This is especially detrimental if a page given a second chance turn out not to be accessed during the time it is in the log space. This leads to the idea that if we can predict the likelihood whether a page in the random area will be used, we can make a better decision on whether to delay merging this page or not. In

general, a page at the front of the random area that is likely to be accessed again should be given a second chance in a merge process. Otherwise, if it is unlikely to be updated in the near future, then it should be directly merged.

Table 5.2: Latencies of Large-block SLC NAND Flash Memory [38]

Read	Write	Erase
130.9 μs (2KB)	405.9 μs (2KB)	2 ms (128KB)

As with most forms of prediction, the principle of temporal locality can be applied here. Particularly, a page that has been written recently is most likely to be updated again. We utilize the historical write information of a page to predict its future access possibility. Hence, on deciding if a page should be given a second chance, we shall examine whether its data were recently updated.

The data structure for prediction of ADAPT is the *historical access table* (HAT). HAT records a history of recent writes to logical pages at runtime. It is a hashed queue maintained in RAM. The key used for hashing is the logical base address in a write request, i.e., the concatenation of the logical block and page numbers. Each entry consists of the key and the number of pages that was written within a historical request. Hashing allows for queries about the existence of an entry to be answered quickly. Entries in the HAT are updated dynamically and managed via a queue discipline. On a coming write request, if its logical base address and the size to be accessed are already cached in the HAT, it will be moved to the rear; otherwise, a new entry will be enqueued in the HAT. If the HAT is already full, the entry at the head of the queue (the least recent one) will be deleted to make room for the new entry. In this way, we maintain a history of the most recent writes for the purpose of prediction.

The overhead of HAT is insignificant. It is resident in RAM together with address mapping tables, thus having a much shorter access latency than flash memory. The HAT does not need to be persistently stored since access behaviors are always changing and not correlated over a long time. The HAT is also small. As shown in Figure 5.1, each entry of the HAT has two fields, the base page number (4 bytes), and the number of pages accessed (2 bytes). Thus, 1KByte of RAM can hold about 170 requests. Our experiments showed that a 1KByte HAT could perform well. More discussions of the size of HAT will be given in Section 5.6 with various configurations.

Figure 5.1 gives an instance of merging with prediction. A rectangle is a physical block, and each has four squares for four pages. The number in a

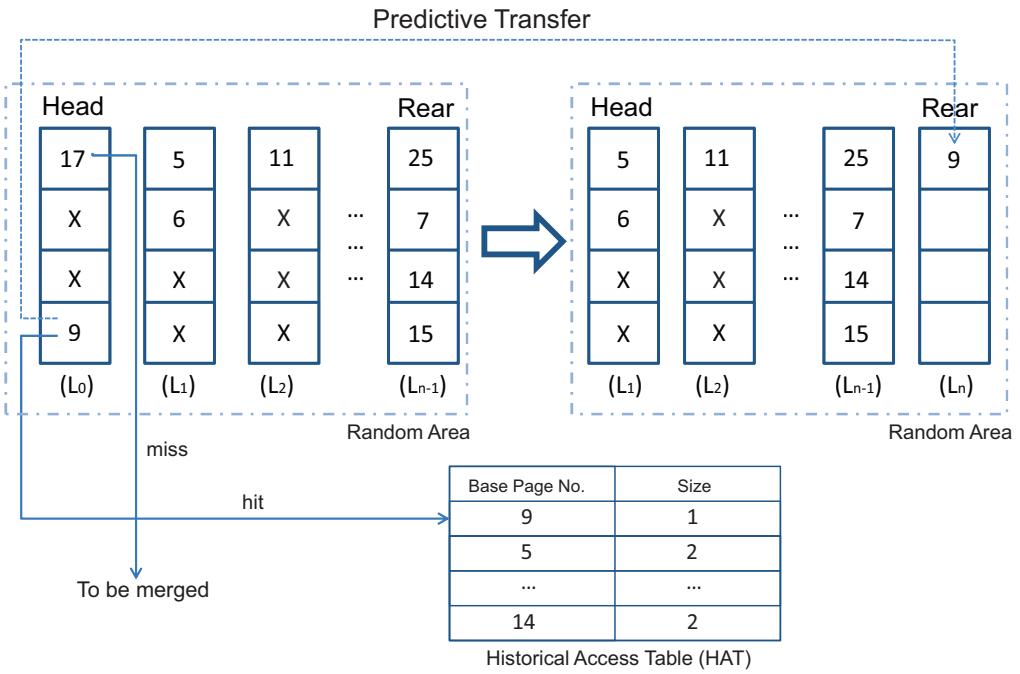

Figure 5.1: Predictive Transfer with the Historical Access Table

physical page represents the mapped logical page. An 'X' means invalid data that will be skipped in merging. Suppose the random area space runs out of free pages. A merge procedure will be performed. Firstly, a new block (L_n) will be allocated from the free block pool, and enqueued to the rear of the random area. In return, block L_0 will be removed and examined. In Figure 5.1, there are two valid pages in block L_0, namely page 0 and page 3. Page 3 corresponds to logical page 9. Its access record exists in the HAT, and so it is given a second chance, i.e., it will be copied into block L_n. However, the record for page 0 (which maps to logical page 17) cannot be found in the HAT, and it will be merged immediately with its relevant data block.

ADAPT's predictive transfer is different from the *adaptive merge* of a recently proposed hybrid mapping FTL named MAST [93]. MAST uses 2D-striping to access data. When merging has to be performed, MAST will also migrate valid log pages to other log blocks. However, in merging or migrating a log page, MAST considers the logical block it is related to. If that logical block is cold, and its total number of related log pages is bigger than a fixed threshold, the log page will be merged. Otherwise it will be migrated. Therefore, MAST's criterion is the number of log pages that a logical block is using, while ADAPT utilizes the recent access history of the logical page of the corresponding log page.

5.4 Aggregated Data Movement

As we have observed in our experiments, with the workloads from media and file servers, the victim log block to be merged may still have a lot of valid pages. I/O requests of non-OLTP systems may not be that small, as is shown in Section 5.1. Especially in multi-tasking environments, the access to storage may be switched to other applications frequently, thereby leaving many log pages valid even when they are to be merged. It is inefficient to process these pages one at a time. Therefore, we propose an *aggregated data movement* scheme to solve the problem. The example in Figure 5.2 will be used to explain this scheme. When the random area runs out of free pages, the merge procedure will be called. Unlike before, we shall first examine the two blocks at the head of the random area, i.e., L_0 and L_1 in Figure 5.2. If the number of valid pages in L_0 does not exceed an *aggregated move threshold*, τ, or if *both* L_0 and L_1 exceed τ, we will just merge L_0 with its relevant data blocks using the predictive transfer described above, i.e., a situation similar to Figure 5.1. The only remaining case is when L_0 exceeds τ, but L_1 does not. Then we will instead merge L_1, but move L_0 to the back of the log space, just ahead of the newly allocated block that is resulted from the merging of L_1, as shown in Figure 5.2.

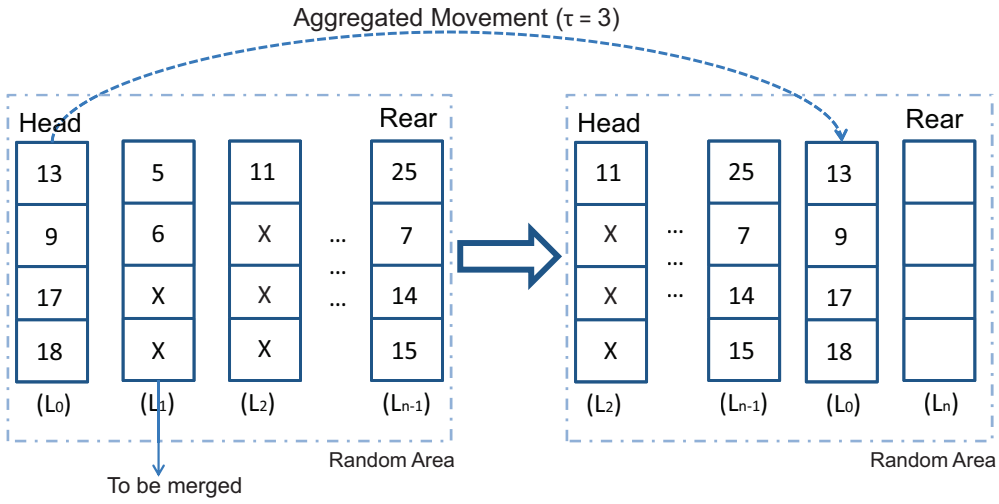

Figure 5.2: Aggregated Data Movement

Here we only consider two blocks at the front of random area. It is because scanning too many blocks will cause performance degradation. Another reason is that we do not want to change too much the FIFO manner of random area.

To support ADAPT's aggregated data movement and ERL modules, we need to know the number of valid pages in each log block. We assume that this is also

stored in a table in RAM. It is possible to store this information in the spare area of blocks [87], but the access latency will be longer. The space requirement for such a table is also comparatively low. A block typically comprises of 64 pages, and one byte is sufficient to store the total number of valid pages. Since log blocks typically take up 3% of the overall capacity, an xGBytes flash SSD with the standard 128KBytes large block configuration, will require a total of $0.24x$KBytes of RAM to store the per-block valid page counts. A 64GBytes SSD, for instance, will require a table of less than 16KBytes. This is quite small compared to the main block mapping table which is about 2MBytes (assuming each entry has a 3-byte physical block number and one byte for mapping status).

The ADAPT FTL scheme that we propose consists of the online adaptive partitioning of log space, the predictive transfer, the aggregated data movement, and ERL described above. We shall now give more details about the implementation of ADAPT, especially during the full merge to make decisions.

5.5 Merge or Move Decision Procedure

Algorithm 3 outlines the decision making procedure that is executed in the merge procedure in ADAPT. It first locates the victim log block to be merged, the new head and rear of random area, as well as the numbers of valid pages of the victim and the head block (line 2 to line 6). At line 7, it checks whether aggregated movement needs to be performed. If so, it will append the block to the rear of the random area (line 9), set the corresponding flag (line 8), and attempt to merge the next block to create the space (line 10 to 11).

If the condition for aggregated movement is not met, each valid page of the log block will be checked (line 14 to 28). At line 19, the HAT is queried to see whether the page has been accessed recently. If so, it will be moved to the rear block (line 21). Otherwise, it will be merged with relevant data block (line 24).

Note that in the implementation, we have two levels of merges, one at the block level (line 11) and another at the page level (line 24). This adds flexibility to resource management at runtime.

5.6 Experiments

5.6.1 Configurations and Assumptions

In this section we will evaluate the effectiveness of ADAPT using a number of workloads. The experiments were conducted using the FlashSim simulator [53]. We implemented FASTer, DFTL [29], WAFTL and ADAPT in FlashSim for

comparison. All the parameters of the NAND flash, including the latencies of read, write and erasure which are shown in Table 5.2, were obtained from [38].

Algorithm 3: Merge Decision Procedure in ADAPT

```
1  begin
2      victim := GetHeadofRandArea(void);
3      head_log_blk := RenewRandAreaHead(void);
4      rear_log_blk := GetRearofRandArea(void);
5      vp_no_vic := GetValidPageNo(victim);
6      vp_no_hd := GetValidPageNo(head_log_blk);
7      if ((vp_no_vic ≥ τ) && (vp_no_hd < τ)) then
8          AG_MOV := true;
9          Insert(rear_log_blk, victim);
10         new_head := RenewRandAreaHead(void);
11         MergeBlock(head_log_blk, AG_MOV);
12         return;
13     end
14     else
15         page_no := 0;
16         while (page_no < BLOCK_SIZE) do
17             state := GetPageState(victim, page_no);
18             if (state == VALID) then
19                 flag := IsHATHit(victim, page_no);
20                 if (flag == true) then
21                     MoveData(victim, page_no, rear_log_blk);
22                 end
23                 else
24                     MergePage(victim, page_no);
25                 end
26             end
27             page_no + +;
28         end
29     end
30     return;
31 end
```

To assess ADAPT's effectiveness, we utilized ten traces from three sources. They have been introduced in Chapter 4. SPC1 is a trace that was collected at a large financial institution [84]. Another trace is a typical OLTP trace from the TPC-C database benchmark [101], TPC-C_20. Other traces are the MSR-series from Microsoft's data centers [77]. The I/O characteristics of these traces have been presented in Table 5.1. We believe these traces are representative of various workload scenarios. The number of write requests in these traces is at least a million. We did not use other shorter traces found in some previous works.

There are also several assumptions in our experiments. Firstly, as with earlier works [64], we assume that the FTL has sufficient DRAM buffer to hold all mapping tables required by FASTer. DFTL and WAFTL were configured to have the same capacity of RAM as FASTer. ADAPT needs less RAM space than FASTer for mapping tables because more log blocks are managed using block mapping for sequential writes. Secondly, the traces used were collected from different machines. Therefore, we had to assigned a capacity configuration to each one based on their access patterns and lengths so that they are of more or less the same scale.

We evaluated each scheme by the elapsed time (or alternatively referenced as service time) to complete the simulation in FlashSim, together with counts of write and erasure. FlashSim has a module that accumulates the time caused by reads, writes and erasures as well as bus competitions on the chip. However, because the absolute value varies significantly with each trace, we chose to present the results in a normalized form. For ADAPT, the HAT size was set to 1KB and the aggregated move threshold τ was 56 by default. The length of the interval to measure δ and φ was 4000 write requests. There will be more discussions about the values of τ, δ and φ later. The default value of κ is 0.9. As with previous works, log space was set to be 3% of the overall data capacity [60, 64]. The buffer zone of WAFTL also took up 3% of data capacity as originally proposed [111]. Since FAST used one block [60] and LAST used 1/16 of the log space [62] for the sequential area, the lower and upper limits of ADAPT's dynamic sequential area were one block and 1/16 of all log blocks by default, respectively.

5.6.2 Performance Evaluation

Figure 5.3 shows the elapsed time for simulating each trace under DFTL, WAFTL and our proposed ADAPT, normalized against that of FASTer. Figure 5.4 and Figure 5.5 show the erase and write counts, respectively, of WAFTL and ADAPT normalized against those of FASTer. FASTer, WAFTL and ADAPT combine page mapping and block mapping in a similar way, and utilize parts of flash blocks for buffering. On the other hand, DFTL does page-level mapping, and its overheads include loading and evicting mapping between flash and RAM. Thus our comparisons using write and erase counts exclude DFTL. The rightmost bars in the three figures represent the sum of ten traces' results normalized against the total for FASTer. Let us first compare ADAPT with FASTer and WAFTL since they share similar designs on flash management. It is evident from Figure 5.3 that ADAPT outperforms them, consuming 35.4% less time than FASTer at best for the SPC1 workload, and 26.5% less than WAFTL for MSR-mds_0

workload. In all, ADAPT is 17.4% and 11.7% on average faster than FASTer and WAFTL respectively.

There is an interesting phenomenon in the case of the TPC-C_20 trace. FASTer was developed for OLTP applications. Even so, from Figure 5.3, we can see that for the TPC-C_20 trace, ADAPT is still marginally better than FASTer. Since I/O requests are predominantly random and small in this OLTP workload, with access severely skewed, there is little opportunity for ADAPT's mechanisms to exact its maximum impact.

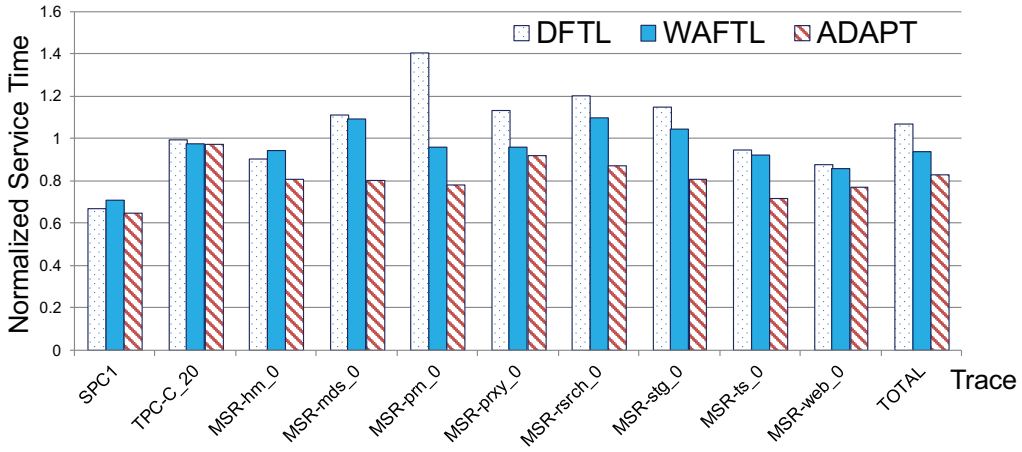

Figure 5.3: Normalized Elapsed Time of DFTL, WAFTL and ADAPT

Figure 5.4 and Figure 5.5 are the results for write and erase counts, respectively. From the two figures we can see that in every trace, ADAPT performs less writes and erasures than FASTer and WAFTL. However, there is something interesting to note in the results. In Figure 5.4 we can see that for MSR-mds_0, FASTer needs four times more erasures than ADAPT, but the results in Fig-

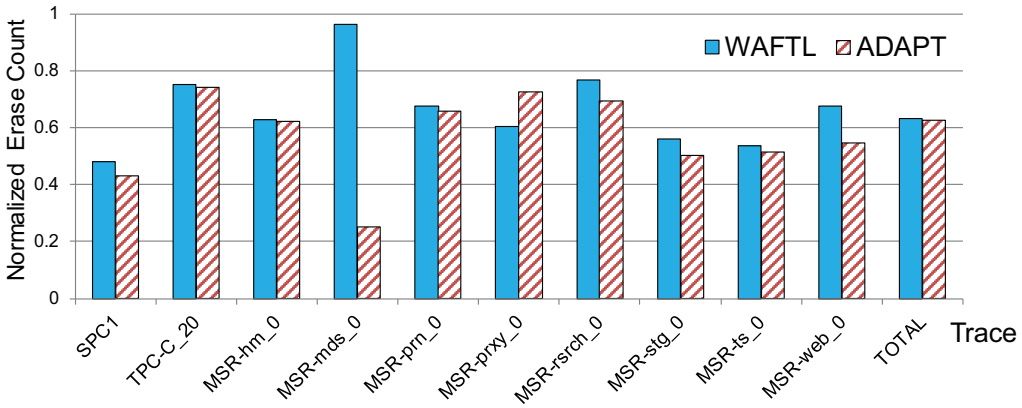

Figure 5.4: Normalized Erase Counts of WAFTL and ADAPT

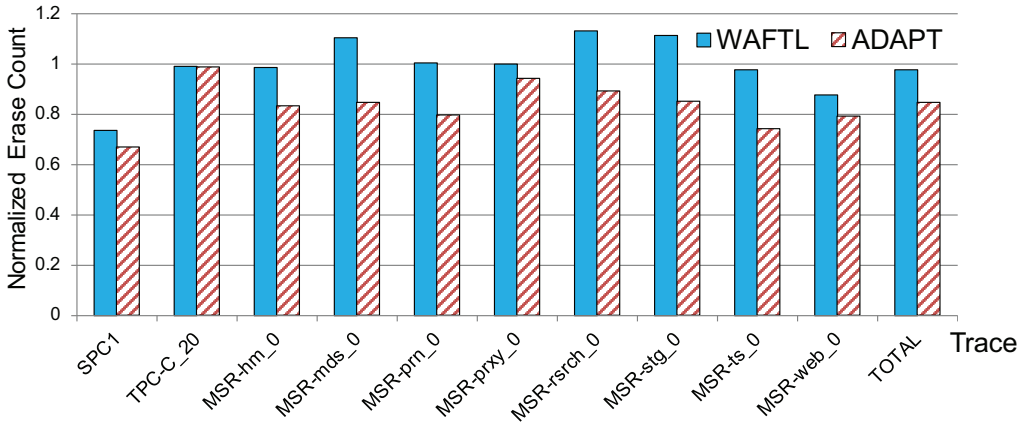

Figure 5.5: Normalized Write Counts of WAFTL and ADAPT

ure 5.3 show FASTer is merely 19.7% slower than ADAPT. This is because the performance is mainly dominated by the number of write operations at runtime. For MSR-mds_0 in Figure 5.5, ADAPT has only 15.2% less writes than FASTer. Consequently, the overall improvement of performance is not as significant as the reduction on erase counts would suggest. The situation is also the same for WAFTL and ADAPT executing MSR-ts_0. WAFTL has less erasures, but slightly more writes making it worse than ADAPT. Moveover, WAFTL was designed to flush all data in the buffer zone because it wants to take advantage of the integration of logical blocks that are buffered. However, to move all data is not trivial, and it will take too many writes and erasures. On the other hand, ADAPT attempts to leave data in the buffer for a longer time to avoid unnecessary movements.

We also implemented a state-of-the-art page mapping scheme. Since lazyFTL was said to have a similar performance to DFTL [69], we have selected DFTL for comparison. The results are also presented in Figure 5.3, normalized against those of FASTer. From the figure, we can see ADAPT is faster than DFTL by as much as 44.4% for the case of MSR-prn_0. Unlike FASTer or WAFTL that considers characteristics of one or more types of workload, DFTL merely loads page-mapping entries to RAM on demand, and handles sequential and random writes in the same way. For MSR-prn_0, 9.46% of its requests would write more than 64KB (32 pages) at a time. ADAPT could respond well to such access patterns. However, these continual large writes from multi-tasks would cause DFTL to reclaim blocks frequently for clean pages as well as load and evict mapping entries, thereby badly degrading overall performance.

Table 5.3 shows the prediction hit rates and the number of aggregated move for each trace. The hit rate is high for most traces. For SPC1, even with a rela-

Table 5.3: Prediction Hit Rates and Aggregated Moves

Trace	Prediction Hit Rate	Aggregated Moves
SPC1	79.50%	132
TPC-C_20	100.00%	0
MSR-hm_0	95.68%	233561
MSR-mds_0	96.49%	1727
MSR-prn_0	99.93%	124608
MSR-prxy_0	99.72%	8323
MSR-rsrch_0	98.75%	2050
MSR-stg_0	93.24%	1045
MSR-ts_0	95.16%	1165
MSR-web_0	96.99%	5408

tively lower hit rate, good performance can still be achieved by the cooperation of all modules in ADAPT. From Table 5.3, we can also see there is no aggregated movement for the OLTP TPC-C_20 trace, and the prediction hit rate is 100%. This agrees with our earlier analysis in Section 5.1.

Aggregated movement and predictive transfer affect each other. If an aggregated move is performed on a block, then its pages will stay longer in the log space. This will result in the block at the rear of the random area having many valid pages. If the heuristics are correct, many of the pages will be accessed again soon, leaving the remaining pages to be processed by predictive transfer when this block again reaches the head of the random area. Therefore, aggregated movement and predictive transfer complement each other.

5.6.3 Effects of Log Space Capacity

The impact of the log space capacity was also investigated. We performed experiments where the log space was provisioned as 3%, 5% and 10% of the overall capacity. The results are shown in Figure 5.6. We normalize the results for provisioning 5% and 10% of space as log space against that for 3%. It can be seen that generally performance improves with larger log spaces. However, the extent of effect varies. For some traces, the impact on performance is significant, but for others, such as TPC-C_20, it is not. We believe this is particularly noteworthy for SSD users to utilize resources.

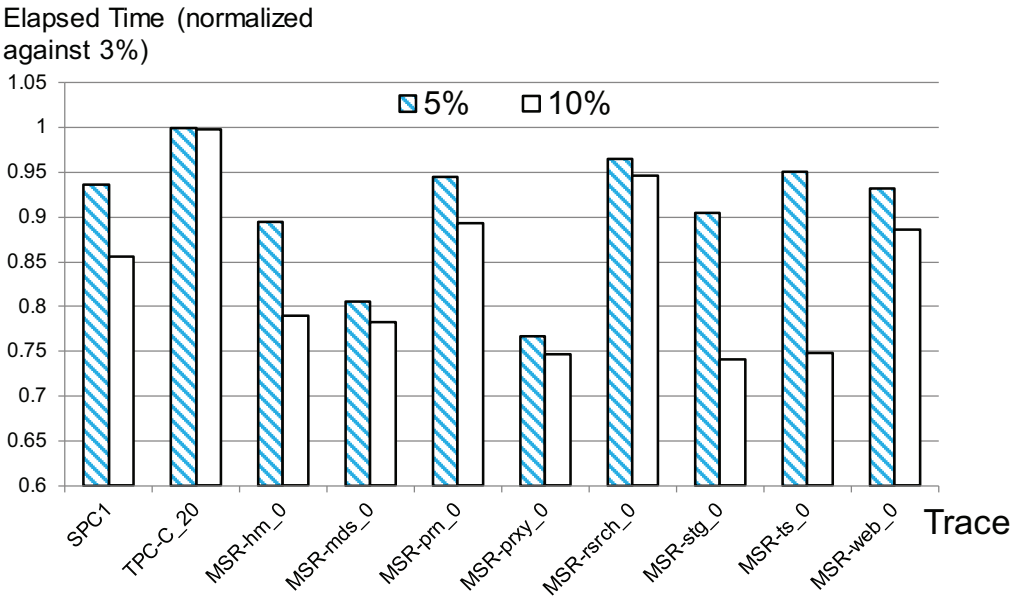

Figure 5.6: Effects of Different Log Space Capacities

5.6.4 Effects of Log Space Partitioning

We also did experiments to verify the effects of adaptively adjusting the partitioning of the log space. In Figure 5.7, ADAPT and ADAPT-sw had the same configuration including predictive transfer and aggregated movement except that ADAPT-sw used only one log block for sequential writes, which is the same as FAST and FASTer. All results are normalized to those of ADAPT-sw. From Figure 5.7 we can see ADAPT can be faster by as much as 31.9% in the case of SPC1. However, TPC-C_20 is still special because it has almost no sequential writes as shown in Table 5.1.

Figure 5.8 shows the result for different thresholds used in the identification of sequential writes. We used three configurations. The first, ADAPT-2, has a threshold of 2 pages (4KB) which is the same as LAST. The threshold of the second, ADAPT-32, is 32 pages (64KB). The last one is the full version of ADAPT that adaptively adjust the threshold based on δ. The lower and upper bounds of ADAPT are set to 2 and 32, respectively. Results of the ADAPT-32 and ADAPT are normalized against those of ADAPT-2 and presented in Figure 5.8. From Figure 5.8 we can see that ADAPT is faster than ADAPT-2 and ADAPT-32 most of the time. But with some workload like MSR-ts_0, ADAPT had to spend 12.7% more time to finish the trace. We analyzed MSR-ts_0, and found that the feedback in current interval does not accurately reflect the access behaviors. The results in Section 5.6.6 will address this by showing

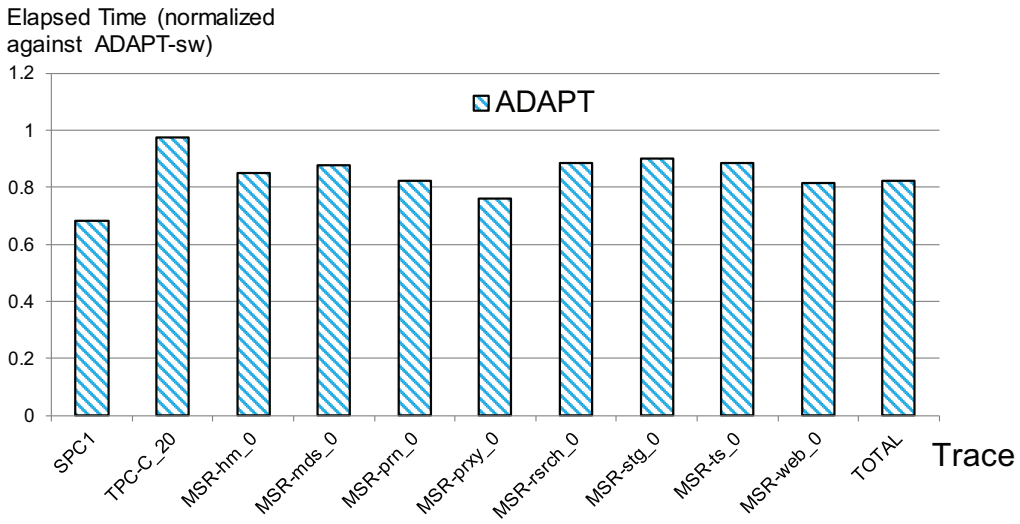

Figure 5.7: Performance Impact of Log Space Partitioning

how performance can be significantly improved with longer intervals.

5.6.5 Impact of κ

κ is used to calculate the two thresholds Δ and Φ for adjusting the partitioning. As $0 < \kappa \leq 1$, we did experiments with κ multiply configured. The results are shown in Figure 5.9 and Figure 5.10.

Note that κ controls the impacts of nearer and farther intervals, respectively. However, from the two diagrams we do not find significant fluctuation with different values of κ. It is because traces from real environments have well access patterns. To explore the reason for variant differences, we have captured two access periods of SPC1 and MSR-prxy_0, as shown in Figure 5.11. After 30% and 60% of all requests, we fetched one thousand consecutive write requests, respectively. We recorded logical page numbers those requests would access. From the diagrams in Figure 5.11 we can find that the access distribution of a trace is quite stable, although accesses may fall into different addresses. As a result, the δ measured in each interval would not badly vary, which hence explains the insignificant difference with variant κ in Figure 5.9 and Figure 5.10.

5.6.6 Effects of the Interval Length on Adaptation

ADAPT needs to observe and calculate δ and φ in an interval. By default, we used an interval of 4000 requests in the experiments. We also experimented with intervals of 2000, 3000, 5000 and 6000 requests. Their results are shown in Figure 5.12 and Figure 5.13. From the results, we can see the length of the

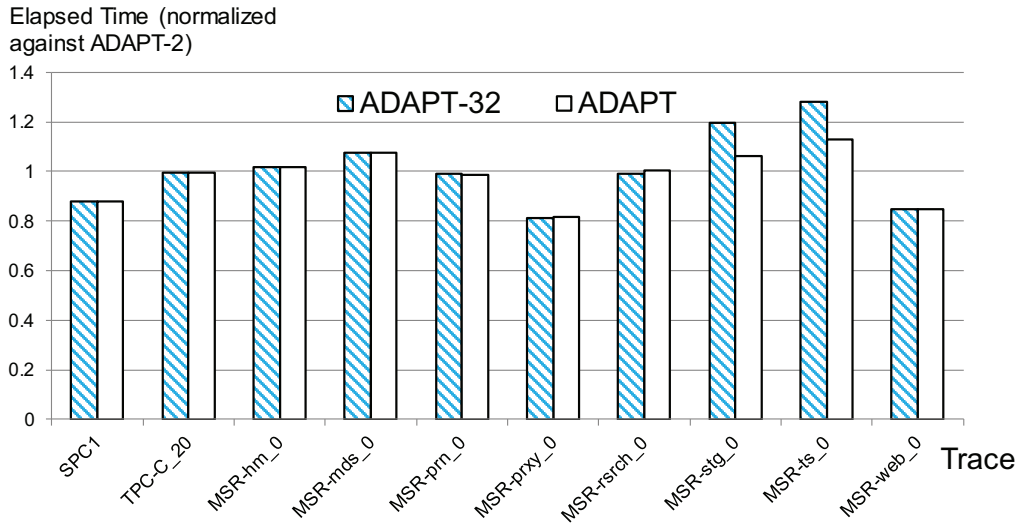

Figure 5.8: Impact of Different Sequential Write Identification Thresholds

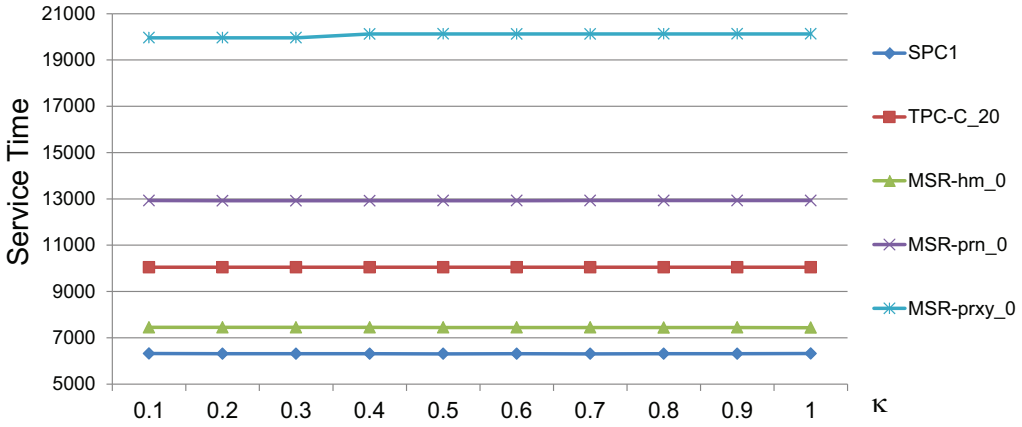

Figure 5.9: The Effects of κ (A)

interval hardly affects the performance. The reason is given by Figure 5.11: over a long time the access behavior remains similar for a trace. For MSR-stg_0 and MSR-ts_0, however, their results may fluctuate a little more. That means current configuration is too short to reflect the online behaviors. By prolonging the interval, better performance can be achieved, as shown in Figure 5.13. This agrees with results in Figure 5.8.

5.6.7 Effects of HAT Size

The HAT is an important data structure for ADAPT. Figure 5.14 presents four results of each trace when the size of HAT was configured to be 512, 1024 (1K), 1,536 and 2,048 (2K) bytes. The results for 512 bytes are used to normalize

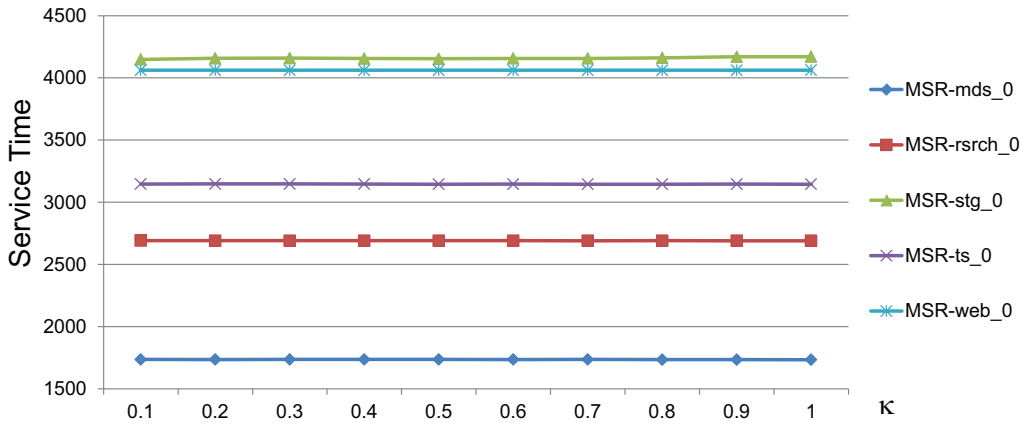

Figure 5.10: The Effects of κ (B)

the other cases. It can be concluded from these results that the optimal size of the HAT depends on the workload. Recall that the HAT is used to record the recent write history which is then used for prediction. Obviously, keeping too long or too short a history may result in wrong predictions. If the HAT is too big, it would store outdated access records, causing pages that should be merged immediately to stay too long in log space. If the HAT is too small, the prediction would not get a full view of the locality, and unnecessary merges may be performed.

The performance of ADAPT on TPC-C_20 trace changes slightly with different HAT sizes. This can also be attributed to its access patterns. Due to the skewness of the accesses in the OLTP workload, a small HAT suffices. It therefore makes little difference in enlarging the HAT. The results also suggest that due to the differences in locality, the size of the HAT should be tuned for each workload.

5.6.8 Tuning of Aggregation Threshold

The threshold τ to trigger aggregated data movement is an important parameter of ADAPT. For the ease of reading, we have separated relevant experimental results into Figure 5.15.

In accordance with [38], each flash block has 64 pages in our simulations. In general, when the number of valid pages in the log block to be merged reaches the aggregated move threshold, i.e., τ, the block would be moved, and the one next to it will be merged instead. If τ equals to 32, aggregated move will be performed if 50% or more pages of the block are valid. If it is 64, all the pages in a block will have to be valid in order for an aggregated move to be activated. Figure 5.15 show the impacts of various values of τ on access performance.

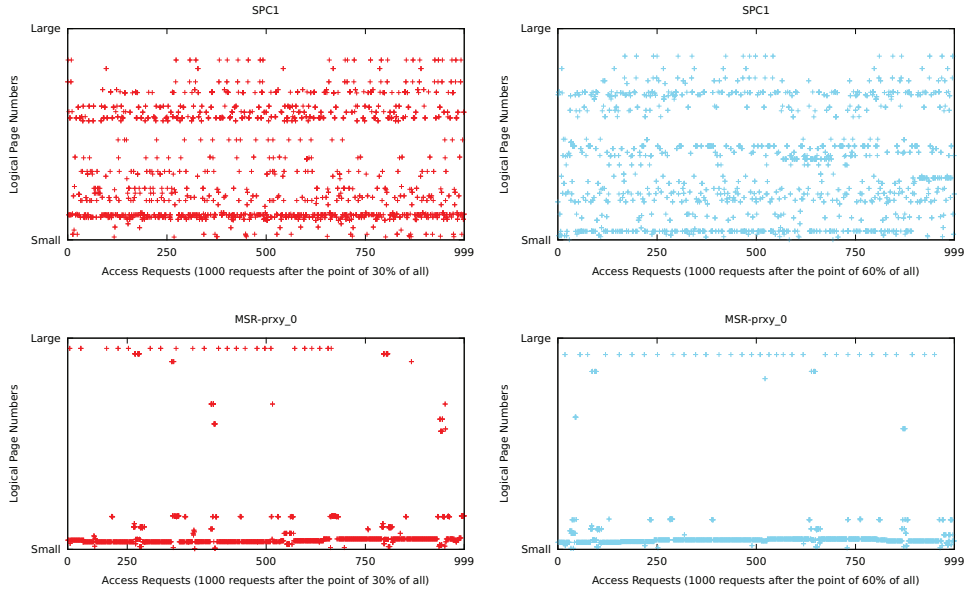

Figure 5.11: Captures of Access Distribution for SPC1 and MSR-prxy_0

We know that TPC-C_20 has no aggregated movements from Table 5.3. For other traces, we can see from the figures that for some of them, including MSR-ts_0, MSR-web_0, MSR_hm, and MSR-prn_0, the results are better with a higher τ. For others, such as SPC1, MSR-rsrch_0, MSR-mds_0 and MSR-stg_0, τ does not affect access performance. However, for the MSR-prxy_0 trace, access performance would degrade with larger τ. Again, we attribute this to the access patterns of the traces themselves. Traces in the first category generally have more valid pages in the log block to be merged than others. Hence, a higher τ is to improve the performance. For traces in the second category, the number of valid pages is moderate and stays fairly constant throughout the execution, and different thresholds showed little impact.

As for MSR-prxy_0, besides capturing the requests shown in Figure 5.11, we found out more about its access behavior. As discussed in Section 5.1, access requests with 2 pages (4KB) are considered to be small. But in MSR-prxy_0, there is a huge number of requests that are even smaller. 77.8% of the requests in MSR-prxy_0 only write to one page, and data in these pages are frequently updated. Moreover, they are scattered across the flash device. Thus the log block to be merged may have dozens of valid pages. However, a higher τ gives aggregated movement little chance to show off its advantage, leaving valid pages to be processed by the predictive transfer sub-module. A larger log window would absorb more small requests. From Figure 5.6 we can see it is MSR-prxy_0

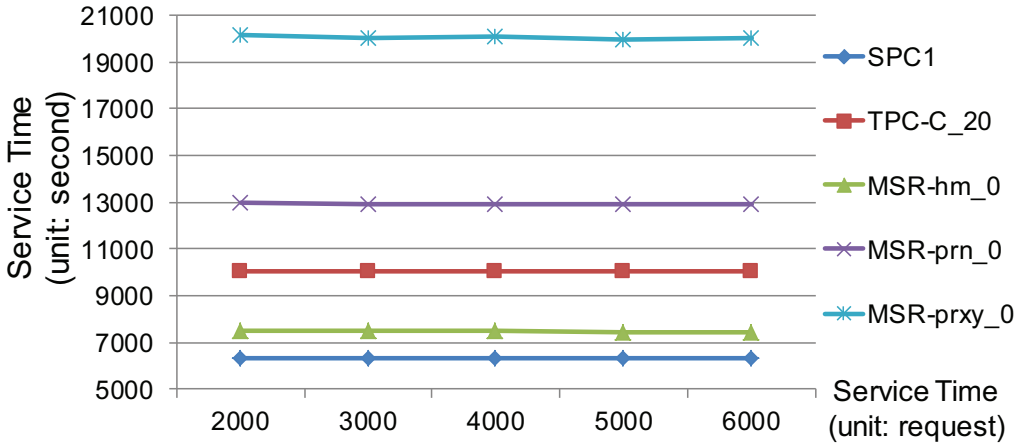

Figure 5.12: The Effects of the Interval Length (A)

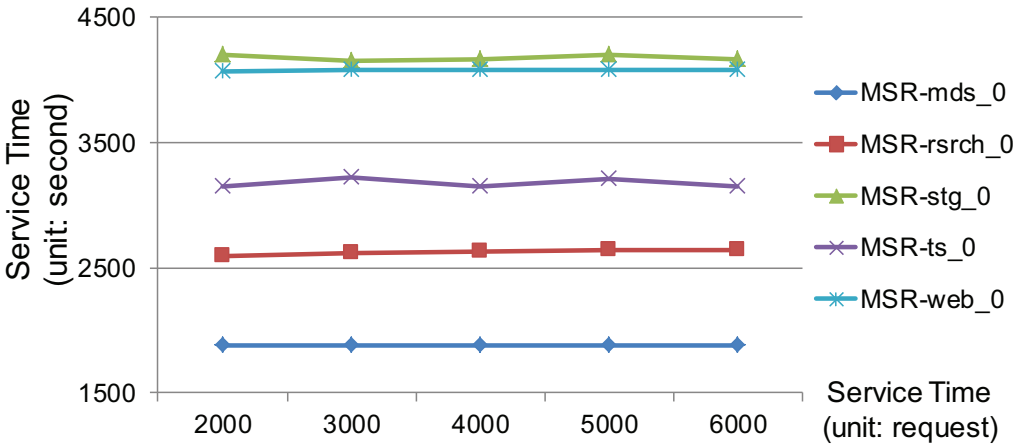

Figure 5.13: The Effects of the Interval Length (B)

that improves the most with larger log space.

5.7 Summary

Address mapping of the flash translation layer is central to the performance of flash-based devices. In this chapter, we presented ADAPT, a hybrid mapping FTL scheme that adjusts to various workloads by exploiting their access behaviors and temporal locality. ADAPT handles both sequential and random writes efficiently by dynamically tuning the partitioning of the two areas of log space that are used to process the respective types of writes. To do so, ADAPT collects statistics on how log blocks are used in each area, and then utilizes these statistics to adjust its parameters. ADAPT also explores the locality to reduce unnecessary data movements in full merges. It employs a prediction mechanism

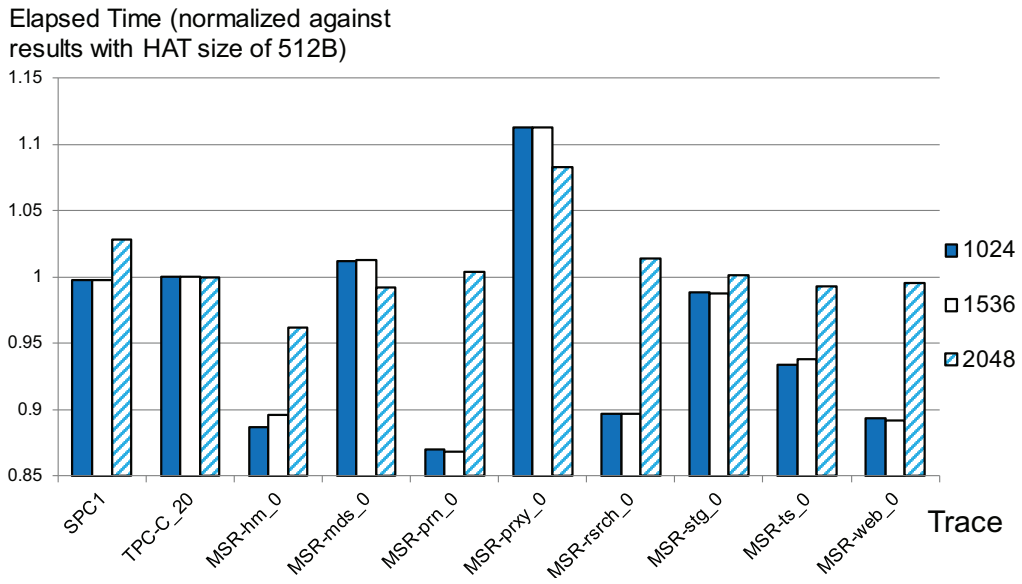

Figure 5.14: Effects of Different HAT Sizes

to decide whether a log page should be merged, or given a second chance. In addition, ADAPT records the number of valid pages in each random log block at runtime. If a block to be merged has more than a certain threshold of valid pages, the entire block would be kept in the log space. Our experiments show that ADAPT can outperform FASTer by as much as 35.4% with a modest additional RAM space requirement of less than 16KBytes for a 64GBytes SSD. ADAPT is also faster than DFTL and WAFTL by as much as 44.4% and 26.5% respectively. The advantage of ADAPT over previous mapping schemes verifies the idea of being workload-adaptive.

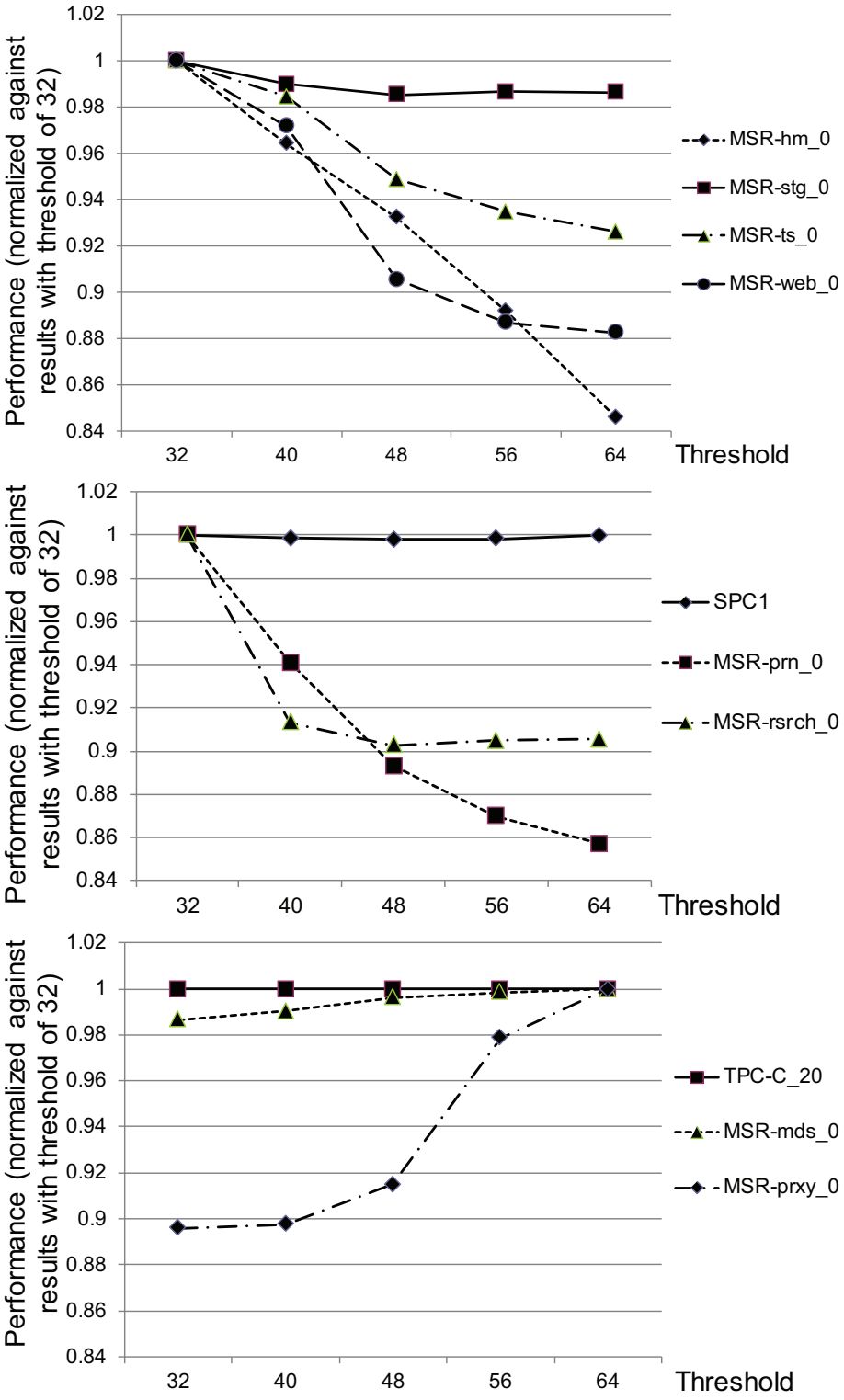

Figure 5.15: Performance of Aggregated Movement

Chapter 6

TreeFTL: An Adaptive Tree in the RAM Buffer

This chapter will show an algorithm that was also devised in the second step of this thesis. Its target is the RAM buffer of flash device. The RAM buffer is an ideal component to present the idea of workload adaptation as it is the one that the most reflects access behaviors of workloads. The proposal in this chapter, i.e., the TreeFTL is unlike sophisticated designs. It is succinct due to its simple *tree* structure maintained in the RAM space. With the tree, TreeFTL is able to adjust itself dynamically to serve access requests of workloads. Moreover, the tree structure also entails lightweight LRU victim evictions, which further enhance the TreeFTL to minimize overheads caused by context switch between workloads. In all, TreeFTL will well present our attempt to make the flash management adapt to online workloads.

6.1 Overview

The RAM buffer, also referenced as the RAM cache, is a very useful resource for NAND flash device. Even though flash memory is much faster than magnetic hard disks on the access speed, it is still slower than SRAM or DRAM. The RAM buffer is hence equipped for higher access performance. How to utilize the RAM buffer is a good spot into the management of flash device. The RAM buffer is the one that buffers address mapping entries and data pages to satisfy access requests. It is known that requests can be analyzed to figure out the access patterns of various workloads. Therefore, the contents cached in the RAM buffer contain meaningful implications of workloads.

Note that technically the RAM buffer of flash devices is unlike the cache

of processors. The issue of associativity of the CPU cache, for example, defines various mapping relationships between cache location and data. The RAM buffer yet has no such limitation. The space of the RAM buffer is flexibly assigned to workloads by the FTL. It is more like the main memory of computer systems.

Most of the primary investigations on the RAM buffer can be deemed to be *space-centric*. That is to say, the emphasis of RAM buffer management is to explore how to make the best use of the limited space. It is because the RAM buffer of an inchoate flash device is in a small capacity. At that time, the RAM buffer is mainly used for address mapping, especially for page-level mapping. Entries from the mapping table are partially cached for quick reference [29, 86]. If the FTL intends to access a location which is not touched recently, the correspondent mapping entries will be loaded into the RAM buffer on demand. In this way, the RAM buffer management module serves the address mapping module. Later, the temporal locality and spatial locality of workloads are taken into consideration to manage the RAM space to map addresses [86].

Data buffering is another important use of the RAM buffer. Obviously it costs shorter time to access data within SRAM or DRAM than NAND flash memory. Data are buffered in the unit of a page which facilitates to access data with NAND flash memory. One common issue of buffering data is how to flush cached pages back to flash with the least performance overhead, especially under a specific address mapping scheme. Algorithms like BPLRU [51], FAB [43] and REF [91] have been proposed to target such an issue.

Note that data pages cached in the RAM buffer are still recorded in the mapping table, which implies that the two uses of the RAM space are not insulated. In terms of the aim, they are both for the improvement of access performance. There are already proposals for the joint management of the RAM buffer for address mapping and data buffering [94, 35]. However, those schemes are either inefficient or ineffective, which motivates us to develop an algorithm that is advantageous both on efficiency and effectiveness. One essential requirement for the algorithm is the simplicity. It must not be too complicated as the RAM buffer management has to respond to access requests as swiftly as possible. Another goal is that the algorithm must be adaptive to running workloads, and the process of adaptation should be lightweight and feasible. As is mentioned, the RAM buffer reflects the access behaviors of workloads. Since workloads alternately access data from the flash device, should the context switch between workloads be too heavyweight, the algorithm would not be a promising one.

Hence we propose TreeFTL in this chapter. It jointly caches mapping entries and data pages in the RAM buffer. Specifically speaking, it maintains cached

translation pages and data pages in a tree-like structure. The translation page is a page that sequentially stores entries of address mapping table of flash memory management [29]. Entries of the cached translation pages connect to cached data pages according to the mapping relationship. The loading and eviction of pages between RAM and flash memory are on demand, which makes the tree naturally grow to adapt to workloads. A cached translation page and its connected cached data page form a caching group, which is an essential concept to perform the lightweight LRU victim selection. The overhead of context switch of workloads is therewith reduced. Experimental results show that TreeFTL is able to outperform the latest algorithms of RAM buffer management, like APS [94] and JTL [35], on service time by as much as 73.9% and 72.3%, respectively.

6.2 The Tree in RAM

The said tree-like structure maintained by TreeFTL in the RAM buffer is sketched in Figure 6.1. It has three levels. The first level and second level are used for address mapping, while the third level is for buffering data pages. Because the tree naturally grows upon access requests, TreeFTL is able to dynamically adapt to the runtime workloads. As for the context switch of workloads, TreeFTL is endowed by its tree structure to set up a lightweight mechanism to select the LRU victim to make space for use. The lightweight mechanism further strengthens the capability of TreeFTL on the workload adaptation. In following, we shall describe TreeFTL based on the page-level address mapping such as DFTL [29]. However, the basic idea of TreeFTL can be easily adopted to a block-level mapping scheme like DAC [85].

6.2.1 The Three Levels

As is mentioned, all mapping entries of demand-based page-level address translation can be stored in the translation pages of flash memory [29]. A structure named the *global translation directory* (GTD) is used to record the physical addresses of these translation pages. The GTD must be resident in RAM as it is the root directory for address translation. Hence, TreeFTL makes the GTD the root of its tree structure.

Figure 6.1 shows the conceptual tree structure of TreeFTL. In Level 1 is the GTD. Level 2 is taken up by the *cached translation pages* (CTPs), while Level 3 holds the *cached data pages* (CDPs). The three levels are connected by unidirectional links which practically represent the mapping relationships. When a translation page is loaded into RAM, its address in the GTD will be

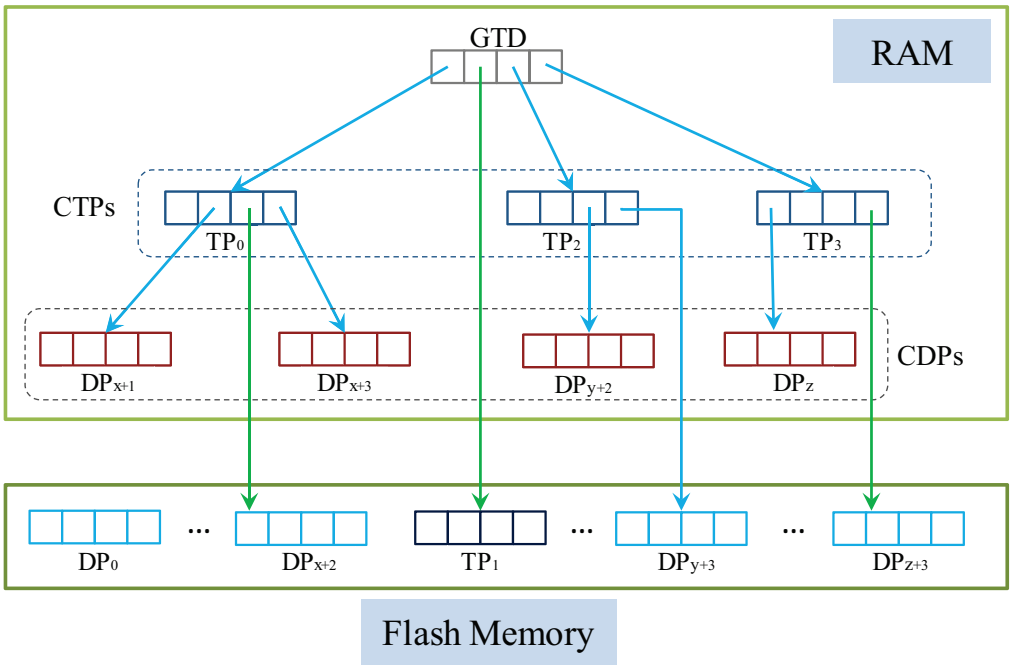

Figure 6.1: A Conceptual Structure of TreeFTL

updated to point to a RAM location accordingly. So TreeFTL treats the RAM cache as a part of storage medium. For a mapping entry in a CTP, if the data page is cached in RAM, the record will be the RAM address instead of a physical address in flash memory. CDPs in Level 3 are the leaves of the tree. They are cached upon write requests. TreeFTL emphasizes on the write buffer similarly as BPLRU [13], l-buffer [13], ExLRU [92] and APS [94] do, because for NAND flash memory the write latency is much longer than the read latency [81] (see Table 6.1). Furthermore, writes may trigger expensive erase operations which cost even longer time [51, 94].

Table 6.1: Latencies of SLC NAND Flash Memory [41]

Read Operation	Write Operation	Erase Operation
25 μs (2KB)	200 μs (2KB)	700 μs (128KB)

Let us first clarify the distinction between TreeFTL and existing schemes such as DFTL and CDFTL that are categorized as the on-demand address mapping algorithms. The structures cached in RAM space for DFTL and CDFTL have been shown in Figure 3.2. Ignoring data buffering, TreeFTL differs from them in

that the former does not cache single entries in a caching mapping table (CMT). There are two reasons to do so. First, to load or evict a single entry entails a read or write for a translation page, respectively. Although *batch update* [29] can group evicted entries from a same translation page, it complicates the design, in addition to the space overhead involved. Second, a translation page covers a wider range of consecutive logical addresses, so caching a translation page can benefit from the spatial access locality [86]. CDFTL keeps both single entries and translation pages in its cache. A translation page will be loaded also when one of its entries is fetched into the RAM cache by CDFTL. Thus, eliminating the CMT can avoid duplications, and hence save space. The process of address translation, which will be described below, is also simplified because in CDFTL a miss of the CMT requires consulting the CTPs first.

6.2.2 Address Translation With The Tree

Address translation in TreeFTL begins by finding the translation page to locate the desired mapping entry. This can be done by using the logical address as a hash key to look for the RAM location of the relevant translation page, which results in either a hit or a miss. However, in order to show the *growing* of the tree, we will describe the process in another way. The address translation process in TreeFTL can be viewed as a traversal from the root to some leaf of the tree. There are three scenarios for a random write request, as shown in Figure 6.2. The first case is when both translation page and data page are cached. In three steps (A-1, A-2 and A-3 in Figure 6.2) the data are written to the target CDP. No operation is performed on the flash memory. The second case is when translation page is cached but data page is in flash memory. The data page has to be loaded into RAM buffer first. So a read operation (B-3 in Figure 6.2) is needed. The third case is neither of the two is in RAM. In such a scenario, two reads have to be conducted (C-2 and C-4 in Figure 6.2). So this case is the most time-consuming.

Any CDP or CTP that has been selected as the eviction victim will be flushed back to flash memory. More details will be given in next subsection.

TreeFTL services read requests in a slightly different way. When a read request comes, the translation page will be loaded if it is not already cached. For the target data page, however, it will not be loaded into RAM buffer if not cached. Instead, the flash page is read directly from flash memory and the data are returned to the file system then.

In all, the process of address translation with the tree is just the process of the tree's growing. Evidently the tree grows in a natural way upon access requests.

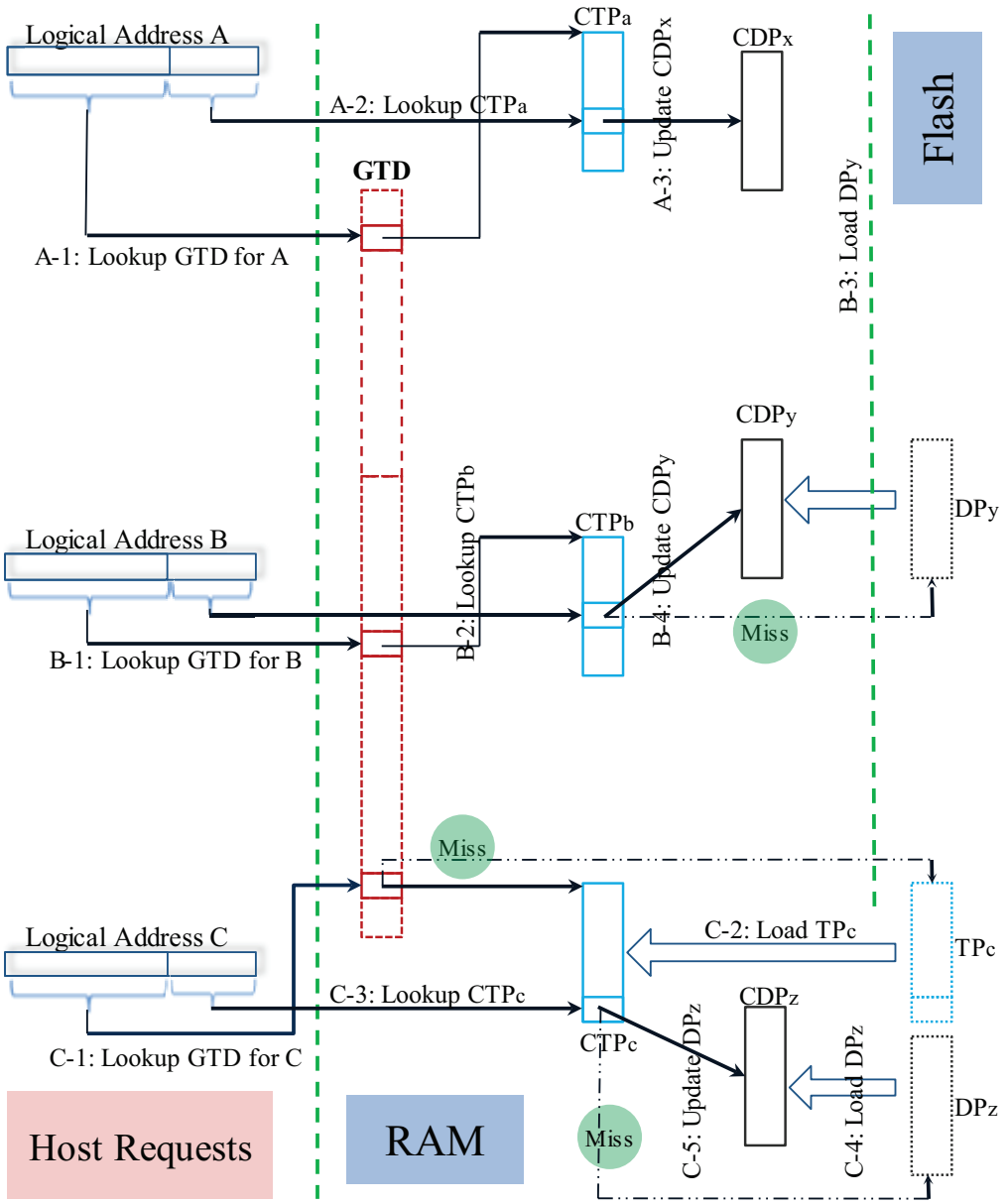

Figure 6.2: Address Translation Process in TreeFTL

Not to mention the consequent workload adaptation, the natural growing makes TreeFTL more preferable and feasible compared to ponderous and complicated methods to allocate the RAM space.

Note that the write or read request mentioned above is random access request. TreeFTL deals with sequential requests in a "write-through" manner. If data are written to or read from flash memory in a bulk RAM cache will be bypassed. This is similar to what JTL has done [35]. It is based on the assumption that data which are sequentially requested are likely to be infrequently accessed. There are many methods to identify a request to be random or sequential. For example, deciding based on the access size is a simple but effective approach [62, 103]. TreeFTL deems a request to be sequential if it intends to access more than half a block, i.e., 32 pages of 64KB data as in [41].

6.3 Lightweight Pruning of TreeFTL

6.3.1 Lightweight Pruning with Caching Groups

The tree grows on access requests. Nonetheless, because of the spatial limitation of the RAM buffer, it is not allowed to overgrow. Sometimes the tree has to be pruned. In other words, when RAM space is exhausted, a victim has to be selected and evicted. The victim ought to be the one that is the least recently used (LRU). APS [94] performs the LRU selection at the level of entries and pages among cached mapping entries and data pages, respectively. JTL's multi-level structure [35] helps it to find the LRU mapping entry or data page easily as less frequently accessed ones are moved down from RAM buffer to flash memory. However, both APS and JTL suffer from LRU victim selection. Assuming that all 64MB of a RAM cache is used for APS's data buffering, there would be 64MB/2KB = 32768 data pages in total. It is not trivial to find the LRU page each time in such a large number of pages. For JTL, its multi-level structure may have to be adjusted on each arriving request.

TreeFTL exploits its tree structure and takes a lightweight way to pick out the LRU victim. In this subsection, we will show the basic idea of TreeFLT's lightweight mechanism to find the LRU victim. Since TreeFTL uniformly caches pages which are just nodes in the tree, the eviction process is exactly like pruning a tree. To do so, TreeFTL introduces a concept of the *caching group* (CG). A CG is a group which includes a CTP and its relevant CDPs in connection. In terms of the tree structure, a CG is just a branch (sub-tree) in the RAM buffer. For example, there are three CGs in Figure 6.3.

Now with the concept of CG we would make use of the timestamps to present

the idea of the lightweight pruning of TreeFTL. Here recording the timestamps is applicatively chosen because it is an intuitive way to perceive and understand the advantages of our proposal against existing page-level selection mechanisms. In the next subsection, a more feasible algorithm for the real implementation shall be shown. Using timestamps, TreeFTL can maintain a hash table called the *Last Access Time Table for Eviction* (LATTE) that records the last access time for each CG. The hashing key of the LATTE is the number that identifies each CTP. This number subsequently identifies a CG. Each entry in the LATTE is a two-tuple. The first element is the time when any CDP of that CG is last accessed. The second element is the page number of the last accessed CDP, which ranges from 0 to 511 (2KB a translation page and 4B for a mapping entry [29]). Hashing enables the LATTE to be swiftly updated after access requests. A sketch of the LATTE is shown in Figure 6.3.

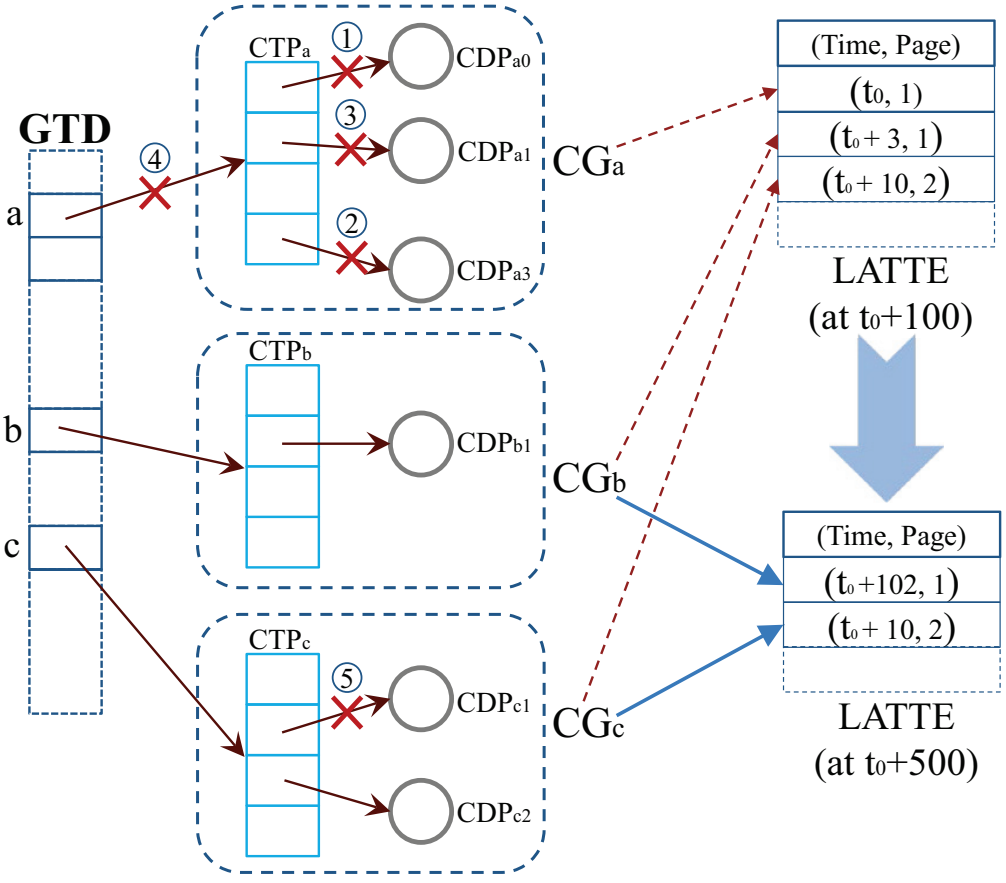

Figure 6.3: The Sketch of TreeFTL's Victim Selection

The victim selection is performed upon an eviction request. It first finds the *victim CG* that has the smallest timestamp. In Figure 6.3, at time $t_0 + 100$ the

victim CG is CG_a. Then the selection inside a CG starts. The CDP that has the smallest offset in a CG will be the victim page. If it is the one recorded in the LATTE entry, however, it will be skipped unless there is no other CDP left. In Figure 6.3, a circled number is the eviction sequence of a CDP. CDP_{a0} is firstly evicted and CDP_{a3} will follow. On the next eviction request, CDP_{a1} will not be skipped again since it is the last CDP of CG_a. This way all the CDPs of CG_a would be pruned. If no new data page joins CG_a (otherwise the LATTE will be updated) before next eviction request, CTP_a will be flushed back to flash as a victim page, and CG_a's entry in the LATTE will be removed.

TreeFTL's pruning policies can be summarized as follows:

- With the LATTE, the LRU selection is conducted at the level of a CG, not page.
- CDPs are preferred for eviction. The one that is the most recently accessed in a CG, i.e., the recorded one in the LATTE, would be the last to be picked.
- If a CG has no CDP left, and it has the eldest timestamp in the LATTE, the CTP will be evicted.

The first rule makes TreeFTL "lightweight" as the granularity of CGs is much coarser than that of CDPs, since a CTP can point to hundreds of CDPs. Spatial locality dictates that consecutive logical pages are likely to be accessed in a short interval of time. Hence, the timestamp of the last accessed CDP can be used to approximate a group's recency. This approximation saves RAM space and reduces processing time. It certainly suffers from the lack of detailed information about each CDP. However, our experiments (in Section 6.8) show that such trade-off is worthwhile to make.

The second rule states to evict CDPs is preferred. It is because a CDP is a leaf of the tree, and a CTP yet connects to tens or even hundreds of CDPs. Moreover a miss of a translation page needs two read operations, while a miss of a data page requires only one read. In addition, the CDPs recorded in the LATTE should be the last one to be evicted. Based on temporal locality, this CDP is the one that is the most likely to be accessed again. Other CDPs will be selected in the sequence of their offsets in their CTP.

The third rule dictates when a CTP is to be flushed back to flash. When all CDPs are removed from a caching group, the CTP can be evicted. However, TreeFTL's eviction is based on demand. Only when a request is raised for free space, will TreeFTL act. This also gives a CTP a second chance to stay for a while in the RAM buffer.

Let us give a discussion on the overhead of the lightweight LRU victim selection at CG-level compared to one at the page-level. We still use records of timestamps as the implementation to give the comparison. In the worst case, each CG just has a CTP and a CDP. The temporal overhead of the lightweight selection would be half that of a page-level selection, since a timestamp is used for two pages (a CDP and a CTP). The spatial overhead of the LATTE is the maximum in this case too. It is less than that of a page-level strategy. The second element of a two-tuple in the LATTE needs less space than a timestamp, and a two-tuple stands for two pages while two pages of a page-level strategy need two timestamps. Such an extreme case is rare. Since a caching group may have many cached data pages, at most 512, it can be expected that the overhead of maintaining and searching at CG-level is significantly less than that of a page-level policy.

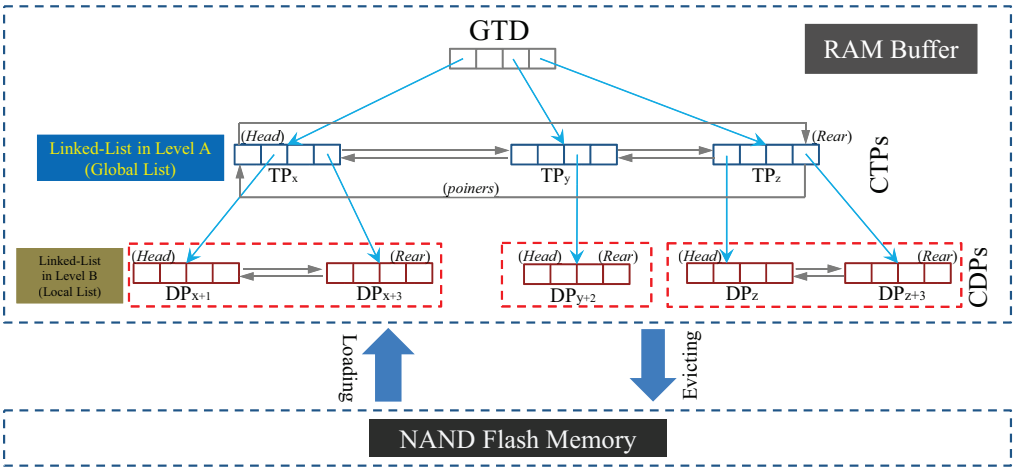

Figure 6.4: The Sketch of TreeFTL's Two-level LRU Selection Mechanism

6.3.2 Two-level LRU Selection Mechanism

The last subsection shows how the lightweight pruning could be conducted. It is based on the tree structure and the concept of caching group. The design in the last subsection is a classical one which is applicable for a RAM buffer in a small size. If the RAM buffer scales up, it might be arduous and unacceptable to maintain a table to record the last access time for each caching group. Not to mention the spatial overhead to temporarily keep the LATTE, the time spent to find the LRU caching group might be significant. Hence, we need a more feasible mechanism to implement the idea of lightweight LRU victim selection.

It is preferred to implement the LRU victim selection using a hash table and a doubly linked list [90, 94]. The hash table is used to quickly find the element

with a given key. The doubly linked list is used to dynamically reorder elements. Usually the most-recently-used (MRU) element is relocated to the front end, and the LRU elements are left behind to approach the rear end. By doing so the LRU victim can be picked out in $O(1)$ time. However, the lower temporal consumption is not free of charge. The hash table itself and the pointers of the doubly linked list inevitably take up the RAM space.

With the idea of linked lists, we have modified the tree structure. A two-level mechanism to conduct the LRU victim selection has been subsequently implemented. Figure 6.4 gives a sketch of the two-level mechanism. The Level A is one linked list that connects all CTPs. So we call it the *global* linked list. At Level B each CG has one local linked list. It is deemed to be local because CDPs of one CG are linked together but no CDP of other CGs is involved.

The idea behind the two-level mechanism is simple. Evidently it is still based on the tree structure and the concept of caching group. The difference between the two-level mechanism and a common LRU caching is that the former now has a global scope as well as multiple local scopes. The process of updating the pointers is straightforward. At the Level A, TreeFTL updates the head pointer to record the MRU CTP. Correspondingly, once a CTP of some caching group has been pointed, TreeFTL will update the head of local linked list to trace the one that is the MRU CDP. So each time TreeFTL performs a two-level updating. When a victim has to be found, TreeFTL can easily offer the CTP that the rear pointer of the global linked list in the Level A points at. Then the local linked list of that victim CTP's caching group is checked. The CDP the rear pointer of the local linked list indicates would be the victim.

The two-level mechanism is profitable and feasible for the implementation of the LRU victim selection. First, the global and local mechanism saves processing time. Even though to locate the LRU victim just costs $O(1)$ time, to adjust the head pointers to trace the MRU element is time consuming. For the two-level mechanism, however, it is avoided to traverse all cached pages to find the LRU victim as the local scope narrows down the area to be searched. The time is consequently reduced. Second, the two-level mechanism is accurate to locate the LRU victim. To adjust the head pointer of the global linked list in Level A leaves the LRU CTP to approach the rear. In Level B, by moving the head pointer of the corresponding local linked list, the LRU CDP of that caching group also comes out. Of course, the accuracy of the two-level mechanism is partially credited to the space TreeFTL takes up to maintain those doubly linked lists; the spatial overhead cannot be ignored. The third profit of the two-level LRU selection mechanism is yet on the reduction of spatial consumption. As is mentioned, a

hash table is needed to quickly locate the element in question. In Figure 6.4, however, there is no hash table employed by TreeFTL. This is contributed by the tree structure and the address mapping module of the FTL. Through the GTD-CTP-CDP mapping way, the CDP can be certainly found. So the hash table is not necessary any longer. The avoidance of a hash table in turn saves space of the RAM buffer.

6.4 Discussions on TreeFTL

6.4.1 Partitioning and RAM Space Utilization

The partitioning of RAM buffer has been investigated through different ways [94]. As for our proposed TreeFTL, the adaptive partitioning is inherently achieved. The tree naturally grows or is pruned upon access requests. The partitions for address mapping and data buffering are accordingly adjusted. There is no fixed division for the two parts.

A possible issue of TreeFTL is the utilization of RAM space. A CTP has many entries, and usually not all of them are connected to CDPs. So unused "holes" scatter across CTPs. The benefits of caching a translation page on spatial locality have been addressed in Section 6.2.1. In terms of RAM utilization, caching a page for a requested entry risks taking up more space, but the potential use of other entries in this page can save valuable time. The lower utilization of RAM cache is more likely to be caused by outdated mapping information and data pages. A RAM management module ought to efficiently identify and move them out.

6.4.2 Workload Adaptation

For TreeFTL the adaptation to workloads is also naturally achieved. It is mentioned that different applications have different access behaviors. An application may write and read on a large number of pages while another one just operates on a handful. TreeFTL does not specifically distinguish one application from the others. It is not necessary. All that TreeFTL does is satisfy requests of each application in an on-demand way. If an application intends to access more pages, more RAM space will be assigned to it; otherwise, that application just takes up a smaller space.

With the lightweight mechanism to perform the LRU victim eviction, the context switch between workloads for TreeFTL is not strenuous. The avoidance of heavyweight switch is attributed to the flexible tree structure. If the RAM

buffer is fixedly partitioned, the recycle and reallocation of RAM space for workloads surely will be a concern. The dynamic partitioning also enables TreeFTL to respond swiftly. TreeFTL takes actions the moment the context switch happens. It has no delay to so. In other words, the response time of TreeFTL to the context switch between workloads is ignorable, which ensures that TreeFTL could adapt to workloads efficiently and effectively.

6.4.3 Reliability and Garbage Collection

Reliability is an important issue of data storage, especially when the RAM buffer is used as a storage medium. DRAM or SRAM is volatile memory, and would lose data if the power supply is unexpectedly off. This problem has been addressed by using non-volatile memory such as the phase-change memory (PCM) [29, 47, 66]. However, even though non-volatile memories bring in better reliability, their innate characteristics must not be neglected. For example, the endurance issue of PCM is unavoidable [66, 57, 88]. Besides the utilization of non-volatile memory, a backup battery can otherwise be equipped. Moreover, CTPs and CDPs can be copied to flash memory when the storage system is idle.

As is mentioned, garbage collection is another important module of NAND flash memory management due to out-of-place updating as well as the time-consuming write and erase operations. If data cached in the RAM buffer can be frequently updated, it will alleviate the pressure on the module of garbage collection since less data will be sent to flash. In addition, the module of garbage collection is likely to take advantage of the knowledge of RAM buffer management to reclaim flash blocks occupied by invalid dirty data. How to make the module of RAM buffer management and the module of garbage collection cooperate is a good spot to look into.

6.5 Performance Evaluation

6.5.1 Experimental Setup

We still evaluated TreeFTL using FlashSim simulator [53] running on a Linux 64-bit system to simulate a 32GB NAND flash device. The compiler was GCC 4.6. The traces we used here are the same as what we used to experiment for OWL and ADAPT. First we restate their sources. SPC1 is from Storage Performance Council (SPC) [84]. TPC-C was collected within TPC-C database benchmark [101]. MSR traces are from Microsoft data centers [77]. We believe that they represent various workloads in the real world. The parameters of the flash memory used for the evaluation, as shown in Table 6.1, were obtained

from a recent datasheet [41]. In previous works, RAM access time was either ignored [13] or unclear [35, 94]. We assumed one RAM operation over a 2KB page costs $2\mu s$, which is the same as in [86]. The RAM capacity has been multiply configured, and will be shown below.

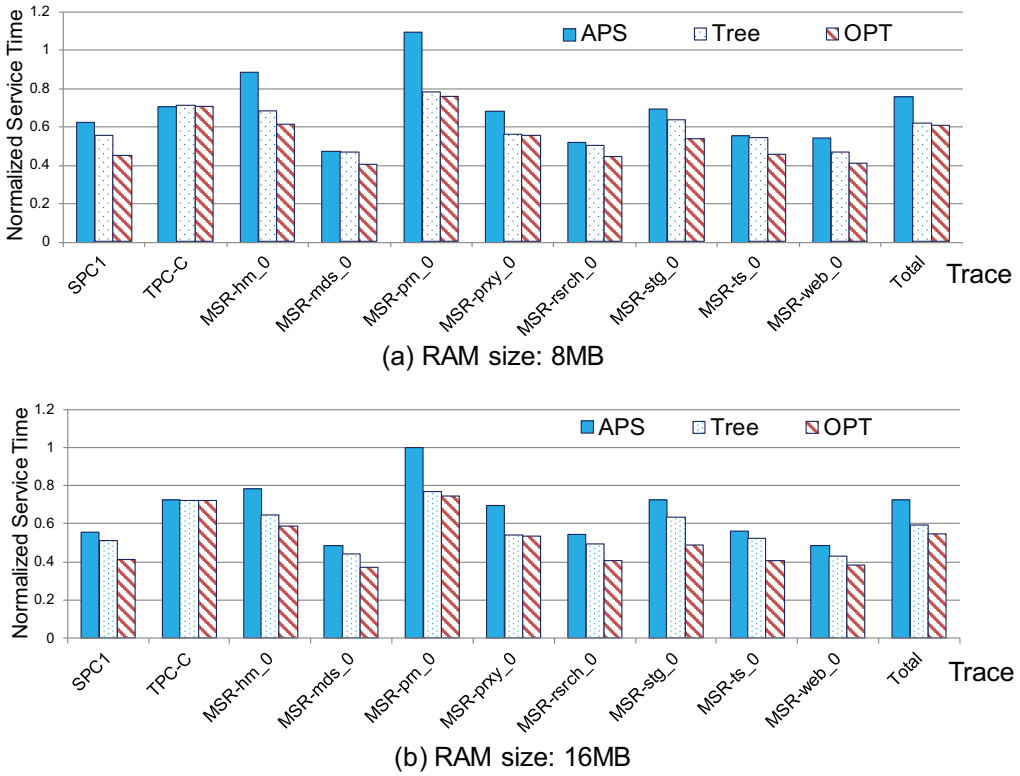

Figure 6.5: Normalized Service Time for Traces (1)

We implemented APS and JTL as comparisons to TreeFTL. Their implementations are referred to as `APS`, `JTL` and `Tree`, respectively. `APS`'s interval length was 1000 requests which is the same as in [94], and the two partitions had equal capacity in the beginning. We also simulated the theoretically optimal algorithm based on TreeFTL. It is the `OPT`; upon victim eviction, `OPT` always picks out the one whose next use will occur farthest in the future. The idea of `OPT` has been proposed for the analysis on the page replacement of virtual memory management of the OS [97]. We include it to show the gap between TreeFTL and the optimal situation. The metric to measure access performance is the *service time* needed to process a trace using a management scheme. For a scheme, the shorter the service time is, the higher its performance.

6.5.2 Performance Improvements by TreeFTL

Figure 6.5 and Figure 6.6 present the results for each trace with the RAM cache configured as 8MB, 16MB, 32MB and 64MB. The rightmost bar in each diagram is the sum of the results of all ten traces. Because service time of ten traces varies over a wide range, we normalized values of `Tree` and `APS` to that of `JTL`. From Figure 6.5 and 6.6 we can see `Tree` always has the least service time under all four configurations, which means it consistently achieves the highest performance. Take the 64MB RAM cache for example. `Tree`'s average time over all traces is less than that of `APS` and `JTL` by 46.7% and 49.0%, respectively. The service time of `Tree` is at best 73.9% and 72.3% less than that of `APS` and `JTL` on MSR-prxy_0, respectively.

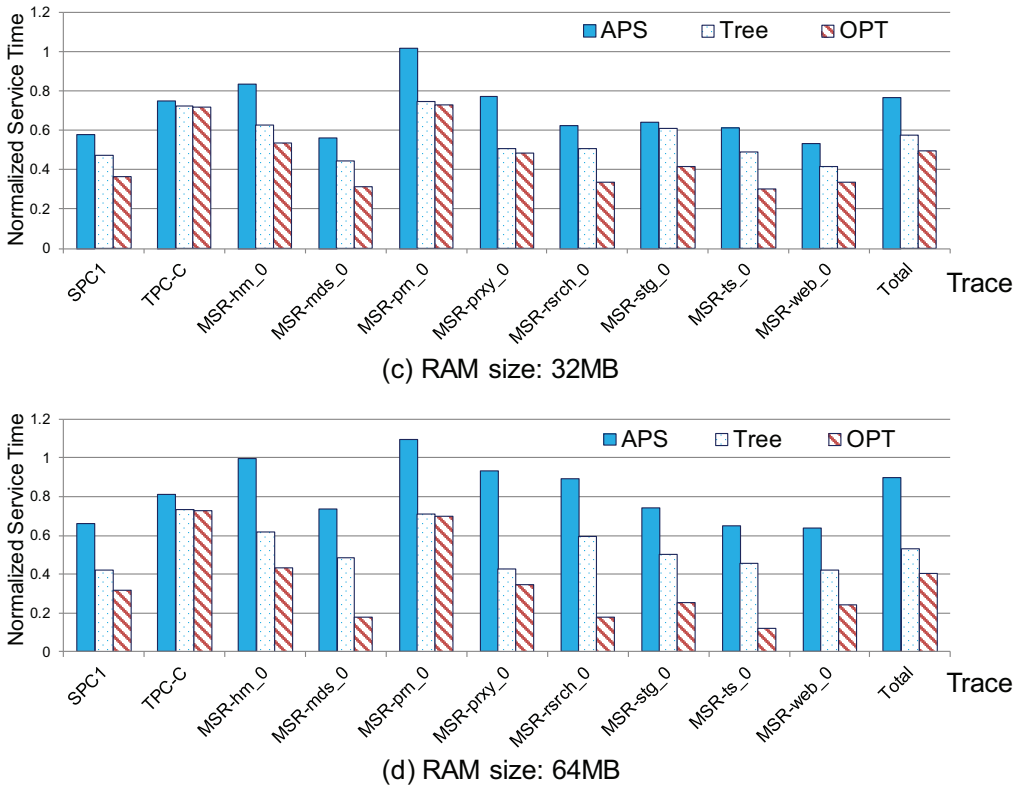

Figure 6.6: Normalized Service Time for Traces (2)

From Figure 6.5 and Figure 6.6 we can see the gap between results of `JTL` (normalized as 1) and those of `Tree` is significant. It is because JTL statically partitions for mapping and buffering into two halves. Evidently, the buffering partition needs to take up more RAM space. One entity in a translation page is only 4 bytes but an entity in the buffering partition is a 2KB data page. This means that the total number of distinct entities held in JTL's mapping partition

far exceeds that in its buffering partition. Misses for data pages cause frequent loading and eviction between RAM and flash, while most of the space dedicated to mapping entries is infrequently used.

From the two figures we can also find that with a small RAM capacity, `Tree` outperforms `APS` marginally. They both can adaptively adjust partitioning. A small capacity cannot effectively cache mapping information or data pages, and evictions and loading dominate the performance. However, owing to the delayed estimation of `APS`, `Tree` is still a little faster. With the increasing of RAM cache, the overhead of LRU selection becomes significant for `APS`.

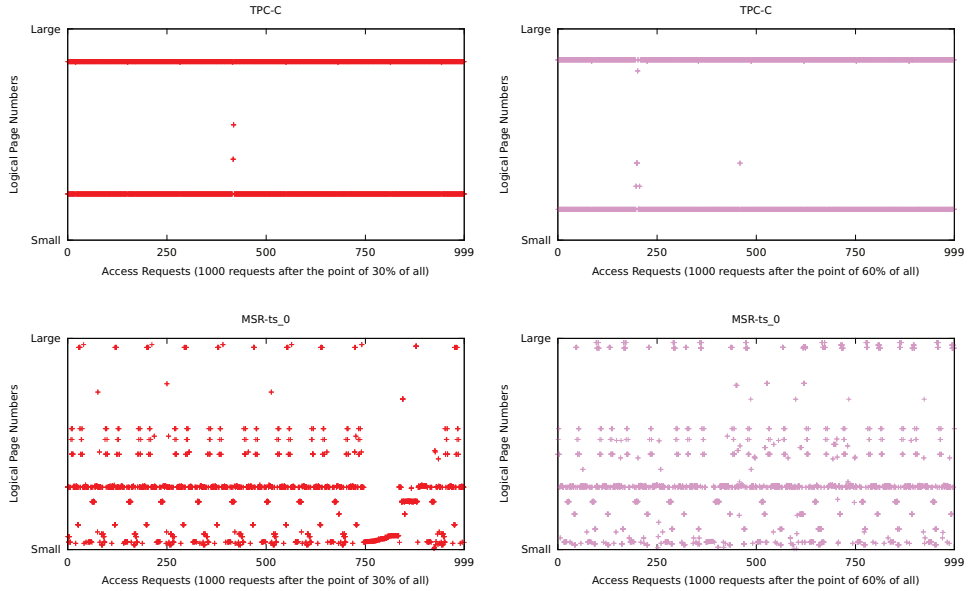

Figure 6.7: Captures of Access Distribution for TPC-C and MSR-ts_0

Also from Figure 6.5 and Figure 6.6 we can look into the gap between `Tree` and `OPT`. Of course `OPT` can spend less time than `Tree` as the former has foreseen the future. The size of the difference, though, depends on each trace. For example, for the trace TPC-C `Tree` and `OPT` achieve the similar performance; `OPT` is just marginally faster than `Tree` by 0.65% with 64MB RAM buffer. However, for the trace MSR-ts_0, which has the most significant difference within 64MB RAM buffer, `OPT` is faster than `Tree` by 72.76%. In order to investigate the cause for variant differences, we have captured two access periods of TPC-C and MSR-ts_0, as shown in Figure 6.7, the same as what we did for ADAPT in Chapter 5. After 30% and 60% of all requests, we fetched one thousand consecutive write requests, respectively. We recorded logical page numbers those requests would access. From the upper two diagrams in Figure 6.7 for TPC-C, we can see the

distribution is very uniform. Most requests are grouped into narrow stripes of logical address space, respectively. The future access hence exactly coincides with the past access. So to evict the LRU one is profitable for TPC-C.. On the contrary, for the lower two diagrams in Figure 6.7 for MSR-ts_0, the distributions are diverse. Accesses are spread in a large range of logical address space, and using the past access to predict the future access seldom sustains. Hence, the LRU victim eviction for MSR-ts_0 is yet imprecise. Even so, the future access can still be roughly reflected from the past access history. As shown in the diagrams of MSR-ts_0, there are consecutive accesses to some logical pages, and the LRU victim eviction is supposed to benefit consequently.

Table 6.2: Hit Ratios (%) of `APS`, `JTL` and `Tree`

Trace	Address Mapping			Data Buffering		
	`APS`	`JTL`	`Tree`	`APS`	`JTL`	`Tree`
SPC1	97.5	97.5	99.8	65.3	24.9	70.5
TPC-C	99.5	97.5	100.0	99.5	99.1	99.5
MSR-hm_0	92.4	94.2	99.4	45.8	18.1	64.1
MSR-mds_0	95.6	98.1	99.7	64.5	31.7	70.3
MSR-prn_0	70.1	96.1	99.6	48.0	26.3	77.9
MSR-prxy_0	98.9	98.6	99.9	53.2	33.9	92.6
MSR-rsrch_0	97.4	98.2	99.5	59.0	34.0	63.3
MSR-stg_0	97.4	98.2	99.6	61.5	23.1	64.8
MSR-ts_0	95.4	97.1	99.6	59.4	21.6	68.2
MSR-web_0	95.1	98.0	99.6	64.6	20.1	75.3

Table 6.2 shows the hit ratios of three schemes for mapping and buffering with the 64MB RAM cache, respectively. `Tree` hardly has any miss for mapping. We ascribe this to the spatial locality of a real workload. `APS`'s and `JTL`'s ratios are a little lower because of their policy of caching single entries. In Table 6.2 generally the hit ratios of buffering are much lower, which is due to the mentioned asymmetry between mapping information and data pages. Yet `JTL` suffers more than `APS` and `Tree` due to its fixed partitioning.

We experimented with the RAM size as 128MB and 256MB also. They are not included as the results are similar to ones with smaller configurations. We did not evaluate with an even bigger RAM cache, as excessive RAM space makes it possible to accommodate everything needed in RAM [35]. This will not correctly highlight the effectiveness of the RAM management schemes.

To further understand the access behaviors of workloads as well as the actions of TreeFTL, we also recorded other meaningful information at runtime.

Figure 7.7 shows the cumulative service time as well as the average size of CG for traces over the increase of write requests. Here the average size of CG is defined as the average number of CDPs per CG at a time. As TreeFTL is specifically designed to handle random write requests, we show the two values for random write requests only. In Figure 7.7 a diagram corresponds to a trace. In a diagram, a dashed line is the cumulative service time for random write requests, and the other solid line presents the fluctuation of average CG size of a specific trace at runtime. From each diagram, on one hand we can see that for a trace the average size of CG fluctuates substantially. The random vibration curve of each diagram is due to the changing access behavior of a workload. As the workload's access behavior is dynamic, TreeFTL has to adjust the tree structure by loading and evicting CDPs and CTPs to adapt. The curve just reflects the situation TreeFTL is facing.

On the other hand, the cumulative service time in each diagram of Figure 7.7 for random write requests shows an interesting phenomenon as for traces it is almost regular. Most of the dashed curves can be approximated as a liner function of random request in a long run. Even though some of them cannot be modeled as one single line, like the scenarios for MSR-prxy_0 and MSR-ts_0, they can still be viewed as piecewise linear functions. We attribute the steadiness of the service time to the adaptivity of TreeFTL. As TreeFTL does not manage the RAM space using heavyweight adjustment mechanism, it can inherently achieve gradual increase of service time at running.

6.5.3 Effect of the Lightweight LRU Selection

The effect of TreeFTL's lightweight victim selection was also measured. We implemented `Tree-PL`, which is the same as `Tree` except that LRU victims are selected at the page-level. Without loss of generality, MSR-hm_0 and MSR-prxy_0 were picked as examples. Figure 6.9 are their results in six cases of "RAM Configuration+Scheme", respectively. It clearly shows the contributions of RAM operations ("RAM"), flash operations ("Flash") and LRU overheads ("LRU") to the overall service time with the RAM cache configured to be 32MB and 64MB using `APS`, `Tree-PL` and `Tree`, respectively. `JTL` differs from `APS` and `Tree` in LRU victim selection, so it was excluded. The results in Figure 6.9 support our claim that as the capacity of the RAM cache increases, LRU selection overhead will be an issue. We can see for `APS` and `Tree-PL`, the overhead of the page-level selection contributes significantly to the worsening of performance as RAM cache scales up. The CG-level LRU selection also suffers from a larger RAM size, but the overhead increases more steadily due to its coarser granularity.

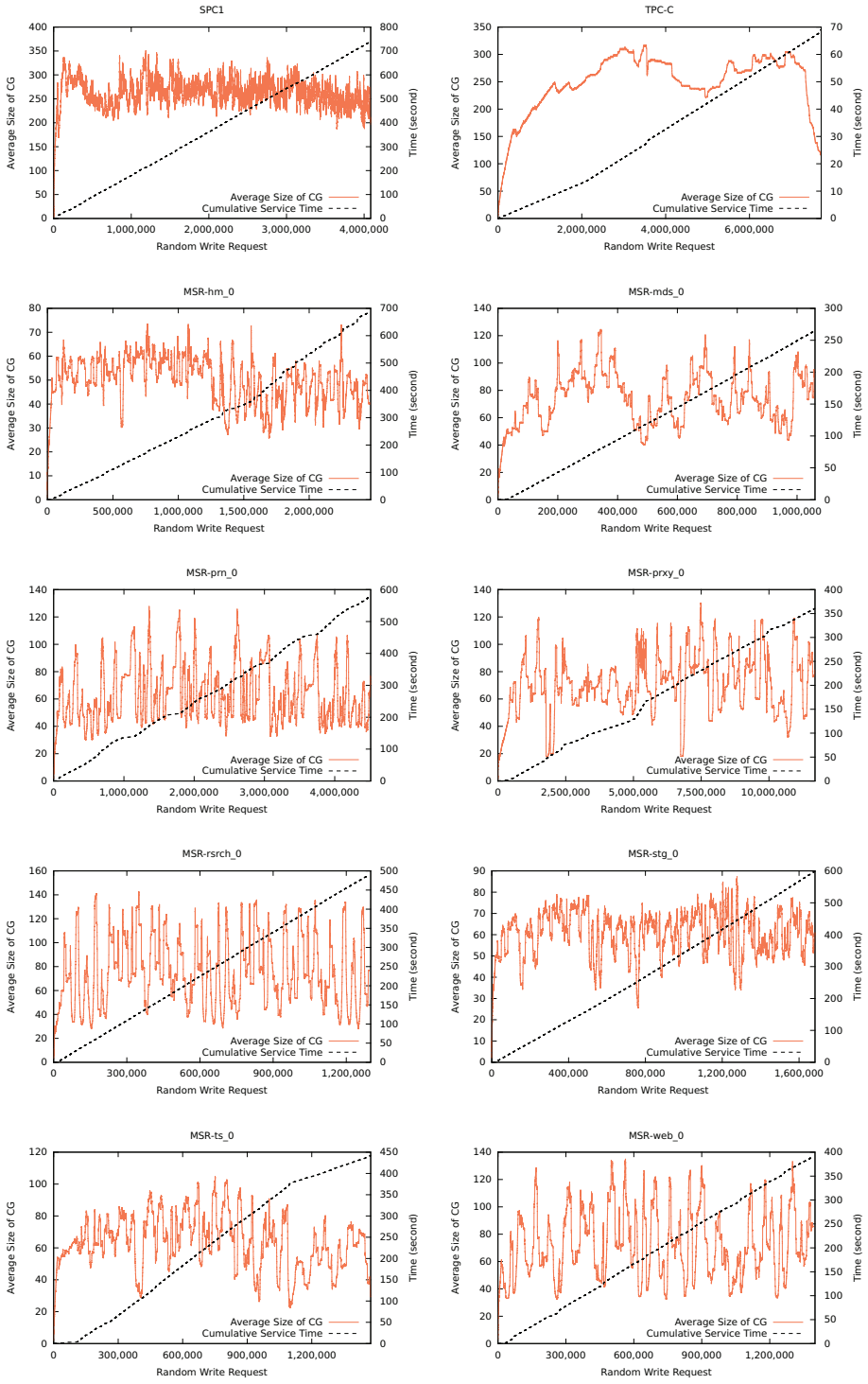

Figure 6.8: Cumulative Service Time and Average Size of CG for Traces at Runtime

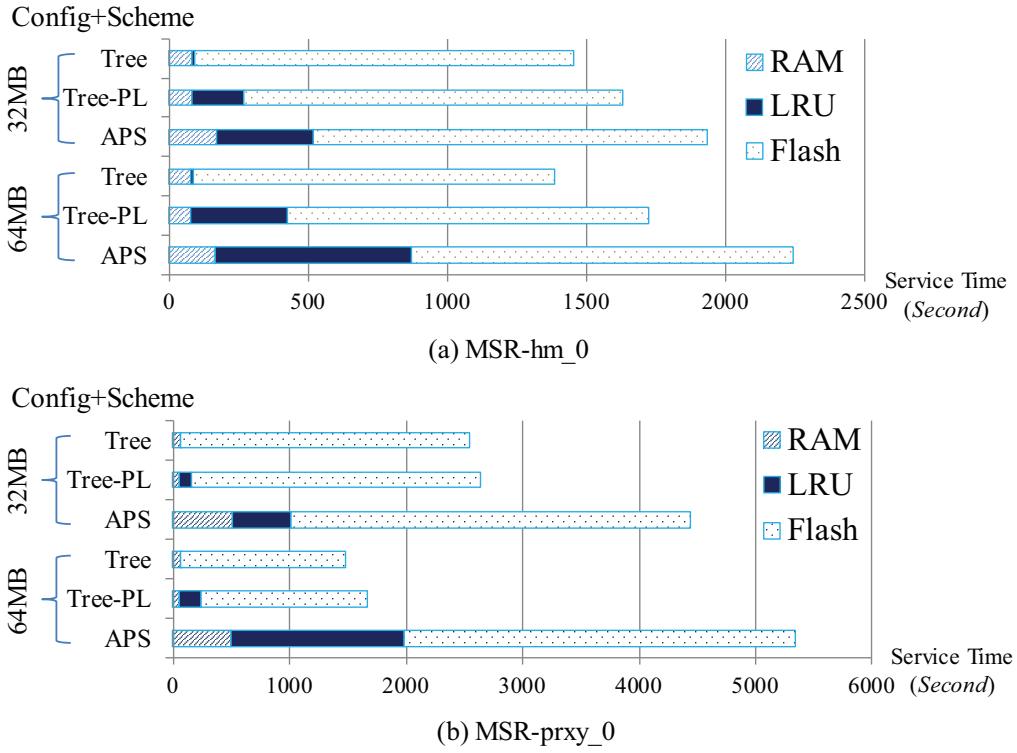

Figure 6.9: Effect of Lightweight Victim Selection

6.6 Summary

The RAM buffer is an important component of a NAND flash-based storage device. It is the one that directly reflects access behaviors of workloads. Managing it efficiently certainly yields substantial performance improvements. TreeFTL proposed in this chapter is capable of utilizing the RAM space jointly for caching information of address translation and buffering data pages. Cached translation pages and data pages are organized in a tree-like structure that naturally adapts to changing workloads. To minimize the overhead of victim evictions, TreeFTL employs a lightweight LRU selection algorithm based on the tree structure. The victim selection is done at a coarse level, but the trade-off in precision results in the significant reduction in processing time. What is more important, the lightweight LRU victim selection enables TreeFTL to adapt to the context switch of workloads with negligible temporal overheads. Experimental results show that TreeFTL is able to outperform previous schemes evidently on various workloads which are from real environments.

Chapter 7

SAW: OS-Assisted Wear Leveling

This chapter will detail what we did in the third step of this thesis. Preceding chapters have proposed three algorithms for flash memory management. Each of them acts as one module of the FTL. As the standalone entity for flash management, the FTL manages the flash device in an autonomous way. This chapter, however, will involve the outside, upper-level operating system (OS) in the management process because the OS has a higher perspective of data and files. The algorithm to be shown is the *operating **S**ystem-**A**ssisted **W**ear leveling* (SAW). As its name suggests, the OS of SAW will assist the FTL to perform wear leveling. In brief, the OS analyzes access behaviors to files, and delivers hints along with data segments to the FTL. On the other side, the FTL references such hints to conduct data accommodation. Experimental results show that the wear evenness can be evidently improved with the participation of the OS. The design of SAW as well as relevant experiments would be detailed.

7.1 Overview

OWL, ADAPT and TreeFTL proposed in previous chapters target different aspects of NAND flash memory management. In general all of them have been designed in the traditional way. That is to say, the management is performed in the full charge of the FTL inside a flash device; the outside, including the OS, has no idea of what is happening in the device. Certainly it is unnecessary for the OS to look into the storage device; the storage device is supposed to be transparent to the OS. What the OS wants is the data. The OS writes data to the device and reads data from the device. It is not the OS's concern how the

data are subsequently processed by an individual storage device.

With the increasing utilization of flash-based device, some mechanisms have been introduced into the OS to suit. A well-known example is the TRIM command [21]. The TRIM command is used by the OS to inform the FTL in advance of reclaiming space occupied by deleted data. The command itself is simple, but its effectiveness is evident. More complicated schemes that seek aid from the OS, like FSAF [75], MFTL [115], and Meta-Cure [108], have been proposed. As is analyzed in Chapter 3, they just ask or even passively wait for information from the OS; either the OS is unaware of FTL's workings, or vice versa. The interplay between the OS and the FTL, if existing, is lacking.

The profits brought in by the involvement of the OS are not limited. Our proposal in this chapter, the said SAW, however, will explore a deep collaboration between the OS and the FTL. Why we declare it to be **deep** is that the OS knows the device it is connecting to is a flash device and the OS itself participates in the process of wear leveling. The key point here is the OS's knowledge of data and files. For a flash device or other storage devices, the data is data. In the opinion of the OS, the data is not just data. They are first presented to the OS in the form of files. The OS creates, opens and operates files for various applications. As has been investigated, the properties of files are meaningful even with basic attributes of files [24, 73]. On the other hand, the knowledge of files is out of reach of the FTL, and the OS yet can provide such information. For SAW, what we let the OS do for wear leveling is quantitatively analyze files to figure out their access patterns. Based on the analysis, each *file type* will be assigned an integer number which we reference as the *temperature* degree. What the temperature is and how to measure the temperature degrees for files would be shown below. As each file may have more than one data segment, the temperature degree will be attached to each segment of a file and delivered to the lower-level FTL alongside. When the FTL receives a data segment with a degree, it will unpack the segment, read the degree and allocate suitable aged blocks accordingly.

Obviously the above process of SAW is founded on the analysis of files. The analysis is performed by the OS for SAW, based on a mathematical model. It is succinct but helpful. Not only the current access frequencies of files are considered but the historical impacts are also taken into account. Details of the analytic model can be found in the next section. Correspondingly, the lower-level FTL also makes a change on the block organization to suit itself to utilize the temperature degrees provided by the upper-level OS. Flash blocks are organized in an exponential division through which blocks of less erase counts are given

more chance to be reused and elder blocks are likely to be protected.

The advantage of SAW over traditional algorithms for wear leveling is evident. Traditional wear leveling is conducted by the FTL itself. The lower-level viewpoint and limited computation of a flash device adversely impact its function. For example, though OWL can also figure out access frequencies of data inside a flash device, the relevant temporal and spatial overhead are not ignorable. For SAW, the OS's participation just gives the FTL extra knowledge, and yet releases the FTL from arduous calculation over data within an embedded device.

The improvement of wear evenness attained by SAW is substantial. To verify the effectiveness of SAW, we have developed a prototype based Linux virtual file system (VFS), an open-source FTL and file system. From the experimental evaluation based on the prototype, thanks to the participation of the OS, the wear evenness can be significantly improved by as much as 85.0%.

7.2 Temperature of File Types

In this section we will present details of how the OS collaborates with the FTL for wear leveling in SAW. First we need a mathematical model to figure out the temperature degree of files. Note that the OS processes files instead of data, which differs from the FTL. For a traditional wear leveling algorithm, data are either coarsely identified to be hot or cold, or arduously classified by the FTL within the limited computation resource of a flash device [32, 58, 22], as has been shown in Chapter 4. It is where SAW is about to make a change. In SAW, the OS is responsible for quantitatively classifying files based on a mathematical model. For a file type, the OS dictates and analyzes its files' update behavior. Not only the current update behavior is pondered, but the file type's historical record is also taken into consideration. When each file type gets an estimate for its update behavior, behaviors of all file types are assessed together. The temperature degree of each file type is accordingly calculated. The degree would be sent along with data segments of files to the FTL for reference. This process briefly expounds how the OS plays its rule in the collaboration with the FTL.

We first need to figure out how to estimate the update behavior given a specific file type. The term of **file type** has been highlighted above. What a type files belong to depends on their file attributes. Files have a number of attributes, such as filename, extension, access mode, and last modified time. Mesnier et al. [73] revealed that files' properties, like the access pattern, can be predicted based on their attributes. Take a text file for example. It is likely to be

rewritten more often than a video file. Mesnier et. al. [73] did an offline mining over collected files. An online exploration of the files' attributes, however, is not simple. Our analytic model must be succinct and reasonable to support the online use.

For simplification, SAW only considers two attributes of a file, namely its filename extension and access mode. Read-only files are hardly rewritten, and will be specially dealt with. In this thesis, a file's type refers to its filename extension, although it is conceivable that other attributes can be used too. Files without any extension will be treated separately. Previous qualitative ways to identify data to be hot or cold by the FTLs are somewhat lacking. With the assistance of the OS, we will perform the classification in a quantitative way. The temperature of a file type, as will be derived below, depends on files' update *frequency* and *recency*.

7.2.1 Update Frequency of A File Type

Measuring the update frequency of a file type is the key issue of SAW. FTLs can record the number of writes to a logical block or a logical page [105]. For files, however, it is not that straightforward. The OS manages a large number of files of the same type. It is neither reasonable nor scalable to keep access information for each individual file. Moreover, two files with the same type may have completely different update frequencies; how to merge files' frequencies to stand for the behavior of a file type is a problem. Hence, we need an approximation to represent the access frequency for a **type** of files. Since not all files are accessed at runtime, we will not consider *dormant* files but focus only on *active* ones. This simplifies the online analysis, and also reduces the overhead of resuming SAW at boot-up.

Table 7.1 is a collection for a quick reference of variables used during the analytic modelling. Several of them are maintained by SAW for an individual file type. Given a file type t, $S_{(t)}$ records the total number of active files of type t. $\varsigma_{(t)}$ is the number of accessed files of type t. This includes the files of type t that have been opened (and possibly then closed) after the current system boot, as well as newly created files. $\delta_{(t)}$ is the number of files of type t that have been deleted (since the last boot). $\omega_{(t)}$ counts the *rewrites* to all t files. $\varsigma_{(t)}$, $\delta_{(t)}$ and $\omega_{(t)}$ are used to compute the update frequencies of file type t. Note that we are interested in rewrites detected in the kernel module of file system, not writes, because the latter is not a good estimate for update frequency. For example, a video file triggers a vast number of writes during its creation. Afterwards its contents are hardly rewritten again. So a video file's update frequency is low.

Table 7.1: Symbols of SAW Model

Symbol	Description
t	File type
$S_{(t)}$	Total number of active files of type t
$\varsigma_{(t)}$	The number of accessed files of type t
$\delta_{(t)}$	The number of deleted files of type t
$\omega_{(t)}$	The counts of rewrites to files of type t
$\varphi_{(t)}$	The update frequency of type t
I	The interval
$f^n_{(t)}$	The predicted value for $\varphi^n_{(t)}$ of the nth interval
$s^n_{(t)}$	The rate of increase of type t files of the nth interval
β	The bound to outlier file types
Θ	The number of active file types
T	A predefined constant as the upper bound of temperature degree
$C^{n+1}_{(t)}$	The temperature for type t in $(n+1)$th interval
Γ	The number of free flash blocks

When a text file is reopened, however, it may be inserted, appended or replaced with new data. Thus, its update frequency is much higher due to many rewrites.

Let $\varphi_{(t)}$, the *update frequency* of type t, be defined as

$$\varphi_{(t)} = \frac{\omega_{(t)}}{S_{(t)}}. \tag{7.1}$$

At the first sight, $\varphi_{(t)}$ seems to be the average rewrite of active files of type t. Nonetheless, as is mentioned, files of the same type may differ significantly in update behavior. Moreover, files are being created and deleted at runtime. So Equation (7.1) is imprecise. But it is infeasible to keep too much information for each file. We shall place more constraints to enhance the accuracy of Equation (7.1).

First, the OS will collect the values of ς, δ and ω **periodically**. The interval is defined as I. The total number of active files of type t after the nth I is to be $S^n_{(t)}$. The base case, i.e., at boot-up, is defined as $S^0_{(t)}$, and initialized to be zero. In the nth I interval, $\varsigma^n_{(t)}$ files were newly accessed or created, and $\delta^n_{(t)}$ files were removed. So the number of type t files before the start of the $(n+1)$th I is

$$S^{n+1}_{(t)} = S^n_{(t)} + (\varsigma^n_{(t)} - \delta^n_{(t)}). \tag{7.2}$$

Hence, the absolute increment of type t files is

$$S^{n+1}_{(t)} - S^{n}_{(t)} = \varsigma^{n}_{(t)} - \delta^{n}_{(t)}. \tag{7.3}$$

The rate of increase of type t files, $s^{n}_{(t)}$, of the nth I, is

$$s^{n}_{(t)} = \frac{\varsigma^{n}_{(t)} - \delta^{n}_{(t)}}{S^{n}_{(t)}}, \tag{7.4}$$

where $n \geq 1$ because at boot-up $S^{0}_{(t)} = 0$, and it is in the first I that files are accessed or created.

$s^{n}_{(t)}$ could be positive or negative, as the number of type t files may increase or decrease. β is a bound such that

$$\text{-}\beta \leq s^{n}_{(t)} \leq \beta, \tag{7.5}$$

or put in another way,

$$S^{n+1}_{(t)} = S^{n}_{(t)} \cdot (1 \pm \beta), \tag{7.6}$$

and Equation (7.1) is hence valid for the calculation of temperature. In this thesis, β is set to be 10%. We do not expect the number of active files of type t changes sharply. If $|s^{n+1}_{(t)}| > \beta$, we will identify t to be an *outlier*. An outlier deserves special attention since many t files are likely to be created or removed in a short period of time.

7.2.2 Update Recency

After the nth I, Equation (7.1) can be rewritten as

$$\varphi^{n}_{(t)} = \frac{\omega^{n}_{(t)}}{S^{n}_{(t)}}. \tag{7.7}$$

Equation (7.7) gives the rewrite frequency on a file type t, and it estimates the update behavior of type t files in the $(n+1)$th interval. However, $S^{n}_{(t)}$ accumulates the number of active files during past n intervals. As time goes by, the updates of type t may change a lot due to the context switch of applications. Hence a value from a long time ago may mislead the estimation. Generally, the most recent intervals are more relevant to the coming interval, and this *recency* should be factored into Equation (7.7).

We introduce another variable to improve Equation (7.7), $f^{n}_{(t)}$, which is defined to be the predicted value for $\varphi^{n}_{(t)}$ of the nth interval. Here we will revisit the

idea of *exponential moving average* again, which has been referenced in Chapter 5. We define an exponential moving average of $f_{(t)}^{n+1}$ for the $(n+1)$th I as

$$f_{(t)}^{n+1} = \alpha \cdot \varphi_{(t)}^{n} + (1-\alpha) \cdot f_{(t)}^{n}, \tag{7.8}$$

in which $0 \leq \alpha \leq 1$. When $\alpha = 0$, the recent interval will have no effect. With $\alpha = 1$, the past history is assumed to have no influence. Given an α that $0 < \alpha < 1$, we have

$$\begin{aligned} f_{(t)}^{n+1} &= \alpha \cdot \varphi_{(t)}^{n} + (1-\alpha) \cdot f_{(t)}^{n} \\ &= \alpha \cdot \varphi_{(t)}^{n} + (1-\alpha) \cdot \left[\alpha \cdot \varphi_{(t)}^{n-1} + (1-\alpha) \cdot f_{(t)}^{n-1}\right] \\ &= ... \\ &= \alpha \cdot \varphi_{(t)}^{n} + (1-\alpha) \cdot \alpha \cdot \varphi_{(t)}^{n-1} + ... \\ &\quad + (1-\alpha)^{i} \cdot \alpha \cdot \varphi_{(t)}^{n-i} + (1-\alpha)^{(i+1)} \cdot \alpha \cdot \varphi_{(t)}^{n-(i+1)} \\ &\quad + ... + (1-\alpha)^{n+1} \cdot f_{(t)}^{0}. \end{aligned} \tag{7.9}$$

Because

$$\alpha > (1-\alpha) \cdot \alpha > ... > (1-\alpha)^{i} \cdot \alpha > ... > (1-\alpha)^{n} \cdot \alpha, \tag{7.10}$$

we can conclude for $f_{(t)}^{n+1}$, the farther an interval is, the less the effect it has ($f_{(t)}^{0} = 0$ and $\varphi_{(t)}^{n} = 0$, so the last $(1-\alpha)^{n+1}$ is ignorable). In other words, $f_{(t)}^{n+1}$ depends the most on $\varphi_{(t)}^{n}$, and also takes the past history into consideration when $0 < \alpha < 1$. Now we can use $f_{(t)}^{n+1}$ to predict the future update behavior to files of type t in the $(n+1)$th interval.

7.2.3 Temperature of File Types

Now that we have f^{n} for all file types, we can compute their temperature before each interval. The temperature degree used in this thesis is from 0 to T. T is a predefined constant. A file with the zero degree is very cold, effectively like a read-only file. If a file's temperature is near to T, it is extremely hot. Given a set of file types, each one with an f^{n+1} for the $(n+1)$th interval, we sort them by their f values in an ascending order. The type t then has a *position number* in the sequence, $P_{(t)}^{n+1}$ for the $(n+1)$ interval, where $0 \leq P_{(t)}^{n+1} \leq \Theta - 1$. Θ is the number of active file types. For example, there are five file types (i.e., $\Theta = 5$), and the sorting sequence is

$$f_{(t_0)}^{n+1} \leq f_{(t_1)}^{n+1} \leq f_{(t_2)}^{n+1} \leq f_{(t_3)}^{n+1} \leq f_{(t_4)}^{n+1}.$$

So $P_{(t_0)}^{n+1} = 0$ and $P_{(t_3)}^{n+1} = 3$. Then we can calculate the temperature for type t, $C_{(t)}^{n+1}$, using

$$C_{(t)}^{n+1} = \frac{P_{(t)}^{n+1}}{\Theta} \cdot T. \tag{7.11}$$

If T is set to be 5, $C_{(t_3)}^{n+1} = 3$ for type t_3. Note that Equation (7.11) is valid when $n \geq 1$. The temperature of each type t for the first interval, i.e., $C_{(t)}^{1}$, is initialized to be zero. Here the absolute value of $f_{(t)}^{n+1}$ for file type t is not so important as we just need to compare f^{n+1} of file types. It in turn means the absolute value of α has no impact to the temperature degrees of file types.

It may seem tedious to have to perform a sort over Θ file types after each interval. As has been investigated in Section 4.4, SAW can apply the lightweight method of OWL, and the sorting can be avoided accordingly. Another feasible alternative way is that, since the access behaviors of the majority of file types are stable, a complete re-sorting is not yet necessary. Instead, SAW scans the previous sequence with updated f values, performing the necessary reordering. This is fairly inexpensive as well.

According to Equation (7.11), it is not possible for a file to have a temperature of T, as T is reserved for outlier files.

7.3 Wear Leveling with Temperature

7.3.1 Exponential Division of Flash Blocks

We use the temperature of a file type to allocate blocks and pages to its data. First, we need a hash table to maintain the temperature degrees for file types. The hash key is a file type t which is hashed to its $f_{(t)}^{n}$ for the nth interval. This table is managed by the OS in the main memory, not in flash devices.

The primary idea of wear leveling is to allocate young blocks to hot data, and old blocks to cold data. To make use of the temperature, free blocks in a flash device should be well organized. SAW has its own customized block organization. It sorts flash blocks in an ascending order by their erase counts. As we have T degrees, all free blocks are divided into T groups. The division is not equal but in an *exponential* way. Assuming there are Γ sorted free blocks, the first group has $\Gamma/2$ blocks that have the smallest erase counts. The second group has $\Gamma/2^2$ blocks. By analogy, the gth group has $\Gamma/2^g$ $(0 \leq g \leq T-1)$ blocks. The Tth group, however, is an exceptional one that keeps $\Gamma/2^{(T-1)}$ blocks that are the most worn at that time. This is because

$$\Gamma = (\frac{\Gamma}{2^1} + \frac{\Gamma}{2^2} + \frac{\Gamma}{2^3} + ... + \frac{\Gamma}{2^g} + ... + \frac{\Gamma}{2^{(T-1)}}) + \frac{\Gamma}{2^{(T-1)}}. \tag{7.12}$$

In SAW, an allocation request with a temperature degree of d is satisfied by the $(T - d)$th block group. Whether to allocate a page or a block depends on the allocation policy of the FTL. Why the mapping relationship between temperature and block groups is in a $\{d : (T - d)\}$ way is evident. For data with larger degrees, they are frequently updated, so they consume much more blocks. Correspondingly a larger block group with less erase counts are prepared for them. For data with smaller degrees, as they are infrequently updated, they unusually raise much less allocation requests. Another explanation can be given from the standpoint of blocks. The exponential division is due to the intention to make the best use of young blocks that are the least worn. SAW maintains more blocks to the higher temperatures. Ones with the smallest erase counts are given more chance to be utilized, while elder block can avoid being frequently erased.

As is mentioned, SAW specially treats read-only and outlier files. The former corresponds to the Tth group, and the latter will be handled with pages and blocks from the first group. There are usually not that many read-only files, so $\Gamma/2^{T-1}$ blocks should be sufficient. Outlier files are quite active. They are accommodated into the youngest $\Gamma/2$ blocks.

7.3.2 Temperature Adjustment

The temperature is re-calculated in every interval. Hence, cold data would lag behind with outdated temperature degrees since they are infrequently updated. Their temperatures should be adjusted. To look for such cold data is not easy. SAW will not yet do it by itself. As is mentioned, there is a module called the garbage collection of flash memory management to clean up obsolete data that are generated due to out-of-place updating. Cold data are left with them. SAW works alongside when the process of garbage collection are being conducted. At that time, SAW checks data to be moved; invalid data are just ignored while the temperature degrees of valid data would be changed. Data with updated temperature degrees are written back by the module of garbage collection then. In this way, the overheads introduced by updating temperature degrees are minimized.

7.4 A Prototype of SAW

We have developed a prototype of SAW based on Linux virtual file system, UBIFS and UBI [36]. As mentioned, there are two types of flash device. One is block-interface flash device, like SSDs. They are emulated to be block devices to file systems by the FTLs. On equipment such as smartphones, raw flash may be

used. UBIFS is designed for the latter, and UBI can be viewed as its special FTL. Unlike commercial file systems and FTLs, UBIFS and UBI are open-source, so they are good candidates to implement SAW.

UBIFS is a log-structure file system. UBI serves UBIFS to access data and performs functionalities of flash management. Several features of UBI and UBIFS facilitate the implementation of SAW. First, UBIFS roughly classifies data to be LONGTERM, SHORTTERM and UNKNOWN. For example, all files' data are hot, i.e., SHORTTERM. Second, data are encapsulated by UBIFS in a *node* with information like the inode number that they belong to [36]. Note that the coarse identification of data is not embedded into nodes. Though, the node structure makes it possible to add our temperature degree into each node. Third, their original wear leveling and garbage collection are not complicated and can be easily replaced or enhanced.

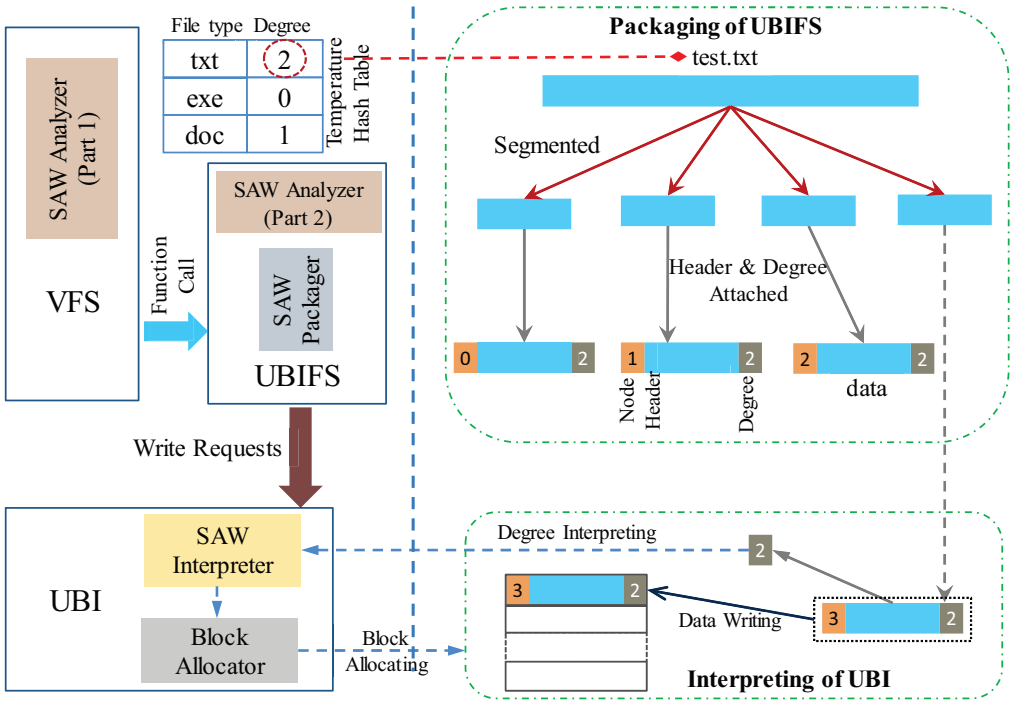

Figure 7.1: A Sketch of SAW Prototype

The prototype of SAW has three components, as is shown in Figure 7.1. The *SAW analyzer* is implemented in the Linux VFS and UBIFS. It maintains the hash table and performs SAW calculations. The *SAW packager* is in UBIFS. It packages data along with relevant temperature degrees into a node. The *SAW interpreter* of UBI supports block allocations using the temperature. Figure 7.1 also gives a sample on text file "test.txt". The temperature degree of "txt" is 2.

The file is segmented into four parts, each packed with the temperature. When a node arrives in UBI, SAW interpreter will suggest to the allocator what would be a suitable age for the block to be allocated. The temperate degree would be written to flash memory along with data. Note that in the real implementation the temperate is inside the header. Here we separate it out for the ease of discussion. For the same reason, the writing sequence of the nodes does not adhere strictly to their header numbers.

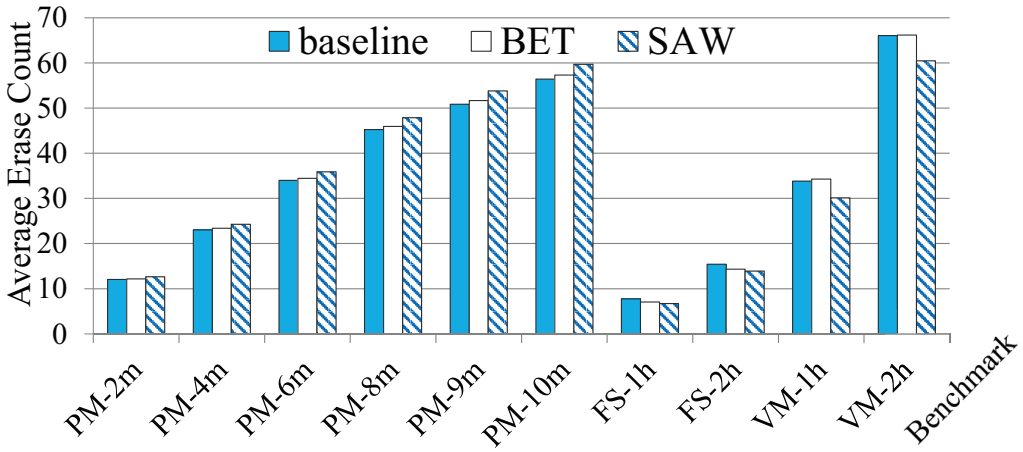

Figure 7.2: Average Erase Count with Prototype

7.5 Experimental Evaluation

The evaluation of SAW was done in two ways. The first is within the above prototype. We compiled the Linux kernel 3.1.6 in Ubuntu 12.04.1. A flash device of 1GB was simulated using the nandsim simulator of Linux kernel. BET was implemented for comparison. The second way we evaluated SAW was with the FlashSim simulator [53], in which we implemented OWL, BET, lazy wear leveling and SAW. The simulated flash was also 1GB. We went on further to enhance BET and lazy wear leveling with the basic idea of SAW on block allocation.

The reason why we did experiments in two ways is that OWL and lazy wear leveling work within hybrid address mapping [105, 10], so they cannot be implemented in UBI. BET does not have such a limitation [14, 105]. The NAND flash in the simulation was configured according to a recent datasheet [41]. The wear evenness is measured using the average erase count and its standard deviation over all flash blocks [105, 104]. For similar average erase counts, the smaller the standard deviation is, the better the wear evenness is. The optimal wear evenness is that no different exists on the erase count of flash blocks. In

other words, the standard deviation is zero, which is impossible.

We did not find any file system benchmarks that target the write endurance of flash memory. What we want are ones that operate on a large number of files and generate sufficient write requests. We examined the analysis of Traeger et al. [102] on various benchmarks for file system, and selected two macro-benchmarks: `postmark` [48] and `filebench` [100]. `Postmark` is single-thread, while `filebench` can be multi-thread. However, they both name file in sequential numbers without any extension. We modified them in order to append a suffix to each file in the form of ".ϵ". ϵ is a lower-case English letter from 'a' to 'z' randomly picked for a file.

The parameters of SAW are set as follows. $T = 10$ and $\alpha = 0.5$. I is relatively measured in terms of write requests. Its default length is 10,000 write requests.

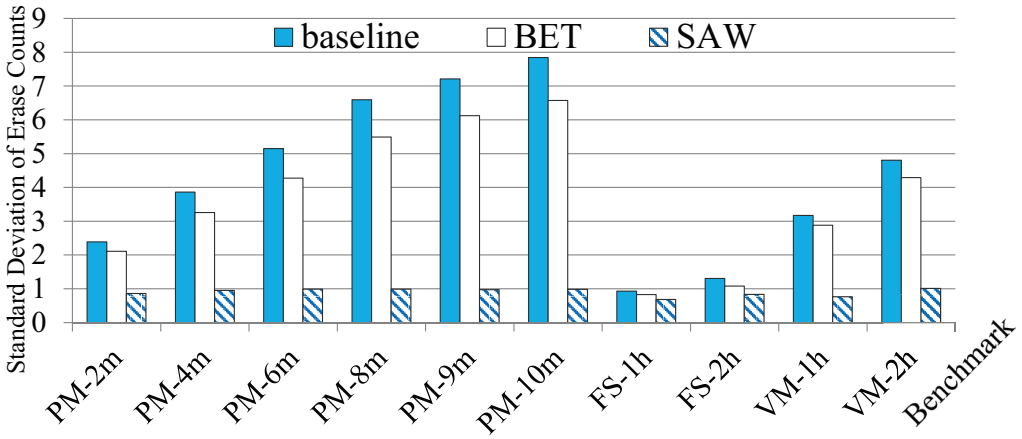

Figure 7.3: Standard Deviation of Erase Counts with Prototype

7.5.1 The Effectiveness of SAW

Figure 7.2 and 7.3 show the average erase count and standard deviation of `baseline`, `BET` and `SAW` with the prototype. `baseline` has the original wear leveling of UBIFS and UBI. `BET` and `SAW` refer to implementations of BET and SAW, respectively. We ran `postmark` with ten settings, from 1 million to 10 million transactions. The number of simultaneous files was 50,000. We selectively present the results of 2, 4, 6, 8, 9 and 10 million transactions, and they are referred to as PM-2m, PM-4m and so on. We ran `filebench` with two public workloads: `fileserver` and `varmail`. For each workload we ran for an hour and two hours, respectively. They are referred to as FS-1h, FS-2h, VM-1h and VM-2h. The number of files was also set to be 50,000. As both `postmark` and `filebench` have random behaviors at runtime [48, 102], we ran our experiments

with each setting thrice, and the results shown in Figure 7.2 and 7.3 are the mean values. Full results are in the next subsection.

The effectiveness of SAW is evident. From Figure 7.2 we can see that in each case `SAW` performed a similar number of erasures compared to `baseline` and `BET`. However, in Figure 7.3, `SAW`'s standard deviation of erase counts significantly decreases compared to `BET`, as much as 85.0% with PM-10m. Even with FS-1h and FS-2h that are read-dominant workloads, the reductions can reach 17.3% and 22.8% compared to `BET`, respectively. Hence we conclude that SAW effectively avoids wear skewness with the cooperation of the OS.

Measuring the performance overheads is not straightforward with the involvement of the OS. Moreover, the changing behaviors of `postmark` and `filebench` during each run make direct comparison difficult. We have recorded counts of write, read and erase operations for each case as indicators of the performance.

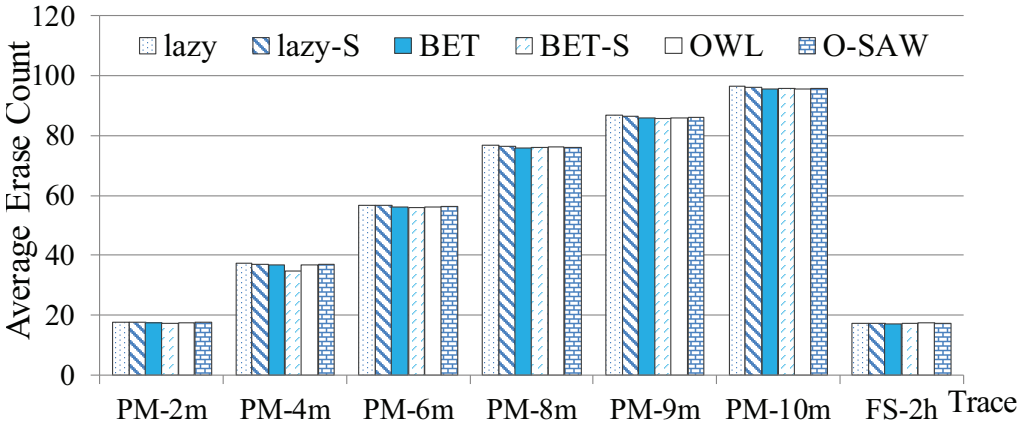

Figure 7.4: Average Erase Count with FlashSim

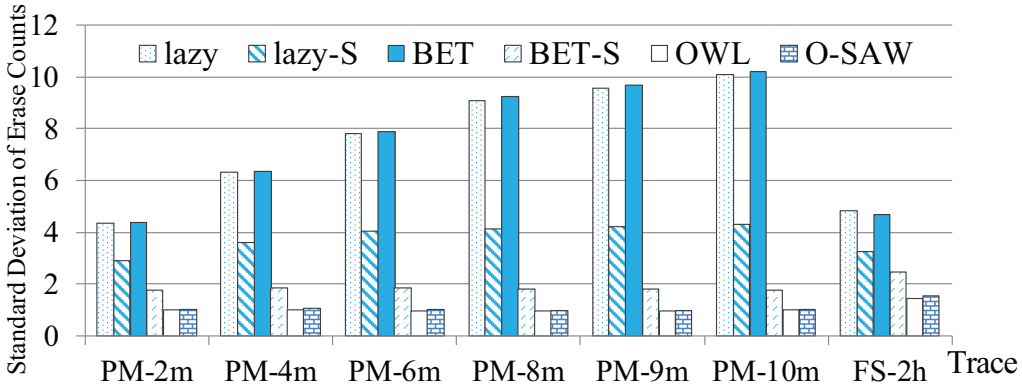

Figure 7.5: Standard Deviation of Erase Counts with FlashSim

OWL, lazy wear leveling, and BET were implemented in FlashSim. The latter two were enhanced with SAW's idea in their block allocation. Their im-

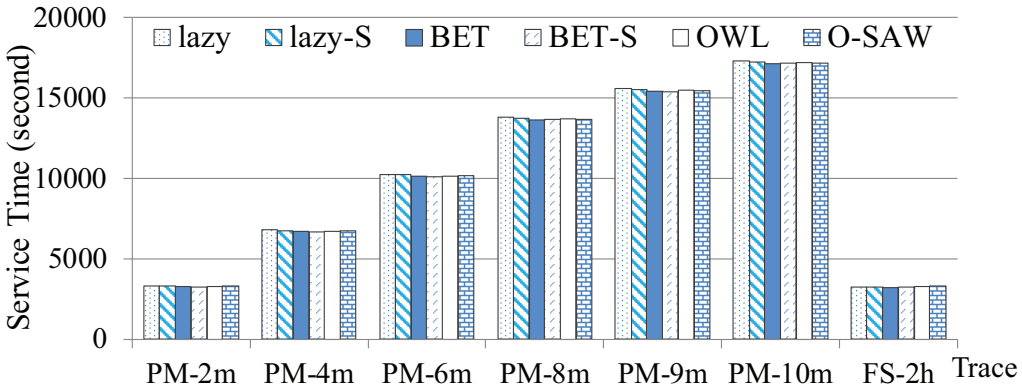

Figure 7.6: Service Time with FlashSim

plementation are referred to as `OWL`, `lazy`, `BET`, `lazy-S` and `BET-S`, respectively. OWL has already considered block allocation. We did not enhance it. Instead, we replaced OWL's block allocation with SAW's. This implementation is referred to as `O-SAW`.

Note that FlashSim is a trace-driven simulator. Previous experiments on FlashSim utilized traces collected from various machines. However, since write requests of those traces have no temperature information, they are not suitable. Instead, we recorded access request in UBI. There, each request does have a temperature. These traces were then fed to FlashSim. Experimental results are partially shown in Figure 7.4, 7.5 and 7.6. Full results are included in the next subsection.

Figure 7.4 and 7.5 show that the average erase counts on a trace for each scheme is similar, but the standard deviation has decreased significantly for `lazy-S` and `BET-S`, by as much as 55.9% and 82.6%, respectively. Thus, wear evenness was highly improved in the presence of SAW. On the other side, `O-SAW` has comparable wear evenness to `OWL`, and the standard deviation of the former is at most 7.0% more. But `OWL` allocates blocks according to its own calculation utilizing the lower computation capability of a flash device, while `O-SAW` just needs to use the temperature of each incoming request. Hence, `O-SAW` has a much lower computation and resource overhead, while achieving a similar level of wear evenness.

The performance overhead can be measured using trace driven simulation because it is entirely deterministic. The time needed to service all requests of a trace is a good indicator of the performance overhead incurred by wear leveling [105, 104]. The more the service time, the greater the performance degradation. Figure 7.6 shows the service time for traces with each scheme. It is obvious that the addition of SAW has little performance impact.

7.5.2 The Accuracy of f for φ

f is used to predict φ for the next interval using Equation (7.8), which is the basis of the temperature calculation. We ran experiments to verify the accuracy of f for φ. Without loss of generality, we selected the file type whose filename extension is "*.c*". We collected f and φ in every interval with PM-5m, PM-10m, FS-2h and VM-2h, and calculated f/φ, as is shown in Figure 7.7. We can see after system boot-up, f/φ fluctuates within tight bounds around 1.0. Hence, we conclude that the prediction of f for φ is accurate.

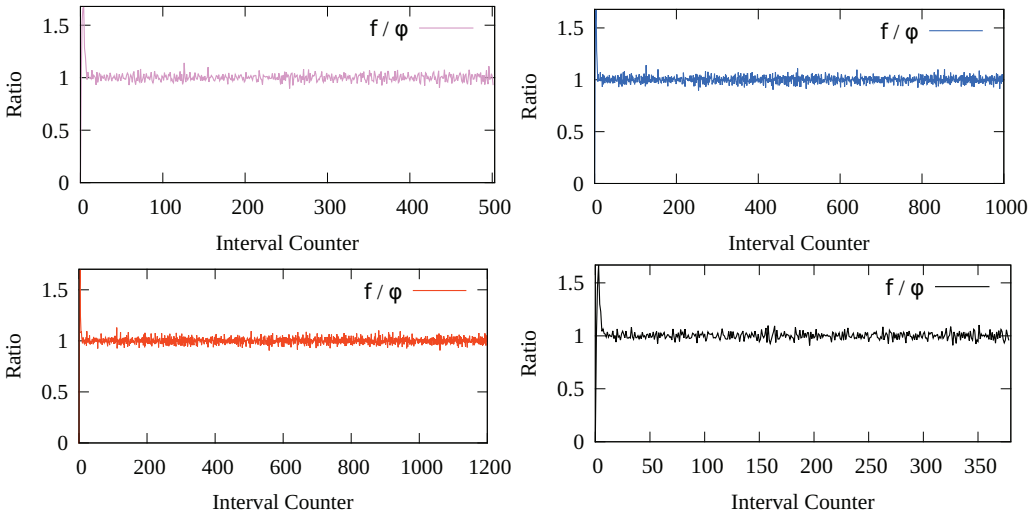

Figure 7.7: Fluctuation of f/φ (Clockwise: PM-5m, PM-10m, FS-2h, VM-2h)

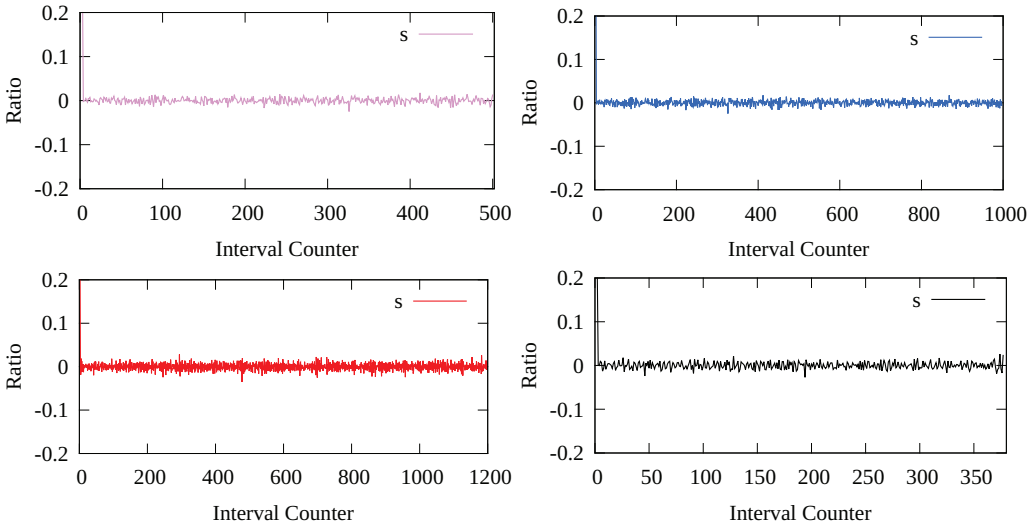

Figure 7.8: s and β at Runtime (Clockwise: PM-5m, PM-10m, FS-2h, VM-2h)

7.5.3 The Impact of β

It was mentioned in Section 7.2.1 that we use β as a bound for identifying whether a file type is an outlier or not. Without loss of generality, we collected the s value in every interval for the file type "*.c*" using the experimental configurations PM-5m, PM-10m, FS-2h and VM-2h. The results are presented in Figure 7.8. At boot-up, many files are accessed, so s is somewhat large. After the system has warmed up, s fluctuates marginally around 0. Note that β was set to be 10% by default. In summary, experiments show that s is typically much less than the selected threshold.

From the experiments conducted to verify the accuracy of f/φ and β we can see our model for the calculation of temperature degrees is quite precise.

Table 7.2: Mean Difference of Standard Deviation with Five Intervals (I)

Benchmark	Mean	Absolute Mean Difference				
		`5k`	`10k`	`15k`	`20k`	`25k`
PM_2m	0.865	0.009	0.001	0.001	0.002	0.013
PM_4m	0.960	0.018	0.004	0.014	0.007	0.007
PM_6m	0.981	0.019	0.002	0.004	0.017	0.004
PM_8m	0.986	0.007	0.025	0.002	0.002	0.018
PM_9m	0.982	0.015	0.012	0.002	0.003	0.022
PM_10m	0.985	0.007	0.011	0.023	0.014	0.009
FS-1h	0.692	0.017	0.001	0.024	0.004	0.007
FS-2h	0.861	0.047	0.009	0.016	0.029	0.007
VM-1h	0.789	0.012	0.004	0.011	0.002	0.003
VM-2h	1.036	0.004	0.021	0.017	0.026	0.026

7.5.4 Impact of Interval Length

I is an important parameter of SAW. Its default length is 10,000 write requests. We also experimented with lengths of 5,000, 15,000, 20,000 and 25,000. They are referred to as `5k`, `10k`, `15k`, `20k` and `25k`, respectively. Here we highlight their standard deviation in each case in Table 7.2 to shown the difference. Results of average erase counts can be found in the next subsection. There are the mean values over five intervals, as well as the absolute mean differences between the value of each I and the mean. From Table 7.2 we can see the fluctuation caused by changes of I is insignificant.

7.5.5 Full Results with the Prototype and FlashSim

We have recorded all experimental results for reference. As is mentioned in the thesis, due to the essentially non-deterministic nature of the operating system, there can be differences between each run of the experiments. Here, we present the full experimental results of `baseline`, `BET` and `SAW` in terms of the average erase count, standard deviation, the counts of write and read operations in Table 7.3, 7.4 and 7.5. The three tables show results recorded at each time, respectively. Note that the count of erase operations is not separately listed because it can be computed using the average erase count in each table.

The detailed experimental results of five settings on the interval I are presented in Table 7.6, including the average erase count and standard deviation.

Results from the FlashSim simulator have been collected too. The average erase count, standard deviation and service time of `lazy`, `lazy-S`, `BET` and `BET-S`, `OWL` and `O-SAW`, are separated into Table 7.7, 7.8 and 7.9 for readability. The traces of VM-1h and VM-2h were not fed to FlashSim because they are too short.

The result of the prototype and that of FlashSim under the same setting may be different, or even vary significantly. We ascribe this phenomenon to the distinction between simulators. The prototype of SAW can be viewed as a full-system simulator while FlashSim performs standalone trace-driven simulation.

7.6 Summary

In this chapter, we revisit the write endurance issue of flash device, and elaborate a novel scheme named *operating **S**ystem-**A**ssisted **W**ear leveling* (SAW). We attempt to seek the assistance of the OS to manage flash device. In this chapter, the OS participates in the process of wear leveling. An analytic model has been set up for the OS to quantitatively estimate the temperature of files. The degree calculated from the model is sent to the lower-level FTL along with data. The FTL subsequently references the temperature of date segments to conduct the block allocation. To make use of the temperature, we customize the block organization of the FTL in an exponential division. We have also developed a prototype with the idea of SAW based on Linux virtual file system, an open-source flash file system and FTL. Experiments show that the collaboration between the OS and the FTL substantially improves the wear evenness by as much as 85.0% compared to the latest FTL-based wear leveling algorithms. This significant improvement confirms our hypothesis that the participation of the OS into flash memory management is able to bring in considerable profits.

Table 7.3: Average Erase Count, Standard Deviation, the Counts of Write and Read Operations of `baseline`, `BET` and `SAW` (1st Time)

Benchmark		Average Erase Count			Standard Deviation			Write Operations			Read Operations		
		`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`
Postmark	PM-1m	6.572	6.624	6.842	1.471	1.350	0.791	2,972,904	3,000,917	3,119,305	5,949,162	5,634,356	5,049,137
	PM-2m	12.043	12.196	12.643	2.387	2.115	0.860	5,877,722	5,950,703	6,196,399	8,443,477	9,615,839	9,516,198
	PM-3m	17.512	17.771	18.464	3.167	2.731	0.912	8,779,117	8,896,513	9,281,541	12,810,709	13,366,750	14,236,254
	PM-4m	22.985	23.430	24.276	3.864	3.271	0.957	11,681,921	11,836,146	12,363,103	16,897,790	17,397,386	20,548,770
	PM-5m	28.446	28.855	30.075	4.497	3.798	0.952	14,578,317	14,767,427	15,437,939	21,119,642	22,706,330	23,939,011
	PM-6m	33.956	34.461	35.929	5.186	4.319	0.992	17,500,788	17,737,843	18,544,059	26,133,797	25,403,395	32,996,678
	PM-7m	39.590	40.135	41.868	5.828	4.894	0.976	20,485,814	20,746,099	21,692,520	30,624,022	29,920,760	49,397,386
	PM-8m	45.249	45.916	47.881	6.606	5.509	0.975	23,484,623	23,796,587	24,881,391	35,741,196	34,436,772	55,940,048
	PM-9m	50.798	51.627	53.847	7.207	6.098	0.979	26,429,906	26,812,825	28,042,582	40,370,526	40,111,468	47,798,743
	PM-10m	56.377	57.412	59.750	7.817	6.525	0.991	29,384,145	29,829,525	31,170,566	45,729,523	47,253,426	48,540,694
Filebench	FS-1h	7.628	7.224	6.655	0.915	0.825	0.678	4,035,325	3,788,685	3,518,173	253,787,028	233,511,118	213,650,267
	FS-2h	15.463	14.195	12.767	1.285	1.068	0.817	8,192,460	7,440,788	6,761,944	520,335,165	465,269,296	417,412,195
	VM-1h	34.619	34.574	27.985	3.219	2.892	0.738	18,354,325	18,180,748	14,829,527	161,193,400	164,616,315	135,871,795
	VM-2h	65.281	68.581	55.862	4.747	4.346	0.976	34,619,953	36,225,506	29,611,400	364,102,368	326,448,894	270,803,909

Table 7.4: Average Erase Count, Standard Deviation, the Counts of Write and Read Operations of `baseline`, `BET` and `SAW` (2nd Time)

Benchmark		Average Erase Count			Standard Deviation			Write Operations			Read Operations		
		`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`
Postmark	PM-1m	6.572	6.621	6.839	1.472	1.328	0.777	2,977,457	2,996,229	3,117,289	5,603,129	4,346,563	5,790,084
	PM-2m	12.060	12.173	12.667	2.398	2.094	0.864	5,886,879	5,935,576	6,208,200	8,700,539	8,570,968	9,520,877
	PM-3m	17.476	17.743	18.469	3.126	2.693	0.917	8,791,218	8,880,335	9,284,858	12,817,376	13,057,314	14,646,823
	PM-4m	23.035	23.379	24.270	3.877	3.255	0.942	11,708,459	11,834,954	12,361,087	17,244,826	19,895,452	19,788,998
	PM-5m	28.509	28.853	30.085	4.561	3.752	0.958	14,610,057	14,766,016	15,442,771	21,478,833	26,090,482	23,782,072
	PM-6m	34.014	34.431	35.895	5.097	4.242	1.004	17,528,201	17,721,035	18,525,806	25,659,251	28,238,393	33,947,237
	PM-7m	39.626	40.155	41.861	5.818	4.811	0.961	20,503,432	20,752,174	21,688,147	30,191,413	32,778,028	45,098,977
	PM-8m	45.270	45.907	47.865	6.602	5.470	0.987	23,498,782	23,801,510	24,869,822	34,692,895	36,817,108	59,203,647
	PM-9m	50.900	51.613	53.799	7.272	6.098	0.966	26,478,332	26,802,807	28,014,754	39,780,798	41,270,430	71,700,908
	PM-10m	56.457	57.172	59.724	7.861	6.566	0.989	29,427,218	29,770,773	31,155,659	44,148,945	48,942,244	47,564,490
Filebench	FS-1h	7.731	6.754	6.695	0.941	0.816	0.684	4,091,164	3,553,758	3,538,685	256,187,246	218,886,483	215,527,014
	FS-2h	15.284	14.553	14.135	1.296	1.086	0.837	8,097,963	7,673,349	7,486,036	508,522,787	481,981,880	462,880,636
	VM-1h	33.276	34.123	29.758	3.130	2.833	0.761	17,642,494	18,003,351	15,769,747	156,901,333	160,309,605	147,363,881
	VM-2h	66.426	63.870	61.740	4.873	4.222	1.003	35,227,359	33,746,903	32,729,150	316,867,995	385,058,182	277,678,522

Table 7.5: Average Erase Count, Standard Deviation, the Counts of Write and Read Operations of `baseline`, `BET` and `SAW` (3rd Time)

Benchmark		Average Erase Count			Standard Deviation			Write Operations			Read Operations		
		`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`
Postmark	PM-1m	6.568	6.635	6.828	1.466	1.342	0.782	2,975,544	3,005,347	3,112,347	5,269,322	5,423,570	7,082,480
	PM-2m	12.031	12.201	12.644	2.382	2.115	0.866	5,871,655	5,951,349	6,194,713	8,804,954	10,184,471	9,911,950
	PM-3m	17.504	17.769	18.445	3.165	2.718	0.918	8,773,932	8,890,051	9,272,150	13,180,203	16,262,391	14,982,829
	PM-4m	23.012	23.407	24.278	3.837	3.245	0.964	11,692,882	11,826,157	12,363,732	17,205,703	17,782,500	19,032,648
	PM-5m	28.446	28.845	30.067	4.475	3.776	0.961	14,577,788	14,760,804	15,433,961	21,460,584	21,652,666	23,525,050
	PM-6m	33.976	34.449	35.902	5.163	4.259	0.979	17,509,942	17,729,383	18,528,649	26,179,472	27,685,902	37,342,566
	PM-7m	39.569	40.159	41.872	5.806	4.831	0.975	20,471,142	20,758,460	21,692,750	30,716,852	32,483,543	50,465,286
	PM-8m	45.233	45.946	47.891	6.581	5.484	1.012	23,476,279	23,806,545	24,878,595	36,004,233	40,954,853	61,744,099
	PM-9m	50.820	51.693	53.808	7.152	6.176	0.969	26,437,793	26,828,372	28,014,460	39,524,900	48,308,253	68,480,672
	PM-10m	56.366	57.392	59.686	7.849	6.633	0.974	29,380,836	29,811,485	31,137,180	44,703,237	44,835,663	47,361,632
Filebench	FS-1h	7.935	7.251	6.893	0.942	0.842	0.691	4,200,841	3,825,111	3,645,013	261,152,625	238,413,134	221,023,278
	FS-2h	15.579	14.295	14.859	1.334	1.093	0.852	8,253,135	7,544,699	7,870,924	526,400,710	475,705,393	484,400,972
	VM-1h	33.592	34.077	32.638	3.154	2.920	0.793	17,810,247	17,984,757	17,296,913	159,574,537	160,924,500	141,305,567
	VM-2h	66.478	66.029	63.885	4.794	4.295	1.057	35,254,799	34,859,154	33,866,482	327,086,848	322,980,052	296,867,320

Table 7.6: Average Erase Count and Standard Deviation of 5k, 10k, 15k, 20k and 25k

Benchmark		Average Erase Count					Standard Deviation				
		5k	10k	15k	20k	25k	5k	10k	15k	20k	25k
Postmark	PM-1m	6.822	6.828	6.831	6.828	6.832	0.782	0.782	0.772	0.792	0.786
	PM-2m	12.642	12.644	12.644	12.656	12.631	0.873	0.866	0.866	0.867	0.852
	PM-3m	18.448	18.445	18.433	18.442	18.452	0.913	0.918	0.913	0.900	0.921
	PM-4m	24.249	24.278	24.239	24.260	24.237	0.942	0.964	0.973	0.967	0.952
	PM-5m	30.037	30.067	30.058	30.046	30.041	0.989	0.961	0.966	0.963	0.953
	PM-6m	35.877	35.902	35.888	35.891	35.884	1.000	0.979	0.992	0.964	0.978
	PM-7m	41.848	41.872	41.843	41.865	41.823	0.966	0.975	0.992	0.959	0.970
	PM-8m	47.877	47.891	47.884	47.896	47.858	0.980	1.011	0.985	0.988	0.968
	PM-9m	53.797	53.808	53.832	53.825	53.806	0.966	0.969	0.984	0.985	1.004
	PM-10m	59.722	59.686	59.702	59.635	59.687	0.978	0.974	1.009	0.971	0.995
Filebench	FS-1h	7.664	6.893	7.016	7.152	7.378	0.708	0.691	0.668	0.692	0.698
	FS-2h	15.172	14.859	14.173	14.234	14.749	0.908	0.852	0.845	0.832	0.868
	VM-1h	33.422	32.638	32.741	31.086	32.857	0.800	0.793	0.778	0.786	0.785
	VM-2h	63.635	63.885	62.587	63.971	63.940	1.032	1.057	1.019	1.010	1.062

Table 7.7: Average Erase Count, Standard Deviation and Service Time of `lazy` and `lazy-S`

Trace	Average Erase Count		Standard Deviation		Service Time (second)	
	`lazy`	`lazy-S`	`lazy`	`lazy-S`	`lazy`	`lazy-S`
PM-1m	7.887	7.876	2.952	2.184	1,556.628	1,554.672
PM-2m	17.670	17.622	4.352	2.888	3,294.608	3,286.003
PM-3m	27.436	27.395	4.481	3.364	5,029.318	5,022.122
PM-4m	37.263	37.067	6.310	3.627	6,775.204	6,740.659
PM-5m	47.056	46.739	7.159	3.484	8,513.718	8,457.832
PM-6m	56.904	56.667	7.833	4.027	10,263.168	10,221.351
PM-7m	66.756	66.518	8.458	4.127	12,014.693	11,972.629
PM-8m	76.791	76.521	9.103	4.126	13,795.465	13,747.889
PM-9m	86.744	86.437	9.586	4.229	15,564.870	15,510.796
PM-10m	96.480	96.183	10.089	4.329	17,295.008	17,242.518
FS-1h	8.25	8.236	3.256	2.431	1,619.643	1,616.433
FS-2h	17.415	17.312	4.833	3.258	3,246.716	3,228.453

Table 7.8: Average Erase Count, Standard Deviation and Service Time of `BET` and `BET-S`

Trace	Average Erase Count		Standard Deviation		Service Time (second)	
	`BET`	`BET-S`	`BET`	`BET-S`	`BET`	`BET-S`
PM-1m	7.759	7.759	2.932	1.519	1,534.515	1,534.153
PM-2m	17.400	17.400	4.379	1.785	3,246.938	3,246.937
PM-3m	27.048	27.048	5.488	13833	4,960.660	4,960.621
PM-4m	36.732	34.732	6.360	1.875	6,679.744	6,679.732
PM-5m	46.355	46.355	7.255	1.873	8,389.552	8,389.545
PM-6m	56.096	56.096	7.901	1.867	10,119.220	10,119.206
PM-7m	65.948	65.948	8.436	1.839	11,870.850	11,870.873
PM-8m	75.953	75.953	9.256	1.825	13,646.346	13,646.327
PM-9m	85.873	85.873	9.672	1.815	15,409.453	15,458.531
PM-10m	95.618	95.618	10.201	1.772	17,141.411	17,141.443
FS-1h	8.139	8.139	3.214	1.908	1,597.788	1,597.665
FS-2h	17.190	17.190	4.687	2.484	3,202.400	3,202.490

Table 7.9: Average Erase Count, Standard Deviation and Service Time of `OWL` and `O-SAW`

Trace	Average Erase Count		Standard Deviation		Service Time (second)	
	`OWL`	`O-SAW`	`OWL`	`O-SAW`	`OWL`	`O-SAW`
PM-1m	7.926	7.929	1.017	0.977	1,584.611	1,585.733
PM-2m	17.569	17.569	0.987	1.026	3,297.739	3,297.769
PM-3m	27.211	27.217	1.000	1.054	5,010.163	5,012.342
PM-4m	36.883	36.888	1.014	1.045	6,730.632	6,732.437
PM-5m	46.511	46.513	1.002	1.053	8,438.900	8,439.666
PM-6m	56.248	56.252	0.972	1.011	10,168.723	10,169.981
PM-7m	66.098	66.102	0.942	0.987	11,919.311	11,920.698
PM-8m	76.103	16.104	0.934	0.980	13,696.460	13,697.167
PM-9m	86.021	86.022	0.944	0.970	15,458.531	15,458.991
PM-10m	95.763	95.762	0.993	1.028	17,189.121	17,188.892
FS-1h	8.331	8.338	1.049	1.148	1,665.058	1,667.493
FS-2h	17.381	17.390	1.440	1.541	3,276.431	3,279.699

Chapter 8

Conclusion

8.1 Thesis Contributions

Three decades have passed since the invention of flash memory. The widespread utilization of NAND flash memory-based storage devices requires that the management over flash memory must be effective and efficient. Traditional strategies to manage flash device are not so sufficient today. Three new approaches have been discussed in this thesis to show our efforts to explore the arts of developing modules for flash memory management. They are as follows:

- Module-cooperative flash management. A module of the FTL focuses on a specific aspect of flash management. One module can take advantage of another one's perspective to manage the flash device. In this thesis, we have a deep cooperation between address mapping and wear leveling.

- Workload-adaptive flash management. Flash devices serve workloads to store and access data. If access behaviors of workloads are correctly interpreted, the access performance of flash devices can be favourably improved. We have attempted for address mapping and RAM buffer management.

- OS-assisted flash management. The OS has a global perspective of files and workloads. The participation of the OS enables the FTL to utilize the OS's knowledge of data and files for flash memory management. An algorithm with the OS-assisted feature has been devised for wear leveling.

The schemes proposed in this thesis have been respectively verified through experiments. Their effectiveness is significant as reflected from experimental

results. As for the efficiency, each of them could attain their corresponding goals with marginal overheads. So we conclude that they are effective and efficient.

8.2 Future Directions

What we expect is that our contributions will initiate new explorations into flash memory management that can further enhance the utilization of flash devices. Even through flash-based products have been in market for quite a long time, they are still not so mature as ferromagnetic hard disks. There is a capacious field waiting for us to plough. For my future work, some possible directions are:

- *A combination of the three said approaches.* The above methods for flash management modules are not isolated but can be organically combined. For example, SAW already has the cooperation between wear leveling and page-level address mapping with the presence of OS's assistance.
- *From performance or endurance to energy efficiency.* Energy efficient storage is essential for both hand-held devices and enterprise servers. We plan to investigate the issue of power consumption of NAND flash devices. It is surely based on the knowledge of innate characteristics of NAND flash itself and the understanding of access behaviors of workloads.
- *Big Data and cloud storage.* Big Data and cloud storage are drawing attentions of researchers and practitioners. As flash-based SSDs are widely used for enterprise servers and data centers, we want to explore the impact of the huge amount of data and networking environment on flash-based storage devices.

Most of the above thoughts are primary and require comprehensive investigation. Profits are supposed to be gained on access performance, energy consumption or data reliability through implementations of these ideas. The future of flash memory is bright. However, its inherent characteristics and the situations it may be exposed into bring in challenges as well as chances. A deeper understanding of the flash device as well as the many various environments and workloads is the key to advance our exploration. What we expect is that our efforts can help to improve the utilization of NAND flash memory-based storage devices which may in turn attract more attention to the upgrade of the secondary storage and data management of computer systems.

Bibliography

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[2] AMD. Amd radeon ramdisk. http://www.radeonmemory.com/software_4.0.php, 2013.

[3] Amir Ban. Flash file system, April 1995. US 5404485 A.

[4] Amir Ban. Flash file system optimized for page-mode flash technologies, August 1999. US 5937425 A.

[5] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[6] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: measurements and analysis. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[7] Joe Brewer and Manzur Gill. *Nonvolatile memory technologies with emphasis on flash: A comprehensive guide to understanding and using flash memory devices*. Wiley-IEEE Press, 1st edition, 2008.

[8] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *SIGPLAN Not.*, 44(3):217–228, March 2009.

[9] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1126–1130, New York, NY, USA, 2007. ACM.

[10] Li-Pin Chang and Li-Chun Huang. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '11, pages 31–40, New York, NY, USA, 2011. ACM.

[11] Li-Pin Chang and Tei-Wei Kuo. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 862–868, New York, NY, USA, 2004. ACM.

[12] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 3(4):837–863, November 2004.

[13] Li-Pin Chang and You-Chiuan Su. Plugging versus logging: a new approach to write buffer management for solid-state disks. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 23–28, New York, NY, USA, 2011. ACM.

[14] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Trans. Comput.*, 59(1):53–65, January 2010.

[15] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[16] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 22–32, New York, NY, USA, 2011. ACM.

[17] Feng Chen, Tian Luo, and Xiaodong Zhang. 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 stroage technologies*, FAST'11, Berkeley, CA, USA, 2011. USENIX Association.

[18] Hyunjin Cho, Dongkun Shin, and Young Ik Eom. 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*,

DATE '09, pages 507–512, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[19] Yuan-Sheng Chu, Jen-Wei Hsieh, Yuan-Hao Chang, and Tei-Wei Kuo. A set-based mapping strategy for flash-memory reliability enhancement. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 405–410, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[20] Thomas Claburn. Google plans to use intel SSD storage in servers. http://www.informationweek.com/storage/systems/google-plans-to-use-intel-ssd-storage-in/207602745, May 2008.

[21] Intel Corporation. What are the advantages of TRIM and how can I use it with my SSD? http://www.intel.com/support/ssdc/hpssd/sb/CS-031846.htm.

[22] Peter Desnoyers. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 12:1–12:10, New York, NY, USA, 2012. ACM.

[23] Linux Memory Technology Devices. UBI - unsorted block images, 2008. http://www.linux-mtd.infradead.org/doc/ubi.html.

[24] Daniel Ellard, Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Margo Seltzer. Attribute-based prediction of file properties. Technical Report 14-03, Harvard University, Cambridge, Massachusetts, December 2003.

[25] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[26] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 24–33, New York, NY, USA, 2009. ACM.

[27] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, FAST '12, pages 17–24. USENIX, February 2012.

[28] Yong Guan, Guohui Wang, Yi Wang, Renhai Chen, and Zili Shao. BLog: block-level log-block management for NAND flash memorystorage systems. *SIGPLAN Not.*, 48(5):111–120, June 2013.

[29] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 229–240, New York, NY, USA, 2009. ACM.

[30] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, Berkeley, CA, USA, 2011. USENIX Association.

[31] John L. Hennessy and David A. Patterson. *Computer architecture: A quantitative approach (the Morgan Kaufmann series in computer architecture and design)*. Morgan Kaufmann, May 2002.

[32] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient identification of hot data for flash memory storage systems. *Trans. Storage*, 2(1):22–40, February 2006.

[33] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[34] Yingbo Hu. MLC vs. SLC NAND flash in embedded systems. Technical report, Micro Digital, Inc, September 2009.

[35] Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Joint management of ram and flash memory with access pattern considerations. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 882–887, New York, NY, USA, 2012. ACM.

[36] Adrian Hunter. A brief introduction to the design of UBIFS, March 2008.

[37] Micron Technology Inc. Technical note: Design and use considerations for NAND flash memory. Technical report, Micron Technology Inc., 2006.

[38] Micron Technology Inc. Small-block vs. large-block NAND flash devices. technical report (TN-29-07). Technical report, Micron Technology Inc., May 2007.

[39] Micron Technology Inc. Bad block management in NAND flash memories. Technical report, Micron Technology, Inc., July 2010.

[40] Micron Technology Inc. TN-26-61: Wear-leveling in Micron® NAND flash memory. Technical report, Micron Technology, Inc, Oct 2011.

[41] Micron Technology, Inc. NAND flash memory datasheet (MT29F16G08AJADAWP), Feburary 2012.

[42] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. Operating system support for dynamic over-provisioning of solid state drives. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1753–1758, New York, NY, USA, 2012. ACM.

[43] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: flash-aware buffer management policy for portable media players. *Consumer Electronics, IEEE Transactions on*, 52(2):485–493, 2006.

[44] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 160–164, New York, NY, USA, 2007. ACM.

[45] Jürgen Kaiser, Fabio Margaglia, and André Brinkmann. Extending SSD lifetime in database applications with page overwrites. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 11:1–11:12, New York, NY, USA, 2013. ACM.

[46] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, New York, NY, USA, 2006. ACM.

[47] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Trans. Comput.*, 58(6):744–758, June 2009.

[48] Jeffrey Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance Inc., Oct. 1997.

[49] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 327–338, Washington, DC, USA, 2008. IEEE Computer Society.

[50] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *Trans. Storage*, 8(4):14:1–14:25, December 2012.

[51] Hyojun Kim and Seongjun Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 16:1–16:14, Berkeley, CA, USA, 2008. USENIX Association.

[52] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48:366–375, 2002.

[53] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. FlashSim: A simulator for NAND flash-based solid-state drives. In *Proceedings of the 2009 First International Conference on Advances in System Simulation*, SIMUL '09, pages 125–131, Washington, DC, USA, 2009. IEEE Computer Society.

[54] Yohwan Koh. NAND flash scaling beyond 20nm. In *Memory Workshop, 2009. IMW'09. IEEE International*, pages 1–3. IEEE, 2009.

[55] Duckhoi Koo and Dongkun Shin. Adaptive log block mapping scheme for log buffer-based FTL (flash translation layer). In *IWSSPS 2009: International Workshop on Software Support for Portable Storage*, 2009.

[56] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Janus-FTL: finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 169–178, New York, NY, USA, 2010. ACM.

[57] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[58] Hyun-Seob Lee, Hyun-Sik Yun, and Dong-Ho Lee. HFTL: hybrid flash translation layer based on hot data identification for flash memory. *IEEE Transactions on Consumuer Electronics*, 55(4):2005–2011, 2009.

[59] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 863–870, New York, NY, USA, 2009. ACM.

[60] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.

[61] Seungjae Lee, Young-Taek Lee, Wook-Kee Han, Dong-Hwan Kim, Moo-Sung Kim, Seung-Hyun Moon, Hyun Chul Cho, Jung-Woo Lee, Dae-Seok Byeon, Young-Ho Lim, et al. A 3.3 V 4 Gb four-level NAND flash memory with 90 nm CMOS technology. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 52–513. IEEE, 2004.

[62] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, 2008.

[63] Yong-Goo Lee, Dawoon Jung, Dongwon Kang, and Jin-Soo Kim. μ-FTL:: a memory-efficient flash translation layer supporting multiple mapping granularities. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 21–30, New York, NY, USA, 2008. ACM.

[64] Sang-Phil Lim, Sang-Won Lee, and Bongki Moon. FASTer FTL for enterprise-class flash memory SSDs. *Storage Network Architecture and Parallel I/Os, IEEE International Workshop on*, 0:3–12, 2010.

[65] Wen-Huei Lin and Li-Pin Chang. Dual greedy: Adaptive garbage collection for page-mapping solid-state disks. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 117–122, 2012.

[66] Duo Liu, Tianzheng Wang, Yi Wang, Zhiwei Qin, and Zili Shao. A block-level flash memory management scheme for reducing write activities in PCM-based embedded systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1447–1450, 2012.

[67] Samsung Electronics Co. Ltd. XSR 1.5 bad block management. Technical report, Samsung Electronics Co., Ltd, May 2007.

[68] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX conference on File and stroage technologies*, FAST'13, Berkeley, CA, USA, 2013. USENIX Association.

[69] Dongzhe Ma, Jianhua Feng, and Guoliang Li. LazyFTL: a page-level flash translation layer optimized for NAND flash memory. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1–12, New York, NY, USA, 2011. ACM.

[70] Charles Manning. How YAFFS works, March 2012. http://www.yaffs.net/sites/yaffs.net/files/HowYaffsWorks.pdf.

[71] Fujio Masuoka, Masamichi Asano, Hiroshi Iwahashi, Teisuke Komuro, and Shinichi Tanaka. A new flash E^2PROM cell using triple polysilicon technology. In *1984 International Electron Devices Meeting*, volume 30, pages 464–467, 1984.

[72] Lucas Mearian. MySpace replaces all server hard disks with flash drives. http://www.computerworld.com/s/article/9139280/MySpace_replaces_all_server_hard_disks_with_flash_drives, October 2009.

[73] Michael Mesnier, Eno Thereska, Gregory R Ganger, Daniel Ellard, and Margo Seltzer. File classification in self-* storage systems. In *Proceedings of International Conference on Autonomic Computing*, pages 44–51. IEEE, 2004.

[74] Muthukumar Murugan and David. H. C. Du. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–12, Washington, DC, USA, 2011. IEEE Computer Society.

[75] Sai Krishna Mylavarapu, Siddharth Choudhuri, Aviral Shrivastava, Jongeun Lee, and Tony Givargis. FSAF: file system aware flash translation layer for NAND flash memories. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 399–404, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[76] nandsim. How do I use NAND simulator? http://www.linux-mtd.infradead.org/faq/nand.html, Feburary 2008.

[77] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4:10:1–10:23, November 2008.

[78] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 145–158, New York, NY, USA, 2009. ACM.

[79] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Exploiting memory device wear-out dynamics to improve NAND flash memory system performance. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association.

[80] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–23, 2008.

[81] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[82] Young-Bog Park and D.K. Schroder. Degradation of thin tunnel gate oxide under constant Fowler-Nordheim current stress for a flash EEPROM. *IEEE Transactions on Electron Devices*, 45(6):1361–1368, 1998.

[83] Suraj Pathak, Y. C. Tay, and Qingsong Wei. Power and endurance aware flash-PCM memory system. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–6, 2011.

[84] Storage Performance Council. Storage Performance Council (SPC) storage traces. http://traces.cs.umass.edu/, December 2009.

[85] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 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, CODES/ISSS '10, pages 173–182, New York, NY, USA, 2010. ACM.

[86] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 157–166, 2011.

[87] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan. MNFTL: an efficient flash translation layer for MLC NAND flash memory storage systems. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 17–22, New York, NY, USA, 2011. ACM.

[88] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[89] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits: A design perspective*. Prentice Hall, 2ed edition, 2004.

[90] John T Robinson. Data cache using dynamic frequency based replacement and boundary criteria, August 1991. US Patent 5,043,885.

[91] Dongyoung Seo and Dongkun Shin. Recently-evicted-first buffer replacement policy for flash storage devices. *IEEE Transactions on Consumuer Electronics*, 54(3):1228–1235, August 2008.

[92] Liang Shi, Jianhua Li, Chun Jason Xue, Chengmo Yang, and Xuehai Zhou. ExLRU: a unified write buffer cache management for flash memory. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 339–348, New York, NY, USA, 2011. ACM.

[93] Gyudong Shim, Youngwoo Park, and Kyu Ho Park. A hybrid flash translation layer with adaptive merge for SSDs. *Trans. Storage*, 6(4):15:1–15:27, June 2011.

[94] Hyotaek Shim, Bon-Keun Seo, Jin-Soo Kim, and Seungryoul Maeng. An adaptive partitioning scheme for dram-based cache in solid state drives. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12, 2010.

[95] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 338–349, New York, NY, USA, 2009. ACM.

[96] M. Shrestha and Lihao Xu. A quantitative framework for modeling and analyzing flash memory wear leveling algorithms. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 1836–1840, 2010.

[97] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*. J. Wiley & Sons, 2009.

[98] Linux MTD Subsystem. Memory technology device (MTD) subsystems for linux. http://www.linux-mtd.infradead.org/index.html, October 2008.

[99] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[100] File system and Storage Lab. Filebench benchmark, 2011. http://sourceforge.net/projects/filebench/.

[101] BYU trace distribution center. TPC-C database benchmark traces. http://tds.cs.byu.edu/tds/, 2001.

[102] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):5:1–5:56, May 2008.

[103] Chundong Wang and Weng-Fai Wong. ADAPT: Efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies*, MSST '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[104] Chundong Wang and Weng-Fai Wong. Extending the lifetime of NAND flash memory by salvaging bad blocks. In *15th Design, Automation, and Test in Europe (DATE 2012) conference*, pages 260–263, March 2012.

[105] Chundong Wang and Weng-Fai Wong. Observational wear leveling: an efficient algorithm for flash memory management. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 235–242, San Francisco, California, 2012. ACM.

[106] Chundong Wang and Weng-Fai Wong. SAW: system-assisted wear leveling on the write endurance of NAND flash devices. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 164:1–164:9, Austin, Texas, 2013. ACM.

[107] Chundong Wang and Weng-Fai Wong. TreeFTL: Efficient RAM management for high performance of NAND flash-based storage systems. In *16th Design, Automation, and Test in Europe (DATE 2013) conference*, pages 374–379, March 2013.

[108] Yi Wang, Luis Angel D. Bathen, Nikil D. Dutt, and Zili Shao. Meta-Cure: a reliability enhancement strategy for metadata in NAND flash memory storage systems. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 214–219, New York, NY, USA, 2012. ACM.

[109] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, Zili Shao, and Yong Guan. RNFTL: a reuse-aware NAND flash translation layer for flash memory. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 163–172, New York, NY, USA, 2010. ACM.

[110] Qingsong Wei, Bozhao Gong, Suraj Pathak, and Y. C. Tay. FlashCoop: A locality-aware cooperative buffer management for SSD-based storage cluster. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 634–643, 2010.

[111] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, Ling-Fang Zeng, and Kanzo Okada. WAFTL: A workload adaptive flash translation layer with data partition. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–12, 2011.

[112] David Woodhouse. JFFS2: The journaling flash file system, version 2, July 2003. http://sourceware.org/jffs2/jffs2.pdf.

[113] Chin-Hsien Wu and Tei-Wei Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 601–606, 2006.

[114] Guanying Wu and Xubin He. Delta-FTL: improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 253–266, New York, NY, USA, 2012. ACM.

[115] Po-Liang Wu, Yuan-Hao Chang, and Tei-Wei Kuo. A file-system-aware FTL design for flash-memory storage systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 393–398, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[116] Ming-Chang Yang, Yuan-Hao Chang, Po-Chun Huang, and Tei-Wei Kuo. Working-set-based address mapping for ultra-large-scaled flash devices. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '12, pages 493–502, New York, NY, USA, 2012. ACM.

[117] Ming-Chang Yang, Yuan-Hao Chang, Che-Wei Tsao, and Po-Chun Huang. New ERA: new efficient reliability-aware wear leveling for endurance enhancement of flash storage devices. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 163:1–163:6, New York, NY, USA, 2013. ACM.

[118] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 1–16, Berkeley, CA, USA, 2012. USENIX Association.