THE HONG KONG POLYTECHNIC UNIVERSITY

DEPARTMENT OF COMPUTING

Optimizing NAND Flash Memory Management in Resource-Constrained Embedded Systems

By

ZHIWEI QIN

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

June 2012

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signature)

ZHIWEI QIN (Name of Student)

ABSTRACT

NAND flash memory has been widely adopted in the design of various storage systems. The capacity of NAND flash memory chips has been increasing dramatically and has doubled about every two years. The increasing capacity of NAND flash memory poses new challenges for vendors on the system management. Moreover, with the multi-level-cell (MLC) NAND flash memory becoming the mainstream in the market for lower cost and/or large-scale storage systems, some new write constraints have been introduced into the flash memory chips. These constraints further pose big challenges for existing flash memory management techniques that were originally designed for single-level-cell (SLC) NAND flash memory.

In this thesis, we investigate several challenging issues in managing flash memory storage systems for resource-constrained embedded systems. Since flash memory does not support in-place updates and needs to erase before update operations, a block-device-emulation software layer, called the flash translation layer (FTL), is designed so as to provide transparent service. FTLs manage the system with three components: address translation, garbage collection, and wear-leveling. In this thesis, we optimize the management techniques in FTLs from several aspects, including the RAM cost, garbage collection overhead, and real-time storage performance taking into consideration the limited computation resource in embedded system.

First, we focus on reducing the RAM footprint for address translation when doing the mapping from logical addresses to physical addresses. To solve this problem, we propose a demand-based block-level address mapping scheme with a two-level caching mechanism for large-scale NAND flash storage systems. Our basic idea is to store the block-level address mapping table in specific pages in flash memory and design two level caches in RAM

to store the on-demand block-level address mappings. Since the entire block-level address mapping table is stored in flash memory and only the demanded address mappings are loaded into RAM, the RAM footprint can be reduced. The experimental results show that our technique can achieve a 91.68% reduction in RAM cost, while the average system response time presents an average degradation of 2.02% compared with previous work.

Second, we aim to reduce the garbage collection overhead and the average system response time while hiding the new write constraints in MLC NAND flash memory. To solve this problem, we first analyze the garbage collection procedure and conclude that the valid page copy is the essential garbage collection overhead. We then propose two approaches, namely, concentrated mapping and postponed reclamation, to effectively reduce the number of valid page copies. The experimental results show that, by reducing the garbage collection overhead, our scheme can achieve a minimum reduction of 30.92% in the average system response time compared with previous work.

Third, we study the problem of improving the real-time storage performance of NAND flash memory in real-time embedded systems. To obtain an upper bound for system response time, we propose a real-time flash translation layer scheme to hide the variable garbage collection by using a distributed partial garbage collection policy that enables the system to simultaneously reclaim space and serve the write requests. The experimental results show that our scheme not only improves the worst-case system response time and the average system response time, but also shows a significant reduction in RAM cost compared with previous work.

Keywords: NAND flash memory, flash translation layer, MLC flash, two-level cache, real-time, embedded systems.

PUBLICATIONS

Journal Papers

1. Duo Liu, Yi Wang, **Zhiwei Qin**, Zili Shao, Yong Guan, “A Space Reuse Strategy for Flash Translation Layers in SLC NAND Flash Memory Storage Systems”, Accepted in IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 2011.

2. Yi Wang, Duo Liu, **Zhiwei Qin**, Zili Shao, “Optimally Removing Inter-Core Communication Overhead for Streaming Applications on MPSoCs”, Accepted in IEEE Transactions on Computers (TC), 2011.

3. Yi Wang, Hui Liu, Duo Liu, **Zhiwei Qin**, Zili Shao, E. H.-M. Sha, “Overhead-Aware Energy Optimization for Real-Time Streaming Applications on Multiprocessor System-on-Chip”, ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 16, Issue 2, pp. 14:1-14:32, March 2011.

4. Meng Wang, Yi Wang, Duo Liu, **Zhiwei Qin**, Zili Shao, “Compiler-Assisted Leakage-Aware Loop Scheduling for Embedded VLIW DSP Processors”, Elsevier Journal of Systems and Software (JSS), Volume 83, Issue 5, pp. 772-785, May 2010.

Conference Papers

1. **Zhiwei Qin**, Yi Wang, Duo Liu, Zili Shao, “Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems”, 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012), pp. 35-44, April 16-19, 2012, Beijing, China.

2. **Zhiwei Qin**, Yi Wang, Duo Liu, Zili Shao, Yong Guan, “MNFTL: An Efficient Flash Translation Layer for MLC NAND Flash Memory Storage Systems”, in the 48th IEEE/ACM Design Automation Conference (DAC 2011), pp. 12-18, June 2011, San Diego, CA, USA.

3. **Zhiwei Qin**, Yi Wang, Duo Liu, Zili Shao, “A Two-Level Caching Mechanism for Demand-Based Page-Level Address Mapping in NAND Flash Memory Storage Systems”, in the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011), pp. 157-166, April 2011, Chicago, IL, USA.

4. **Zhiwei Qin**, Yi Wang, Duo Liu, Zili Shao, “Demand-Based Block-Level Address Mapping in Large-Scale NAND Flash Storage Systems”, in the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES +ISSS 2010), pp. 173-182, October 2010, Scottsdale, Arizona, USA.

5. Duo Liu, Tianzheng Wang, Yi Wang, **Zhiwei Qin**, Zili Shao, “A Block-Level Flash Memory Management Scheme for Reducing Write Activities in PCM-based Embedded Systems”, in the 15th Design, Automation and Test in Europe (DATE 2012), March 2012, Dresden, Germany.

6. Duo Liu, Tianzheng Wang, Yi Wang, **Zhiwei Qin**, Zili Shao, “PCM-FTL: A Write-Activity-Aware NAND Flash Memory Management Scheme for PCM-based Embedded Systems”, in the 32nd IEEE Real-Time Systems Symposium (RTSS 2011), Vienna, Austria, Nov. 29-Dec. 2, 2011.

7. Yi Wang, Duo Liu, **Zhiwei Qin**, Zili Shao, “An Endurance-Enhanced Flash Translation Layer via Reuse for NAND Flash Memory Storage Systems”, in the 14th Design, Automation and Test in Europe (DATE 2011), pp. 14-20, March 2011, Grenoble, France.

8. Yi Wang, Duo Liu, **Zhiwei Qin**, Zili Shao, “Memory-Aware Optimal Scheduling with Communication Overhead Minimization for Streaming Applications on Chip Multiprocessors”, in the 31st IEEE Real-Time Systems Symposium (RTSS 2010), pp. 350-359, November 2010, San Diego, CA, USA.

9. Yi Wang, Duo Liu, Meng Wang, **Zhiwei Qin**, Zili Shao, “Optimal Task Scheduling by Removing Inter-core Communication Overhead for Streaming Applications on MPSoC”, in the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), pp. 195-204, April 2010, Stockholm, Sweden.

10. Yi Wang, Duo Liu, Meng Wang, **Zhiwei Qin**, Zili Shao, Yong Guan, “RNFTL: A Reuse-Aware NAND Flash Translation Layer for Flash Memory”, in ACM SIGPLAN/ SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 2010), pp. 163-172, April 2010, Stockholm, Sweden.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Zili Shao, who offered me the opportunity to pursue my PhD study with a group of talented and energetic people. His professional supervision, vast knowledge and skill in many research areas really impressed me. His expertise, understanding, and patience, added considerably to my graduate experience. I am so lucky to be a student of Prof. Shao, and I would like to thank him for supporting me over the years, for giving me so much freedom to explore new areas, and for training me to be a good researcher, to be a good human being. Without his encouragement and help during the difficult times in my PhD study, this body of work would not have been possible.

I would like to thank the other members of Prof. Shao's research group - Yi Wang, Duo Liu, Tianzheng Wang, Meng Wang, Luguang Wang, Guohui Wang, Qi Zhang, and Chunjing Mao - not only for their kindly help and corporation during these years, but also for the friendly and relaxing working environment created by them. The days and nights that we worked and discussed together will become a good memory in my whole life. I would like to give a special thank to Yi Wang, who was always willing and ready to help me at any time. Without his help, I would not make so much progress in my study. I appreciate Duo Liu, Guohui Wang, Qi Zhang, and Chunjing Mao for their constructive suggestions on research ideas during my PhD study. I wish to them all the success.

I also would like to thank all my teachers from whom I learned so much in my long journey of formal education. Specially thanks go to Prof. Jiannong Cao, Prof. Qin Lu, Dr. Zhijun Wang, Dr. Bin Xiao, Dr. Qixin Wang, Dr. Yan Liu, Dr. Lei Zhang, and Dr. Alvin Chan at the Hong Kong Polytechnic University. Furthermore, I acknowledge my gratitude to Dr. Lin Zhang, Dr. Qingjun Xiao, Dr. Xiaopeng Fan, Yuanyuan Wang, Lei Xu, Yi Xu,

Yue Min, Meng Yang, Beiming Sun, Zhian He, and Kaihua Zhang, who shared with me the pleasure of the Ph.D. study at the Hong Kong Polytechnic University.

I must acknowledge Prof. Robert P. Dick at University of Michigan, Ann Arbor, for offering me the opportunity to visit UMich. His truly scientist intuition and invaluable guidance inspires and enriches my intellectual maturity that I will benefit from, for a long time to come. I offer my regards and blessings to all of those who supported me in different respects during my visit at UMich. I would especially like to acknowledge Lide Zhang, David Bild, Xuejing He, Yue Liu, Yun Xiang and Phil Knag, who have directly or indirectly collaborated on my research.

I would like to thank all the visitors who share their research experiences and comment our research work when they visited our research group. Their suggestions give me a deeper understanding of research. Especially thanks to Prof. Tei-wei Kuo, Prof. Tao Li, Prof. Jingling Xue, Prof. Nikil Dutt, Dr. Yiran Chen, and Prof. Lui Sha. They broaden my scientific outlook by providing me with multiple new ideas and research philosophy.

Finally, but most significantly, I want to thank my family. They educated and guided me and have watched over me every step of way. I would like to thank them for their long-term caring and encouragement through my entire life, for letting me pursue my dream for so long and so far away from home, and for giving me the motivation to finish this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

NAND flash memory has been widely used in embedded systems due to its non-volatility, shock-resistance, and low power-consumption property. Well-known examples are cell-phones, cameras, USB flash drives, and solid-state-drives (SSD). Similar to other storage media, the capacity of NAND flash memory chips is increasing dramatically and has doubled about every two years [96]. In 2011, one single flash memory chip with a capacity of 128GB was developed with multi-level-cell (MLC) flash technology using 20-nanometer NAND process technology [12]. The increasing capacity of NAND flash memory brings new challenges for vendors with regard to the management of flash memory storage systems. Moreover, with MLC NAND flash memory becoming the mainstream in the market for lower cost and/or large-scale storage systems, two new write constraints have been introduced into the flash memory chips compared with the single-level-cell (SLC) NAND flash memory. Random programming on pages within one block is prohibited, and multiple partial programming within one page is not allowed. The new write constraints pose big challenges for existing flash memory management techniques that were originally designed for SLC NAND flash memory. This thesis focuses on optimizing the management techniques for NAND flash memory storage systems in resource-constrained embedded systems to enhance system performance.

As a non-volatile storage device, NAND flash memory has many good properties such as small size, shock resistance, and low power consumption. However, NAND flash memory also has some constraints that impose challenges for its management. NAND flash memory does not support in-place-updates, and an update (re-write) operation on existing data in a given physical location (i.e., one page) should be preceded by an erase operation on a larger region (i.e., one block). Besides that, a block has a limited number of erase counts,

and it becomes worn-out if the erase counts reach the threshold [7, 39]. In order to hide these idiosyncrasies and to provide transparent service, a block-device-emulation software module called the flash translation layer (FTL) is built between the file system and the flash memory chip [13, 14]. FTLs manage the system with three components: address translation, garbage collection, and wear-leveling. In this thesis, we investigate several challenging issues in designing the FTLs from several aspects, including the RAM cost for address translation, the time overhead for garbage collection, and the worst-case system response time for real-time storage systems in resource-constrained embedded systems.

Many studies have been conducted on the management of NAND flash memory storage systems. A great deal of work focuses on the design of enterprise-level solid state drives (SSDs) [15, 25, 32, 53, 66, 67, 88, 93], while other work concerns the application of NAND flash memory in embedded systems [19, 42, 50, 56, 62, 70, 78, 92]. To improve the system performance of NAND flash based embedded systems, some work focuses on exploring the storage system architecture [16, 31, 38, 40, 41, 47, 57, 71, 75, 82, 86, 89, 97, 104], and some work exploits the energy consumption [44, 76, 87, 100, 103]; while other work concerns the design of the flash translation layer [13, 14, 20, 29, 91]. FTLs can be divided into three main categories: page-level mapping FTL [13], block-level mapping FTL [10, 14, 18, 73, 95], and hybrid-level mapping FTL [54, 69, 77, 80, 98, 99]. In [13], a fine-grained page-to-page mapping FTL is proposed that shows good address translation efficiency and a fast average response time. However, it suffers from a large RAM footprint problem when maintaining the address mapping table. For example, given a large-block (2KB/page) based 32GB Micron NAND flash memory chip MT29F32G08CBABAWP [3], the size of the mapping table for the page-level mapping FTL scheme [13] is 96MB, which is too big to be kept in RAM.

To reduce the size of the address mapping table, block-level mapping FTL schemes have been proposed and widely adopted in NAND flash memory storage systems [10, 14, 18, 21, 73]. One representative block-level mapping FTL scheme is called NAND flash translation layer (NFTL) [14]. Using coarse-grained block-to-block address mapping, NFTL can significantly reduce the size of the address mapping table compared with the page-level mapping FTL. However, NFTL may still suffer from a large RAM footprint problem due to the

continuously increasing flash memory capacity. For example, given the above mentioned 32GB Micron NAND flash memory chip, the block-level address mapping table could take up 1.5MB of RAM space. This large RAM footprint limits the application of flash memory in some resource-constrained embedded systems, especially in some low-end storage systems. Wu and Kuo [99] proposed an adaptive hybrid-level mapping FTL in which the address translator can dynamically and adaptively switch between page-level mapping and block-level mapping. Kim et al. [55] proposed a log-block based address mapping scheme, called log-block NFTL. In log-block NFTL, blocks in flash memory are partitioned into data blocks and log blocks. The data blocks are managed with the block-level address mapping approach, and the log blocks with the page-level address mapping approach. With the above FTLs, the flexibility of address mapping is greatly improved. However, they ignore the increased size of the address mapping table when applied to large-scale NAND flash storage systems. In this thesis, we propose a novel address mapping management scheme that can solve the RAM footprint problem for large-scale NAND flash memory storage systems in resource-constrained embedded systems.

MLC flash memory is becoming the mainstream in the market for low cost storage systems. In the FTL design for MLC flash, the address mapping approach should follow the write constraints of the flash memory chip. The three kinds of FTLs that have been proposed have mostly been designed based on the SLC flash. It would be inefficient to apply these schemes directly to MLC flash. In page-level mapping FTLs, the pages within one block can be allocated sequentially. Therefore, they can be applied to MLC flash without complicated modifications. Based on page-level mapping FTL, the DFTL scheme [33] stores the address mapping table in the flash memory and adopts a cache to store a small amount of active mapping entries in RAM. Data pages in DFTL can also be written sequentially. However, DFTL suffers from extra valid page copy overhead when flushing the dirty mapping entries from the cache to the translation blocks in the flash memory. The GFTL scheme [28] provides a guaranteed storage service by introducing some additional blocks as the buffer blocks. Although the page allocation approach in both the data blocks and the buffer blocks obeys the two write constraints of MLC flash, GFTL shows a longer average system response time due

to the earlier-triggered garbage collection. The superblock based FTL scheme (SFTL) [48] can also be applied to MLC flash. Nevertheless, it may also trigger the garbage collection procedure very early because of the fixed number of log blocks that are shared by multiple data blocks. Thus, although these FTLs can be applied to MLC flash, they still suffer from longer average system response times because of the earlier-triggered garbage collection. In this thesis, we propose two approaches to effectively reduce the garbage collection overhead in the design of FTLs for MLC NAND flash memory: concentrated mapping and postponed reclamation.

NAND flash memory has been widely used in both hard real-time and soft real-time embedded systems. However, due to variable garbage collection latencies, NAND flash memory storage systems may suffer long system response times, especially when the flash memory is close to being full. Most existing FTL schemes focus on improving the average response time, but ignore the need to provide a desirable upper bound for the worst-case response time. In previous work, several techniques have been proposed to improve the real-time performance of flash memory storage systems. Chang et al. [22] was the first to propose real-time garbage collection for flash memory storage systems, where predictable performance is guaranteed by ensuring that enough free space is always available for write requests. Although an upper bound for the response time can be obtained, their approach suffers from a slow worst-case response time and requires extra file system support. Choudhuri et al. [28] proposed a flash translation layer called GFTL to guarantee an upper bound for the system response time. GFTL reduces the upper bound by adding extra blocks as the write buffer and using a partial block cleaning policy to hide the long garbage collection latency. In order to provide enough free space to serve write requests, a block that is full will be put into a garbage collection queue, and garbage collection is consecutively performed as long as the queue is not empty. GFTL guarantees a worst-case response time for write requests; however, it suffers a longer worst-case response time for read requests. Moreover, it introduces a large number of extra page copy operations, which significantly degrades the average system response time. Since garbage collection does not occur very often, a scheme should not sacrifice too much average response time when reducing the worst-case response

time. Therefore, in this thesis, we propose a real-time flash translation layer, called RFTL, which provides not only an ideal upper bound for the worst-case system response time but also a faster average system response time.

In summary, we propose three techniques to improve the system performance of both the SLC and MLC flash memory in resource-constrained embedded systems. We first focus on reducing the RAM footprint to address translation management in large-scale NAND flash memory storage systems. The proposed FTL scheme can be applied to embedded systems that have only limited RAM space. Then, we aim to reduce the garbage collection overhead so as to improve the average system response time for MLC NAND flash memory. Finally, we study the problem of improving the real-time performance of MLC flash memory storage systems in real-time embedded systems. The proposed real-time FTL is useful in some time-critical applications.

The rest of this chapter is organized as follows: Section 1.1 presents the related work. Section 1.2 presents the unified research framework. Section 1.3 summarizes the contributions of this thesis. Section 1.4 gives the outline of the thesis.

1.1 Related Work

In this section, we briefly introduce the NAND flash memory technology, the flash memory storage systems, and some related FTL schemes.

1.1.1 NAND Flash Memory

In the past decade, NAND flash memory has been widely adopted as a secondary storage in embedded systems. As shown in Figure 1.1, a typical NAND flash memory is partitioned into two planes or four planes. Each plane consists of multiple blocks, while each block is further divided into fixed number of pages (32 pages or 64 pages). Each page contains a data area (512Bytes or 2KB) and an OOB (Out Of Band) area (32Bytes or 64Bytes). The OOB area is primarily used to store the Error Correction Code (ECC) of the corresponding page

and other information such as logical page numbers. There are three basic operations that can be performed on a NAND flash memory: *erase*, *write*, and *read*. A block is the smallest unit of erase operations, while a page is the minimum unit of read/write operations.

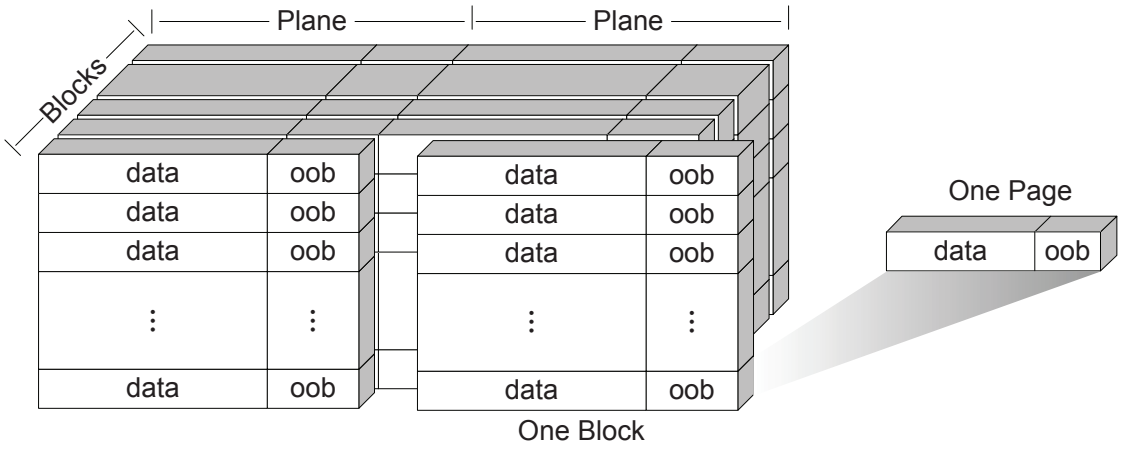


Figure 1.1: A typical NAND flash memory array.

Compared with a hard-disk drive, NAND flash memory has many advantages, such as non-volatility, a smaller size, shock resistance, and faster access times. However, NAND flash memory also has some constraints that impose challenges for its management. First, NAND flash memory suffers from out-of-place updates. An update (re-write) of existing data in a given physical location (known as a page) should be preceded by an erase operation on a larger region (known as a block). In NAND flash memory, data must be written on free pages, which could cause space to run out after a number of write operations. Thus, a block-reclaim operation known as garbage collection [20, 63] is invoked to regenerate free space for reuse. Second, a block has a limited erase lifetime. For example, one block in a SAMSUNG K9F1G08U0C SLC (Single-Level Cell) NAND flash has 100K erase counts, while one in a SAMSUNG K9G4G08U0A MLC (Multi-Level Cell) NAND flash has only 5K erase counts. A block becomes worn out if its erase counts reach the limit [7]. Third, for some NAND flash memory management schemes, not all blocks in a NAND flash get erased at the same rate, so the lifetime of specific blocks may decrease faster, which would affect the usefulness of the entire flash memory. To overcome these constraints, it is very important to guarantee that erase or write operations be evenly distributed across all blocks.

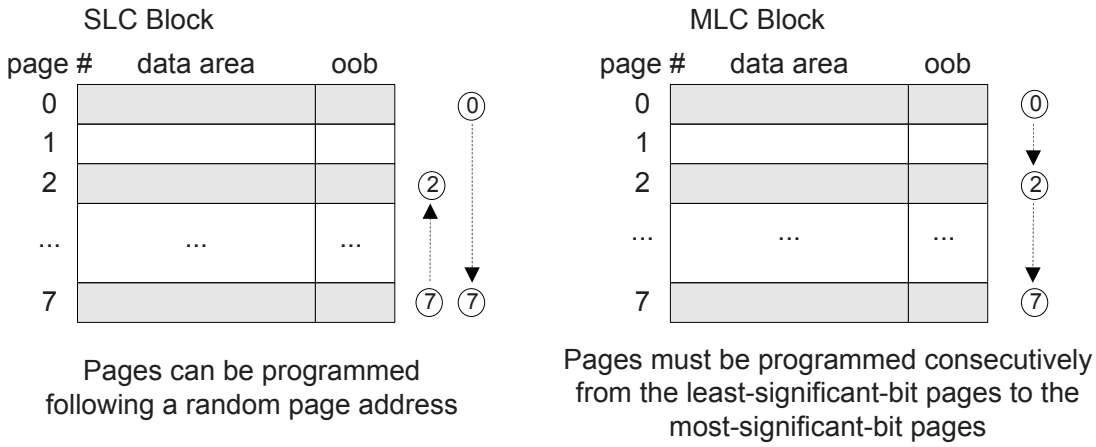


Figure 1.2: Page program addressing in SLC and MLC NAND flash memory.

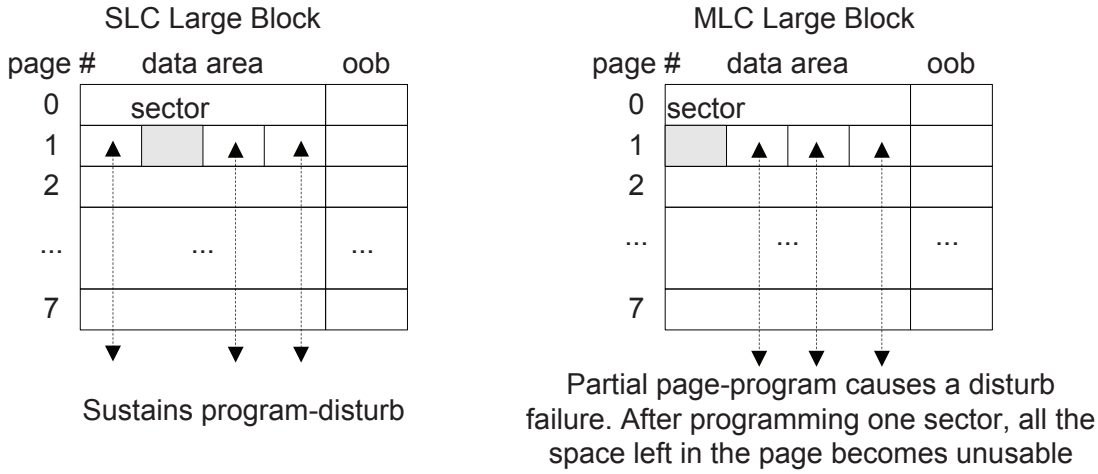


Figure 1.3: Partial page programming in SLC and MLC NAND flash memory.

To date, two types of NAND flash memory technologies have been developed: SLC flash and MLC flash. In single-level-cell (SLC) flash memory, each cell can exist in one of two states, storing one bit of information per cell. Multi-level-cell (MLC) flash memory has at least four states per cell, so it can store at least two bits of information per cell. The primary benefit of MLC flash memory is its lower cost per unit for storage due to the higher density of data. However, software complexity is also increased to compensate for a larger bit error ratio. Moreover, MLC introduces two new write constraints in the flash memory chip. First, the pages within one block must be programmed consecutively from the least significant bit (LSB) pages to the most significant bit (MSB) pages [11]. Second, only one

partial page program is allowed within one page [6]. Figure 1.2 and Figure 1.3 compare the page program addressing and the partial page programming in SLC and MLC NAND flash memory, respectively.

1.1.2 Flash Memory Storage Systems

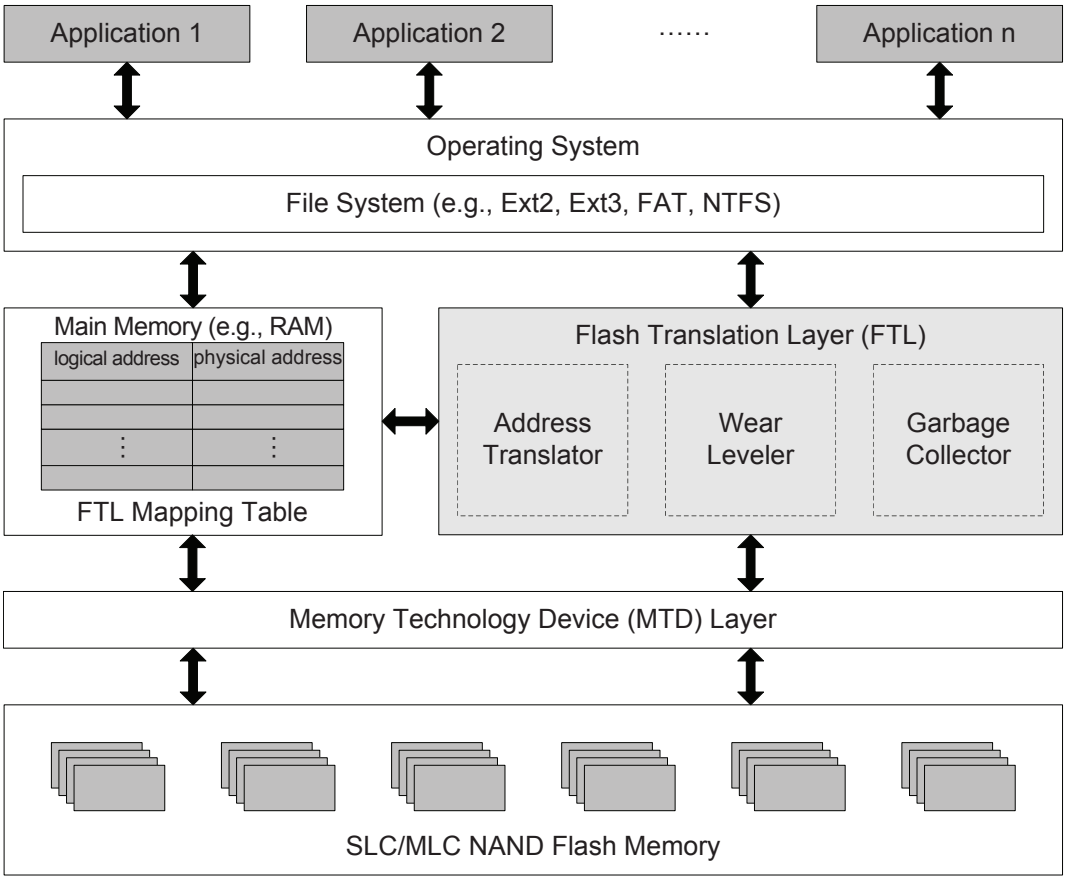


Figure 1.4: A typical management architecture of a NAND flash memory storage system with a flash translation layer.

To hide the idiosyncrasies of NAND flash memory, an intermediate software module called a flash translation layer (FTL) is employed to emulate NAND flash memory as a block device [2]. Figure 1.4 shows a software-level architecture of the incorporated flash translation layer module [58]. In this architecture, the flash translation layer provides three components: the address translator [14], garbage collector [20, 34, 90], and wear-

leveler [17, 23, 35, 36, 46]. In FTL, the address translator maintains an FTL mapping table, which is usually located in RAM, that can translate between logical addresses and physical addresses; the garbage collector reclaims space by erasing obsolete blocks in which invalid data exist; the wear-leveler is an optional component that distributes erase operations evenly across all blocks, so as to extend the lifetime of NAND flash memory. This thesis focuses on optimizing the address translator and the garbage collector in the flash translation layer so as to improve the flash memory storage performance in resource-constrained embedded systems.

In NAND flash memory based storage systems, the file system issues a read request or a write request with a logical address to the flash memory chip. The address translator first locates the corresponding physical address by searching the address mapping table in main memory. According to the out-of-place update property, if a logical address is mapped with a physical address that contains previously written data, the input data should be written to an empty physical location in which no data had previously been written. The mapping table should then be updated due to the newly changed address mapping. This procedure is called address translation. The time cost in this procedure is the address translation overhead. After the address translation, the MTD layer can perform the read or write operation on the flash media. With the write operations propagating in the flash memory, free space shrinks and garbage collection is triggered to reclaim the invalid space for reuse. The valid pages in the victim block, which is selected based on certain garbage collection policies [20, 59–61], are copied to a free block, and the original victim block is then erased. In this process, the time consumed by the valid page copy and the block erase operation is the garbage collection overhead. A write request cannot be served immediately if the garbage collection process is running. Thus, the time cost from the request issued by the file system to the finishing time of the requested operation is called the *system response time*. The system response time reflects the efficiency of the FTL schemes.

1.1.3 FTL Schemes

Many designs and implementations of NAND flash memory management have been proposed in the literature. As FTL plays a critical role in NAND flash memory management, different FTL schemes have been proposed, which can be categorized into three major types: page-level mapping FTL [13], block-level mapping FTL [10, 14, 85], and hybrid-level mapping FTL [18, 21, 27, 55, 68, 74, 77, 95, 99].

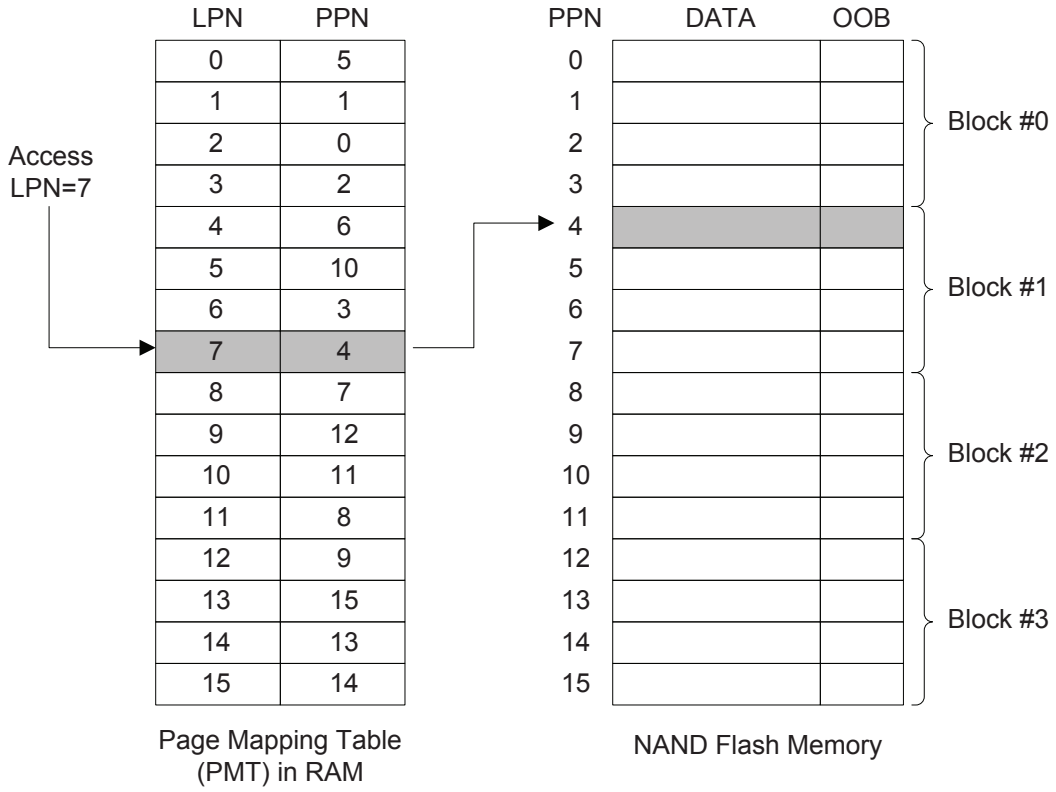


Figure 1.5: An illustration of a page-level mapping FTL scheme.

In page-level mapping FTLs, every logical page is mapped with one physical page. If the file system contains *n* logical page units, its mapping table should also have *n* entries. Figure 1.5 illustrates an example of a page-level mapping FTL. For demonstration purposes, we assume that one flash chip includes four blocks, and that each block has four pages. In that case, 16 address mapping entries are needed in the address mapping table. When the file system accesses the logical page number (LPN) 7, the physical page with the physical page number (PPN) 4 can be found from the page mapping table in RAM. The file system

can get the requested data from page 4 in flash memory. As a fine-grained page-to-page mapping approach is adopted, page-level mapping FTLs could become efficient at translating addresses. However, page-level mapping FTLs require a large amount of RAM space, which limits their usage in some resource-constrained embedded systems.

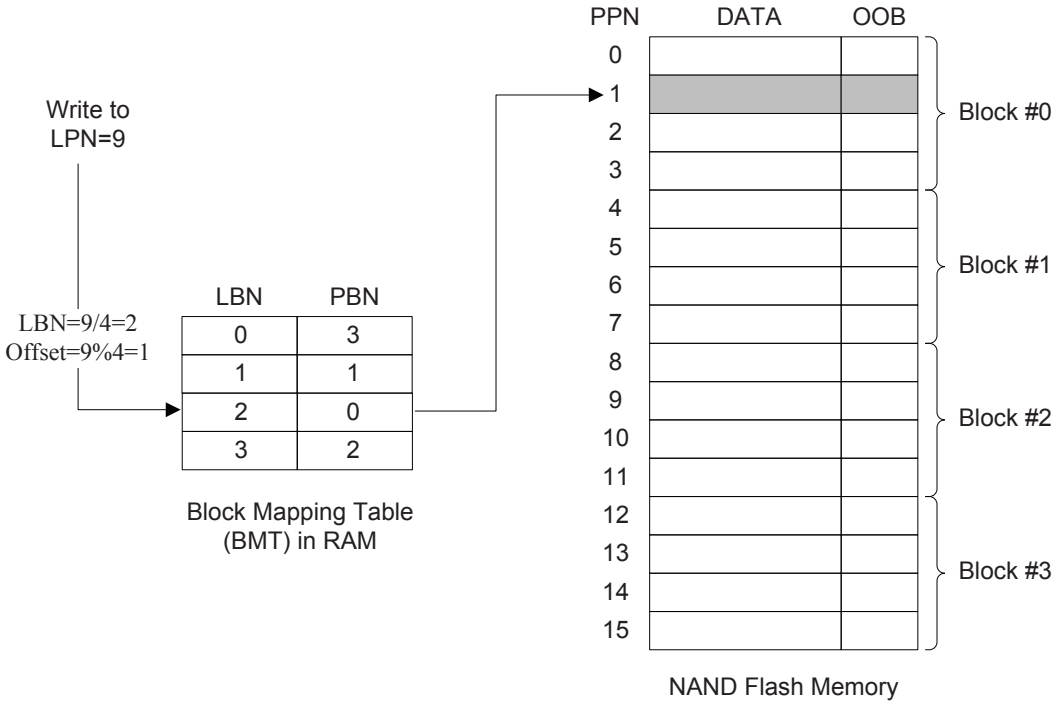


Figure 1.6: An illustration of a block-level mapping FTL scheme.

For this reason, block-level mapping FTLs [10,14] are proposed. Figure 1.6 shows an example of a block-level mapping FTL scheme. In this example, one logical block is mapped with one physical block. Given the logical page number (LPN), divided by the number of pages in each block (i.e., four), the quotient is the logical block number (LBN), and the remainder is the block offset. For the given LPN 9, the LBN and the block offset are 2 and 1, respectively. Since physical block 0 is mapped with logical block 2, the target physical page can be located using the block offset 1 in the physical block 0. As coarse-grained block-to-block mapping is used, block-level FTL requires a smaller number of address mapping entries compared with page-level mapping FTLs. However, a logical page in a block-level mapping FTL can only be written to a physical page with the designated block offset. Thus, a block-level mapping FTL is not as good as a page-level mapping FTL in mapping flexibility

and space utilization ratios.

To make a trade-off between RAM cost and address translation efficiency, hybrid-level mapping FTLs [24, 26, 28, 55, 65, 99, 102] have been introduced. Wu and Kuo [99] proposed an adaptive hybrid-level mapping FTL in which the address translator can dynamically and adaptively switch between page-level mapping and block-level mapping. Kim et al. [55] proposed a log-block based address mapping scheme, called log-block NFTL. In log-block NFTL, blocks in flash memory are partitioned into data blocks and log blocks. The data blocks are managed with the block-level address mapping approach, and the log blocks with the page-level mapping approach. The above hybrid-level mapping FTLs are a great improvement in terms of address mapping flexibility. However, they ignore the increased size of mapping tables when applied to large-scale NAND flash storage systems. The mapping table size of hybrid-level mapping FTLs tends to be larger than that of block-level mapping FTLs.

In flash memory storage systems, one approach to solving the large RAM footprint problem is to store the address mapping table in flash memory but not in RAM. However, this approach may incur extra address translation overhead for fetching the address mapping table from flash memory. In the literature, several techniques have been proposed to use a caching mechanism to improve the system performance [33, 37, 45, 49, 51, 52, 64, 79, 81, 101]. In DFTL [33], one small address mapping table cache is designed to selectively cache the on-demand page-level address mapping entries. In order to achieve higher cache hit ratios, temporal locality in workloads is well considered and the least recently used (LRU) replacement algorithm [43] is used to keep the potential requested mapping entries staying in cache. Therefore, DFTL shows good system response times for workloads with intensive temporal locality. However, it ignores the spatial locality and the access frequency of workloads, which are also important factors for accessing data. Moreover, DFTL adopts a fine-grained page-level mapping approach, and one page write operation in data blocks may cause one address mapping update operation in cache, which may incur one translation page copy operation in the translation blocks. These frequent page copy operations will lead to the erasing of more translation blocks, which will affect the efficiency of address translation. Therefore,

how to effectively reduce the RAM cost without excessively penalizing system performance becomes an important issue.

The three kinds of FTLs proposed in the literature are mostly designed based on SLC flash. Applying them to MLC flash would be too inefficient or limiting. In page-level mapping FTL schemes, pages are allocated sequentially within a block and no page status (valid or invalid) needs to be recorded in its spare area. Therefore, it is still usable to MLC flash. However, page-level mapping FTL is unsuitable for a large-sized MLC flash due to the large address mapping table. Block-level mapping FTLs [14, 95] use the block offset to locate the pages within a block, and the pages may be programmed randomly within a block. Therefore, block-level mapping FTLs may not be applicable to MLC flash. In hybrid-level mapping FTLs [28, 48, 68, 77, 99], physical blocks are logically partitioned into data blocks (primary blocks) and log blocks (replacement blocks). A data block is used to store the first written data, while log blocks are designated to record the updated data. In data block management, most hybrid-level mapping FTLs adopt the block-level mapping approach and use the block offset to locate the pages. In the GFTL scheme [28], the pages can be allocated sequentially without the block offset; however, the average system response time is longer due to the earlier-triggered garbage collection. A superblock-based FTL scheme (SFTL) [48] obeys the write constraints of MLC flash, but it may also trigger the garbage collection very early due to the fixed number of log blocks. Thus, it is necessary to design an FTL that not only can be applicable to MLC flash but that can also incur less garbage collection overhead.

In previous work, several techniques have been proposed to improve the real-time storage performance of NAND flash memory storage systems. Chang et al. [22] was the first to propose real-time garbage collection for flash memory storage systems, where predictable performance is guaranteed by ensuring that enough free space is always available for write requests. Although an upper bound for the response time can be obtained, their approach suffers from a slow worst-case response time and requires extra file system support. Choudhuri et al. [28] proposed a flash translation layer called GFTL to guarantee an upper bound for the system response time. GFTL reduces the upper bound by adding extra blocks as the write buffer and using a partial block cleaning policy to hide the long garbage collection la-

tency. In order to provide enough free space to serve the pending write requests, a block that is full will be put into a garbage collection queue, and a garbage collection is consecutively performed as long as the queue is not empty. GFTL guarantees a worst-case response time for write requests; however, it suffers a longer worst-case response time for read requests. Moreover, it introduces a large number of extra page copy operations in garbage collection, which significantly degrade the average system response time. Since garbage collection does not occur very often, a scheme should not sacrifice too much average response time when reducing the worst-case response time.

1.2 The Unified Research Framework

In this section, we present the unified research framework for the proposed techniques. Figure 1.7 illustrates the sketch of our research framework.

In this thesis, we target NAND flash memory storage systems in resource-constrained embedded systems, where RAM (SRAM or DRAM) is adopted as the main-memory and NAND flash memory serves as the secondary storage media for storing user data accessed by the file system.

In this thesis, three flash memory management techniques are presented to improve the system performance of NAND flash memory storage systems in resource-constrained embedded systems, in terms of management cost and management efficiency. As shown in Figure 1.7, we propose three techniques for designing flash translation layers, with the objective of reducing the RAM footprint, the garbage collection overhead, and the worst-case system response time, respectively.

- For the first technique, in Chapter 2, we propose a Demand-based block-level Address mapping scheme with a two-level Caching mechanism, named DAC. In DAC, we endeavor to reduce the RAM cost in maintaining the address mapping table for FTLs. To achieve this, we adopt an on-demand strategy by storing the entire block-level address mapping table in the flash memory and caching the demanded mapping entries

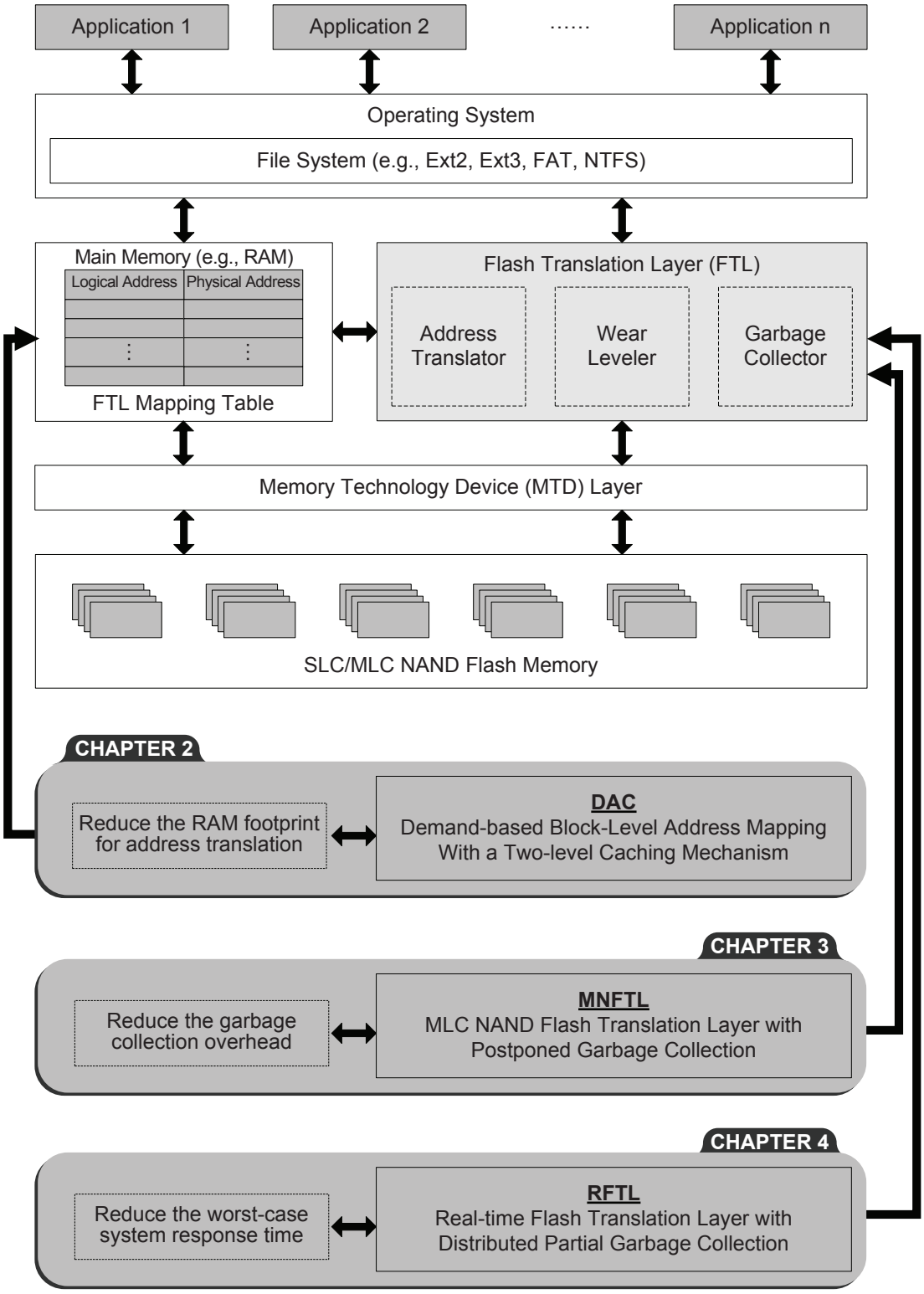


Figure 1.7: The unified research framework.

in RAM. A two-level caching mechanism is designed that takes into consideration the access patterns of workloads, such as the temporal locality, the spatial locality, and the access frequency. Taking into account the expensive overhead for flushing the dirty items from cache to flash memory, a cost-aware cache replacement policy is introduced. The proposed techniques in DAC not only effectively reduce the RAM cost but also guarantee the cache hit ratio, thereby improving the average system response time.

- For the second technique, in Chapter 3, we propose an MLC NAND Flash Translation Layer, named MNFTL, to hide the two write constraints of MLC flash and, at the same time, to effectively reduce the garbage collection overhead. To achieve this, we propose two techniques, namely concentrated mapping and postponed reclamation, to fundamentally reduce the garbage collection overhead. We analyze the garbage collection procedure adopted in traditional FTLs, and conclude that the valid page copies are the essential overhead. A novel block-level mapping approach is presented to centralize the invalid space together, so that the number of valid pages that need to be copied by the garbage collector are minimized and the latency for garbage collection is reduced. Moreover, a novel garbage collection policy is designed to postpone the time for garbage collection, which can also increase the number of invalid pages within a fully occupied block.

- For the third technique, in Chapter 4, we present a Real-time Flash Translation Layer, named RFTL, to effectively reduce the worst-case system response time for NAND flash memory storage systems. Our work is motivated by the observations that most FTLs focus on improving the average system response time but ignore the worst-case system response time, and most flash storage systems suffer from long system response times due to the variable garbage collection latency. Therefore, in RFTL, we propose a novel address mapping approach with a distributed partial garbage collection policy to shorten the blocked time caused by the garbage collection process. By introducing some extra flash blocks as buffer blocks, RFTL enables the storage system to serve the

write requests and to do the garbage collection simultaneously, which can effectively reduce the worst-case system response time.

In this thesis, we evaluate the proposed techniques of DAC, MNFTL, and RFTL using a variety of realistic I/O traces. The traces reflect the real workload of the system in accessing the secondary storage system with applications that are used daily, such as those for web surfing, document typewriting, downloading, and playing movies and games. DiskMon [1] is adopted as the tool for collecting these traces from the notebook with an Intel Pentium Dual Core 2GHz processor, a 200GB hard disk, and a 2GB DRAM. We also use some well-known traces that have been widely adopted in other studies, such as the *Financial* trace [5] and the *Websearch* trace [8]. The evaluation is conducted by a trace-driven simulation. We have developed a simulator to evaluate our three flash memory management techniques against some representative FTL designs.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- We propose for the first time a demand-based block-level address mapping scheme for NAND flash memory management, called DAC, to reduce the RAM storage overhead for large-scale NAND flash memory storage systems in resource-constrained embedded systems. DAC is designed mainly based on the idea of selectively caching some on-demand block-level mapping entries while storing the entire mapping table in the flash memory. A novel two-level caching mechanism is proposed based on the access pattern of workloads and the access behavior of flash memory chips. Experimental results show that our technique can achieve a 91.68% reduction in RAM cost while the average system response time presents a 2.02% degradation on average compared with previous work.

- We present for the first time a MLC NAND flash translation layer, called MNFTL, to hide the write constraints for MLC NAND flash memory and to essentially reduce the

garbage collection overhead. In MNFTL, a concentrated address mapping approach is proposed to distribute the invalid pages more close to each other, while a postponed garbage collection policy is used to delay the time required to perform the block reclamation, so that more valid pages may become invalid and fewer valid page copies are required. Compared with previous FTLs applicable to MLC flash, the experimental results show that, by reducing the garbage collection overhead, our scheme can achieve a minimum reduction of 30.92% in the average system response time compared with previous work. In addition, the results show that the RAM cost of MNFTL is well under control.

- We present a real-time flash translation layer for NAND flash memory management, called RFTL, to effectively reduce the worst-case system response time in real-time embedded systems based on NAND flash memory. A novel block-level address mapping scheme is presented to provide enough free space for pending write requests, while a distributed partial garbage collection policy is used to cut one garbage collection process into smaller phases and then interleave each phase with a write operation. Compared with previous work, experimental results show that our scheme not only improves the worst-case system response time and the average system response time, but also effectively reduces RAM cost compared with previous work.

- A trace-driven simulation framework is implemented to evaluate the proposed flash memory management schemes in NAND flash based embedded systems. We conducted experiments and compared our proposed schemes with representative FTL schemes. The experimental results prove the effectiveness of the proposed schemes in running with different kinds of I/O workloads.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

- In Chapter 2, we present our demand-based block-level flash memory management

technique, DAC, and show its efficiency in reducing RAM cost for large-scale NAND flash memory storage systems.

- In Chapter 3, we present our flash translation layer design for MLC NAND flash memory storage systems, MNFTL, to overcome the write constraints in MLC NAND flash memory chips. We also show that MNFTL can effectively reduce the garbage collection overhead in MLC flash management, so as to reduce the average system response time.

- In Chapter 4, we present our real-time flash memory management technique, RFTL, and show that RFTL can improve the average system response time and the worst-case system response time as well as the RAM cost for real-time embedded systems.

- In Chapter 5, we present conclusions and discuss possible future directions for research arising from this work.

CHAPTER 2

DAC: DEMAND-BASED BLOCK-LEVEL ADDRESS MAPPING WITH A TWO-LEVEL CACHING MECHANISM

2.1 Overview

The density of flash memory chips has doubled every two years in the past decade and the trend is expected to continue. The increasing capacity of NAND flash memory poses big challenges for its management. In particular, the management of address mapping in the flash translation layer (FTL) suffers from a large RAM footprint, which limits the application of flash memory in resource-constrained embedded systems. In the past decade, various FTL schemes have been proposed and different FTLs show different RAM costs and system response times. Among them, page-level FTLs [13] can achieve efficient address mapping. However, they suffer from a large RAM footprint problem in maintaining the page-level address mapping table. To reduce the size of the address mapping table, block-level FTL schemes have been proposed and widely adopted in NAND flash memory storage systems in embedded systems [10, 14, 18, 21, 27, 73]. NFTL (NAND flash translation layer) is one representative block-level FTL scheme [14]. Different from fine-grained page-level FTLs that keep the mapping information between logical pages and physical pages, NFTL maintains a block-level address mapping table in which one logical block is mapped with one primary block and one replacement block. Using coarse-grained block-to-block mapping, NFTL can significantly reduce the size of the address mapping table compared with page-level FTLs. However, NFTL may still suffer from the large RAM footprint problem due to the increasing flash memory capacity. How to effectively reduce the RAM footprint for address management becomes an important issue.

An on-demand page-level mapping scheme called DFTL [33, 84] has been proposed to solve this problem. Unlike the traditional approaches to maintaining the page-level address mapping table in RAM, DFTL stores the address mapping table in specific flash memory pages, and one cache in RAM is designed to store the on-demand address mapping entries. Moreover, a global translation directory (GTD) is created in RAM to track the location of the address mapping table from the flash memory. Taking advantage of the on-demand strategy and the temporal locality of workloads, DFTL effectively reduces the RAM footprint. However, DFTL is designed based on the page-level address mapping approach, and cannot be directly applied to solve the RAM footprint problem of NFTL, which is based on the block-level address mapping approach. Moreover, the page-level mapping table in flash memory in DFTL is still big. It not only takes up extra flash space but also introduces more management overhead. Besides that, spatial locality and access frequency are not explored by the caching scheme in DFTL.

In this chapter, we propose a Demand-based block-level Address mapping scheme with a two-level Caching mechanism (DAC) to solve the RAM footprint problem for NFTL in large-scale NAND flash memory storage systems. The basic idea of DAC is to store the block-level address mapping table in specific pages (called the translation pages) in the flash memory, and a two-level caching mechanism is designed in RAM to store the on-demand block-level address mapping entries. Considering the temporal locality of workloads, the first-level cache in DAC is dedicated to caching a small number of active block-level mapping entries. The second-level cache consists of two caches that are used to explore the spatial locality of workloads and the access frequency of address mapping entries, respectively. Since the entire block-level address mapping table is stored in flash memory and address mapping entries are loaded into RAM in an on-demand fashion, DAC is effective at reducing the RAM footprint. Moreover, the two-level caching mechanism in DAC can effectively explore the reference locality and the access frequency. Therefore, the cache hit ratio is improved and the average system response time can be kept well under the control.

We have conducted experiments on a set of traces collected from real workloads by DiskMon [1]. We compare DAC with NFTL [14] and DFTL [33] in terms of the RAM cost

and the average system response time. The experimental results show that our technique can effectively save the RAM cost with very little penalty on the average response time. On average compared with NFTL, for a 32GB NAND flash memory with 64 2K-bytes pages per block, DAC reduces the RAM cost by 91.68% with a 2.02% penalty on the average system response time.

This chapter makes the following contributions:

- We present for the first time a demand-based block-level address mapping scheme with a two-level caching mechanism for reducing RAM cost in large-scale NAND flash memory storage systems.
- We demonstrate the effectiveness of this address mapping management scheme and compare it with representative techniques using a set of real traces.

The rest of this chapter is organized as follows. Section 2.2 introduces the background and the motivation of this chapter. Section 2.3 describes our proposed on-demand block-level address mapping scheme. Section 2.4 and Section 2.5 present the performance analysis and performance evaluation of our scheme, respectively. Finally, Section 2.6 concludes the chapter.

2.2 Background and Motivation

In this section, we first introduce the evolution of NAND flash memory technology, and then describe the base-line NFTL scheme [14] that will be used in a later section. Finally, we present our motivation for coming up with our scheme.

2.2.1 Trend of Flash Memory Technology

Due to its relatively simple structure and the great demand for higher capacity, NAND flash memory is one of the most aggressively scaled electronic devices. Figure 2.1 shows the

evolution of NAND flash memory technology [4]. The aggressive trend of the decreasing process design rule in NAND flash memory technology effectively accelerates Moore' Law. In late 2011, 20-nanometer NAND process technology was developed and one flash memory chip with a 128GB capacity was released [12]. As the feature size of flash memory cells is close to its minimum limit, further increases in density will be driven by greater levels of MLC, possibly the 3-D stacking of transistors, and improvements to the manufacturing process [4]. With the increasing capacity of flash memory chips, many existing FTLs suffer from a large RAM cost when they are adopted to manage large-scale NAND flash memory storage systems. As a block-level address mapping scheme, NFTL has the smallest RAM footprint among the FTL schemes.

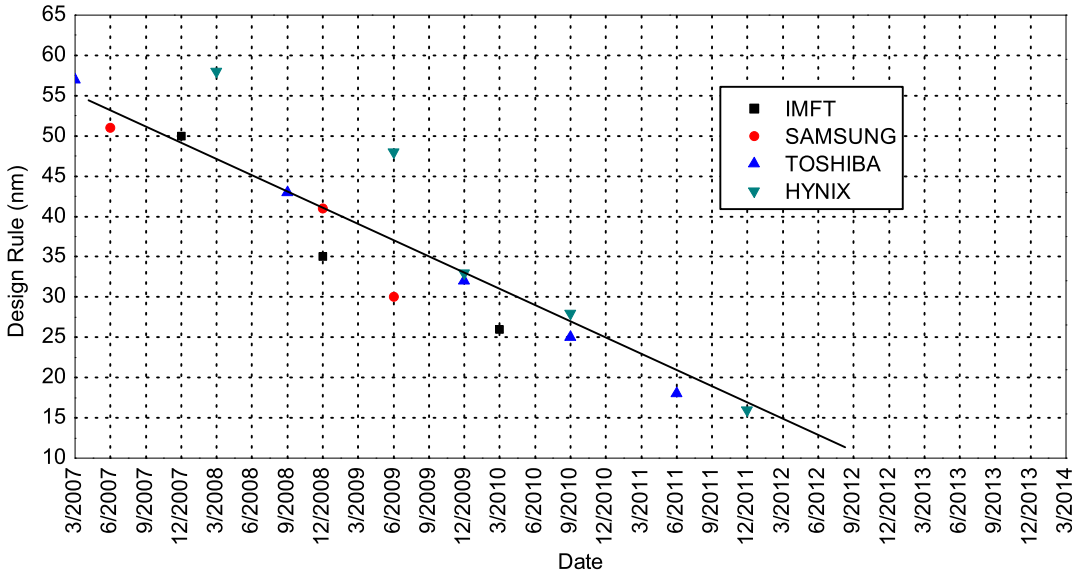


Figure 2.1: The trend in the evolution of NAND flash memory design rules or technology.

2.2.2 Block-level Mapping NFTL

In block-level mapping NFTL [14], one logical block is mapped with two physical blocks, which are called the primary physical block (PPB) and the replacement physical block (RPB), respectively. The primary block is designated to store the first written data, and the replacement block is designed to store the re-written (updated) data. A logical page

number (LPN) in NFTL is partitioned into a logical block number (LBN) and a block offset. Using the logical block number and the block offset, NFTL can obtain the physical page that stores the requested data from the primary block. If the data in the target page becomes invalid (stale), NFTL comes to the replacement block to find the valid data by searching the physical pages sequentially. Figure 2.2 shows an example of the NFTL scheme. Given the LPN 7, divided by the number of pages in each block (i.e., 4), the quotient is the LBN 1, and the remainder is the block offset 3. Using the offset 3 in the primary block 2, NFTL can locate the target physical page 11. Since the target page is invalid, it turns to the replacement block 1 to search sequentially from the first page to find the valid page. As coarse-grained block-to-block mapping is used, NFTL requires a smaller number of address mapping entries compared with page-level FTLs. However, if the file system issues write requests to an identical logical page address, the primary physical block may present a low space utilization ratio and expensive block merge operations (including many page copy operations and block erase operations) may be triggered, which may increase the system response time.

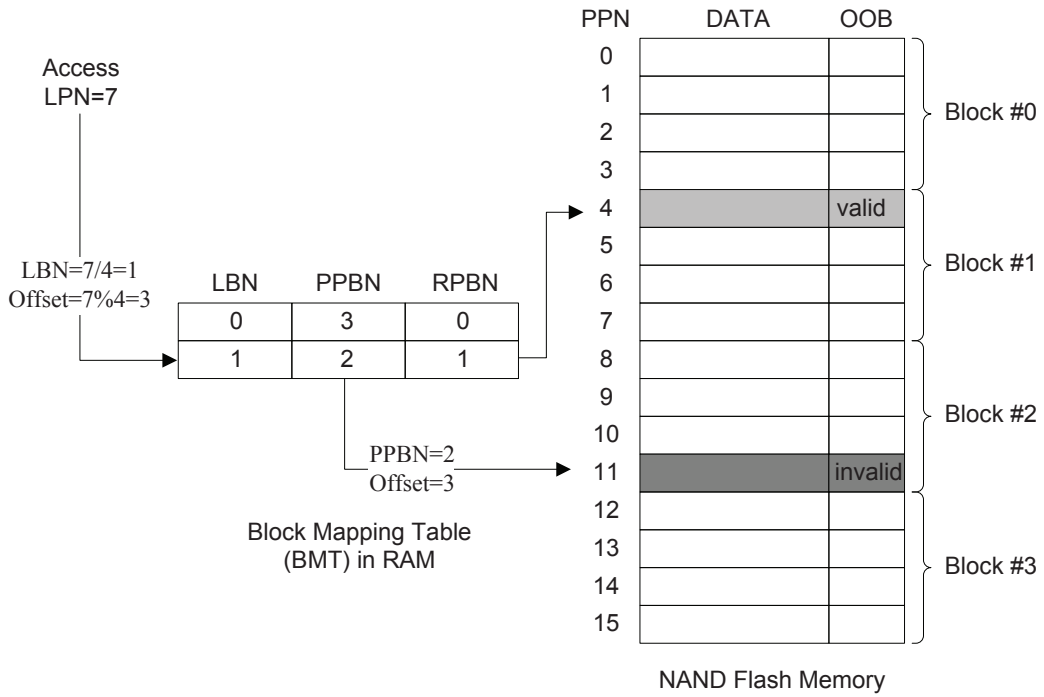


Figure 2.2: An illustration of the block-level mapping NFTL scheme.

2.2.3 Motivation

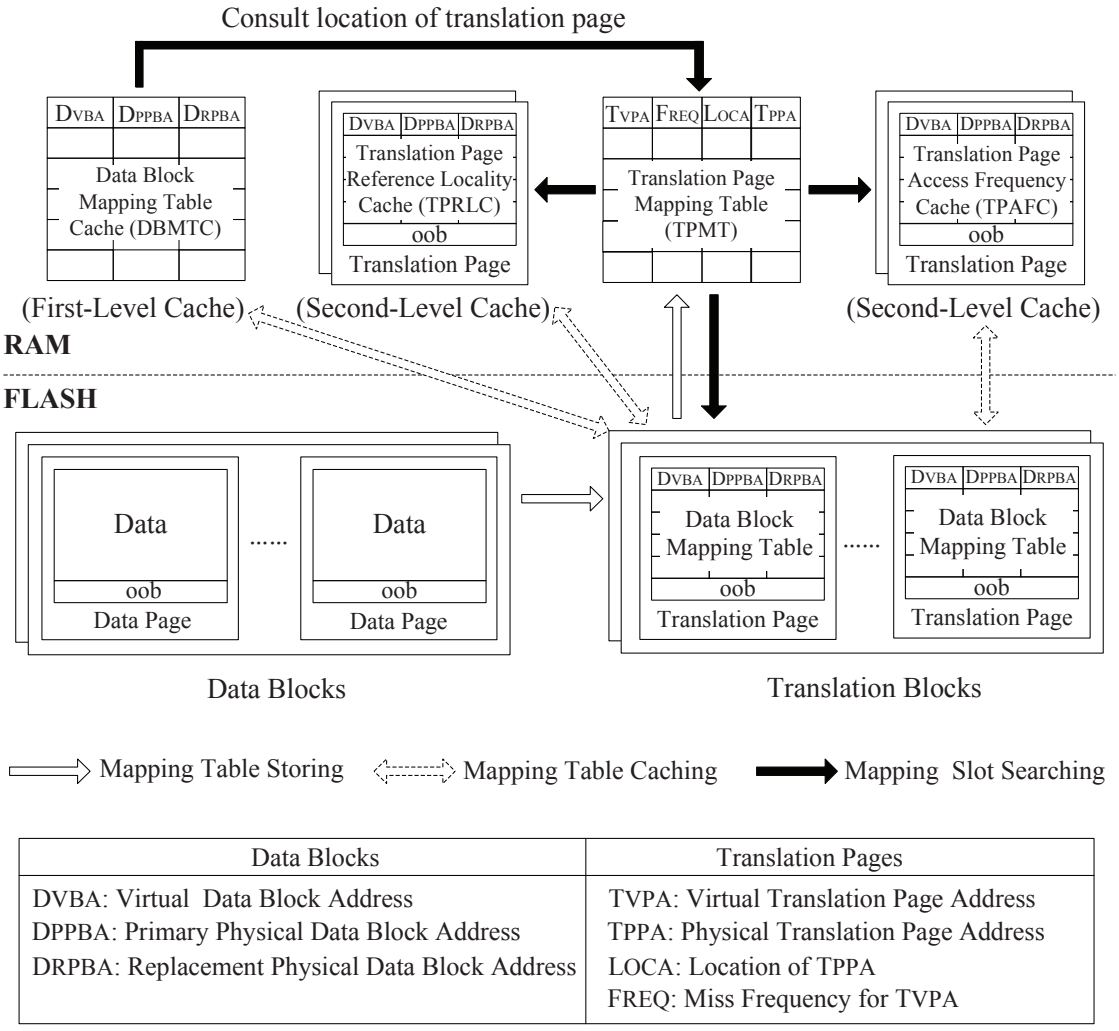
Although the block-level NFTL scheme is effective at reducing the RAM cost, it still suffers from large RAM footprint due to the continuously increasing NAND flash memory capacity. For example, given a large-block (2KB/page) based 32GB Micron NAND flash memory chip MT29F32G08CBABAWP [3], the block-level address mapping table could take up 1.5MB of RAM space. This large RAM footprint limits the application of flash memory in some resource-constrained embedded systems, especially in some low-end storage systems. The DFTL scheme [33] shows potential at solving the RAM cost problem; however, it introduces many extra address translation overheads. Moreover, it ignores the spatial locality and access frequency of workloads, which degrades the system performance. A demand-based block-level address mapping scheme is a promising solution to this problem. These observations motivated us to design an on-demand block-level address mapping scheme that can further reduce the RAM cost without excessively compromising the system performance of large-scale NAND flash memory storage systems.

2.3 DAC: Demand-Based Block-Level FTL

In this section, we introduce our **DAC (Demand-based block-level Address mapping with two-level Caches)** approach. We first give an overview of our scheme in Section 2.3.1. The detailed on-demand address mapping approach and the address translation procedure are then described in Section 2.3.2. Next, we present the fetch-in policy and the kick-out policy for our cache design in Section 2.3.3. Finally, we show the read/write operation and garbage collection procedure in Section 2.3.4.

2.3.1 Overview of DAC

The system architecture of our DAC scheme is shown in Figure 2.3. In DAC, the physical blocks in the flash memory are divided into two types: data blocks and translation blocks. Data blocks, which are dedicated to storing the real data from I/O requests, are managed



Data Blocks	Translation Pages
DVBA: Virtual Data Block Address DPPBA: Primary Physical Data Block Address DRPBA: Replacement Physical Data Block Address	TVPA: Virtual Translation Page Address TPPA: Physical Translation Page Address LOCA: Location of TPPA FREQ: Miss Frequency for TVPA

Figure 2.3: Architecture of DAC.

in the block-level mapping approach. Unlike the traditional method of storing the address mapping table in RAM, DAC stores the entire block-level address mapping table in the pages of translation blocks. The translation blocks, which store the block-level address mapping table, are managed in the fine-grained page-level mapping approach, and the corresponding translation page mapping table (TPMT) is stored in RAM.

As shown by the single-ended arrow with a solid line in Figure 2.3, the data block mapping table is stored in the translation pages, while the translation page mapping table is stored in RAM. Taking advantage of the reference locality and the access frequency of workloads, we designed two-level caches in RAM. The data block mapping table cache

(DBMTC), which serves as the first-level cache, is used to cache the on-demand data block address mapping entries. The second-level cache, which includes two different caches, is designed to cache the translation pages. The translation page reference locality cache (TPRLC) is dedicated to selectively caching those translation pages that contain the on-demand mapping entries in the first-level cache; and another translation page access frequency cache (TPAFC) is designed to cache those translation pages that are frequently accessed when the requested mapping misses in the DBMTC and the TPRLC. Different cache replacement policies are proposed for different caches. The double-ended arrow with a dotted line in Figure 2.3 describes the address mapping table caching procedure, and the arrow with a bold line shows the address mapping entry searching process. A requested address mapping entry will first be searched for in the first-level cache, and then located in the translation blocks by the TPMT in RAM if a cache miss occurs.

2.3.2 Demand-Based Address Mapping with a Two-Level Cache

In this section, we present the data structure and caching mechanism used to implement the demand-based block-level address mapping.

Data Blocks and Translation Pages. In our technique, the data blocks are mapped in a block-level mapping approach, where one virtual data block address (DVBA) is mapped with one primary physical data block address (DPPBA) and one replacement physical data block address (DRPBA). Therefore, one address mapping entry in the block-level address mapping table is represented as <DVBA, DPPBA, DRPBA>. The pages in the translation blocks that are used to store this address mapping table are called *translation pages*. One translation page can store a number of logically fixed address mapping entries. For example, if 8 bytes are needed to represent one address mapping entry, then we can store 256 logically consecutive mapping entries in one translation page. The space overhead incurred by storing the entire block-level address mapping table is negligible compared to the whole flash space. 32GB of flash memory needs only about 1.5MB of flash space to store all of these mapping entries.

Translation Page Mapping Table. In order to track the location of the address mapping table, a translation page mapping table implements the address mapping from one virtual translation page address (TVPA) to one physical translation page address (TPPA). Given the requested virtual data block address (DVBA), divided by the number of mapping entries that each physical translation page can store, the quotient is defined as the virtual translation page address (TVPA). Using the entries in the TPMT, we can immediately locate the physical translation page that stores the requested virtual data block address. Furthermore, one item LOCA in the TPMT is used to record the location of the physical translation page (in cache or in flash memory) for each virtual translation page address, which will also be helpful for locating the translation pages. In the TPMT, another item FREQ is used to record the access frequency of each translation page when the requested mapping misses in the first-level cache and the translation page reference locality cache. The value of FREQ needs to be increased by one if the requested mapping misses in the first two caches. The accumulated value of FREQ indicates the tendency of the corresponding translation page to need to be fetched into RAM from flash memory. Although the TPMT is permanently maintained in RAM, it does not pose many space overhead. For example, for a 32GB flash, 1,024 translation pages are needed, which requires only about 4KB of RAM space.

Data Block Mapping Table Cache. Making use of the temporal locality in workloads, we design the data block mapping table cache in RAM to cache a small number of active mapping entries associated with the on-demand blocks. If the requested mapping hits in this cache, DAC can use or update it immediately without searching for or updating it in the flash memory. If the requested mapping is not stored in the cache and the cache is not yet full, it will be fetched into cache directly once it is found in the flash memory. Otherwise, if the cache is full, one victim mapping must be kicked out to make room for the newly fetched-in mapping, which may lead to an extra translation page copy operation in the flash memory. In order to avoid this extra overhead, we designed a new replacement algorithm taking into consideration both the LRU replacement algorithm and the kick-out overhead (to be explained in a later section). As the first-level cache in RAM, the DBMTC can flexibly be set to different sizes based on the size of the address mapping table that needs to be cached.

For example, it can be set to 16KB, which is only about 1% of the size of the whole mapping table (1.5MB). If one mapping entry takes up 8 bytes, then 2,048 entries are included in the DBMTC. When the active mapping set is large, we adopt a set associative mapping approach (i.e., 2-way or 4-way) for cache organization to guarantee the query efficiency.

Translation Page Reference Locality Cache. The translation page, which stores the on-demand mapping entry that has just missed in the first-level cache, will be selectively cached in the TPRLC. Since the translation page covers a wider spectrum of logically consecutive address mapping entries, according to the spatial locality in workloads, it will be possible for one request to hit in this cache when it misses in the first-level cache. As one part of the second-level cache, the fetch-in operation in the TPRLC is invoked by the fetch-in operation in the first-level cache. When the TPRLC is full, one victim page should be kicked out to make room for the coming fetched-in translation page. The LRU replacement algorithm is adopted as the replacement algorithm in this cache.

Translation Page Access Frequency Cache. The translation page that shows the strongest tendency to be fetched into RAM will be selectively cached in the TPAFC. When the requested mapping frequently misses in the first-level cache and the translation page reference locality cache, it should be fetched into RAM from flash memory in order to guarantee the hit ratio and reduce the address translation overhead. As another part of the second-level cache, the translation page access frequency cache is designed to cache those translation pages that contain frequently requested mapping entries. In such a way, the requested mapping that misses in the first two caches may hit in this cache. The Least Frequently Used (LFU) replacement algorithm is used to evict the victim translation page when the cache is full.

In both levels of cache, a binary *one-bit* tag is designed to indicate whether one item is clean or dirty. The status of this bit can be used in the victim kick-out operation in the two caches. The size of the second-level cache (the TPRLC and the TPAFC) can also be tuned flexibly within the RAM size constraint. For example, 10 translation pages take up about 20KB of RAM space. In terms of cache query efficiency, sequential lookup is sufficient

for searching the requested mapping in the translation pages since the mapping entries are organized consecutively according to the virtual data block address.

Given a request issued from the file system, DAC first searches in the cache for the requested address mapping. If the requested mapping hits in the first-level cache, we can get the requested mapping directly. Otherwise, we need to find the location of the translation page that contains the requested mapping from the TPMT. If the requested mapping is located in the second-level cache, we can find it by searching the cache sequentially. If both two level caches miss and are full, the requested mapping will be fetched into the cache from flash memory and then be used by DAC. Algorithm 2.3.1 shows the procedure of translating from a logical data block address to a physical data block address.

2.3.3 Replacement Policy in a Two-Level Cache

In the first-level cache, the replacement policy is designed based on the LRU replacement algorithm and a cost-benefit analysis. We first select a potential victim mapping based on the LRU replacement algorithm. The victim mapping is then evaluated according to a simple cost-benefit analysis: if the potential mapping is also currently included in the second-level cache currently, it can be the victim; or else, if the potential mapping has not been updated since it was fetched into cache, it can be the victim; otherwise, a new potential mapping will be selected according to the LRU replacement algorithm and be evaluated again. If all of the mapping entries in the first-level cache violate the rules, no victim is selected and no fetch-in operation is performed in the first-level cache. The victim mapping that is selected based on this rule can either be erased directly (if no update occurred on this mapping) or be kicked out to the second-level cache (if the corresponding translation page exists). In both cases, no time-consuming write-back operation towards flash memory is incurred in the first-level cache. Therefore, no extra address translation overhead is caused.

In the second-level cache, the LRU replacement algorithm and the LFU replacement algorithm are used in the TPRLC and the TPAFC, respectively. After the fetch-in operation in the first-level cache, the corresponding translation page should also be loaded into the

Algorithm 2.3.1 Address_Translation($RDlba$)

Input: DBMTC, TPRLC, TPAFC, TPMT, Requested logical data block address $RDlba$, Requested virtual data block address $RDvba$, Victim entry in DBMTC $Evictim$, Page numbers in one block $Npage$, Mapping numbers in one translation page $Mpage$.

Output: Requested primary physical data block address $RDppba$ and Requested replacement physical data block address $RDrppa$.

1: $RDvba \leftarrow RDlba/Npage$; $RTvba \leftarrow RDvba/Mpage$.
2: Search $RDvba$ indexed by $Hash(Dvba)$ in the DBMTC.
3: **if** hit **then**
4: return $RDppba$ and $RDrpba$.
5: **else**
6: Locate $RTvpa$ in the TPMT indexed by $Tvpa$; $RFreq \leftarrow Freq$; $RLoca \leftarrow Loca$; $RTppa \leftarrow Tppa$.
7: **if** $RLoca$ is equal to 0 **then**
8: Search $RTvpa$ indexed by $Tvpa$ in the TPRLC; Get $RDvpa$ indexed by $Dvpa$ in $RTvpa$.
9: **else**
10: $RFreq \leftarrow RFreq + 1$.
11: **if** $RLoca$ is equal to 1 **then**
12: Search $RTvpa$ indexed by $Tvpa$ in the TPAFC; Get $RDvpa$ indexed by $Dvpa$ in $RTvpa$.
13: **else**
14: Read translation page $RTppa$ from flash memory; Get $RDvba$, $RDppba$, $RDrpba$.
15: **if** $the\ DBMTC\ is\ not\ full$ **then**
16: $Fetch\_in(RDvba, RDppba, RDrpba, DBMTC)$.
17: **end if**
18: $Fetch\_in(RTppa, TPRLC, TPAFC)$.
19: **end if**
20: **end if**
21: Return $RDppba$ and $RDrpba$.
22: **end if**

second-level cache. If both caches in the second-level are full, one victim translation page will be selected in each of the two caches. For fetch-in operation, we first consider the TPAFC. If the access frequency for the requested translation page is higher than that of the victim page in the TPAFC, the requested translation page will first be fetched into the TPAFC after kicking the victim page out. If the requested page in the flash memory and the victim page in the TPAFC have the same access frequency, the fetch-in operation is performed based on a simple cost-benefit analysis: the victim page that has not been changed since it was fetched into the TPAFC will be kicked out, and the requested translation page will then be fetched-in; otherwise, the requested translation page will be fetched into the TPRLC. When the access frequency of the requested page is lower than that of the victim page in the TPAFC, the requested page will be finally fetched into the TPRLC after the kick-out operation. Algorithm 2.3.2 shows the procedure of fetching the requested physical translation page $RTppa$ into the second-level cache.

It is noticed that the second-level cache in our scheme not only captures the spatial locality in workloads, but simultaneously serves as the kick-out buffer for the first-level cache. When the requested mapping misses in the first-level cache, no fetch-in operation is performed if no victim is selected based on the cost-benefit analysis as discussed in above section. In this situation, the requested mapping can still be cached in RAM as its corresponding translation page must be fetched into the TPRLC from flash memory. This policy can effectively guarantee the cache hit ratio. Meanwhile, when the victim mapping in the first-level cache is kicked out, it will be evicted to the second-level cache but not the flash memory. The second-level cache actually delays the kick-out operation in the first-level cache and then performs the kick-out operation in bulk. This mechanism can effectively reduce the kick-out operations towards the flash memory and decrease the time-consuming page read or page write operations in the flash memory, which can significantly improve the address translation efficiency.

Next we introduce the kick-out policy in two-level caches. The victim mapping in the first-level cache is either erased directly or kicked out to the second-level cache. Therefore, no flash page operations are triggered. However, an extra flash page read or page write

Algorithm 2.3.2 Fetch_in ($RTppa$, $TPRLC$, $TPAFC$)

Input: DBMTC, TPRLC, TPAFC, TPMT, Victim page in the TPRLC/TPAFC $TRvictim/TAvictm$.

Output: Location of $RTppa$ in the TPRLC or the TPAFC

if $the\ TPRLC\ is\ not\ full$ **then**
  Fetch translation page $RTppa$ into the TPRLC; $RLoca \leftarrow 0$.
else
  if $the\ TPAFC\ is\ not\ full$ **then**
    Fetch translation page $RTppa$ into the TPAFC; $RLoca \leftarrow 1$.
  else
    Select $TRvictim$ in the TPRLC by LRU algorithm.
    Select $TAvictim$ in the TPAFC by LFU algorithm.
    if $RFreq$ is greater than $Freq$ of $TAvictim$ **then**
      $Kick\_out(TAvictim, TPAFC)$; $RLoca \leftarrow 1$; Fetch translation page $RTppa$ into the TPAFC.
    else
      if $RFreq$ is equal to $Freq$ of $TAvictim$ **then**
        if $TAvictim\ is\ not\ changed$ **then**
          Erase $TAvictim$; $RLoca \leftarrow 1$; Fetch translation page $RTppa$ into the TPAFC.
        else
          $Kick\_out(TRvictim, TPRLC)$; $RLoca \leftarrow 0$.
          Fetch translation page $RTppa$ into the TPRLC.
        end if
      else
        if $RFreq$ is smaller than $Freq$ of $TAvictim$ **then**
          $Kick\_out(TRvictim, TPRLC)$; $RLoca \leftarrow 0$.
          Fetch translation page $RTppa$ into the TPRLC.
        end if
      end if
    end if
  end if
end if
Return the location of $RTppa$ in the TPRLC or the TPAFC

operation may be required when doing the kick-out operation in the second-level cache. If the victim translation page has never been updated since it was fetched into the cache, it can be erased directly and a free space will be released. Otherwise, if the victim translation page is changed, it should be flushed back to flash memory in order to maintain the consistency. The victim translation page is written to a new free translation page; meanwhile, the original translation page becomes invalid. The address of the new translation page will correspondingly be recorded in the TPMT. The extra overhead caused by the flush-back operation is only one page write operation in the worst case. Algorithm 2.3.3 shows the procedure of the kick-out operation in the TPRLC, and the kick-out operation in the TPAFC obeys the same rules.

Algorithm 2.3.3 Kick_out ($Pvictim$,$TPRLC$)

Input: TPRLC, TPMT, Victim translation page in the TPRLC $Pvictim$.

Output: Free location in the TPRLC.

1: $PTvpa \leftarrow Pvictim.Tvpa$; $PTppa \leftarrow Pvictim.Tppa$.

2: Sequentially search $PTvpa$ indexed by $Tvpa$ in the TPMT; Get $PTvpa$ and $PT'ppa$.

3: **if** $PT'ppa\ is\ updated$ **then**

4: Write $PT'ppa$ to a new free translation page $FT'ppa$; $PTppa \leftarrow FT'ppa$ in the TPMT.

5: **end if**

6: Erase $Pvictim$ in the TPRLC.

7: Return the free location in the TPRLC.

An illustrative example of the address translation procedure when both levels of cache miss is shown in Figure 2.4. Given the requested logical data page address (DLPA) 65, the corresponding virtual data block address (DVBA) is 1 (65/64=1), where 64 is the number of pages within one physical block. In step (1), the mapping entry of DVBA 1 misses in the first-level cache. In steps (2)-(3), the requested mapping entry is located in physical translation page TPPA 12 in flash memory after consulting the TPMT with the virtual translation page address (TVPA) 0 (1/256=0), where 256 is the number of mapping entries in each translation page. Since translation page TPPA 12 is not cached in the second-level cache, the item FREQ for the requested TVPA 0 in the TPMT needs to be increased by 1, which reaches 3. In steps

(4)-(6), the mapping entry for DVBA 511 in the DBMTC is supposed to be the victim slot, and the requested mapping entry is fetched into the DBMTC after kicking the victim out. In steps (7)-(9), the access frequency of the requested virtual translation page TVPA 0 is obtained from the TPMT with a value of 3, which is smaller than the access frequency of the victim translation page in the TPAFC with a value of 4. The requested translation page TPPA 12 should be fetched into the TPRLC. In steps (10)-(12), the physical translation page TPPA 18 is supposed to be the victim in the TPRLC, and it has been changed compared with corresponding mapping entries stored in flash memory. Therefore, the physical translation page TPPA 18, together with the updated mapping entries, are copied to the free translation page 39. The entry for TPPA 18 in the TPMT is changed to 39 correspondingly. In step (13), translation page TPPA 12 is fetched into the TPRLC after the victim translation page TPPA 18 is kicked out. In step (14), the newly fetched-in mapping entry is the requested one.

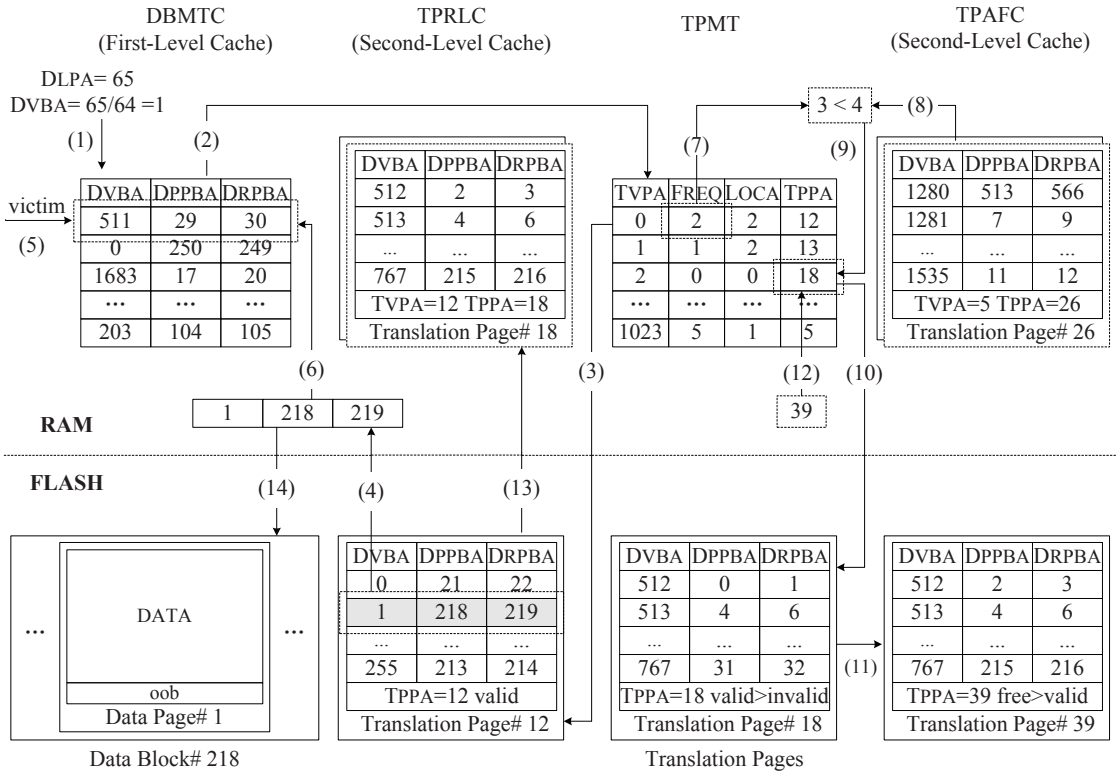


Figure 2.4: Illustration of the address translation process in the DAC scheme with both levels of caches missing.

2.3.4 Read/Write Operation and Garbage Collection

After address translation, the target primary physical block address can be obtained. A read or write operation can locate a data page through the offset in the corresponding primary physical block. If the target data page is invalid or occupied, it is necessary to find the valid page or a free page by searching the replacement physical block sequentially. For a write (rewrite) operation, the new mapping should be updated in the cache correspondingly. If the replacement block or the primary block is full, the block merge operation for the data blocks will be invoked to release free space. The valid pages in both the primary block and the replacement block will be copied to a new free block and the two blocks will then be erased. The page copy operations incur an address mapping change. In order to keep the cache synchronization, the mapping entries in both levels of caches should be updated correspondingly after the garbage collection for data blocks. Since the first-level cache is implemented in a set associative mapping approach while the mapping entries in the second-level cache are sequentially organized, it is easy to locate and update these changed mapping entries.

The garbage collection is triggered when the number of free blocks decreases to a threshold (i.e., 5% of blocks in the whole flash memory). For garbage collection, a fully occupied block with the fewest number of valid pages will be selected as the victim block based on a greedy policy. The valid pages in this block are copied to a new free block and the victim block will then be erased. The page copies may also trigger the mapping update; therefore, the corresponding mapping in the two-level cache and the translation page mapping table should be updated simultaneously. In the block management technique, we maintain one free block pool, which contains the newly erased blocks. One free block will be allocated to serve the garbage collection either for the data blocks or for the translation blocks. The wear-leveling of flash memory is managed automatically by locating the blocks in a round-robin approach. Moreover, in order to reduce the block erase count and improve the space utilization of NAND flash memory, we adopt the reuse-aware strategy [72, 94, 95] when doing the block reclaim operations in our scheme. For the lower space utilization

problem caused by random writes in NFTL, the reuse-aware strategy can efficiently reduce the block erase count, which improves the average system response time.

2.4 The Performance Analysis of DAC

This section provides an analysis of DAC to show how the main-memory (i.e., RAM) cost is reduced and how the average system response time and the cache hit ratio are enhanced. We first analyze the RAM requirements of different FTL schemes. We then discuss the system performance of DAC and DFTL [33] when limited RAM space is available. Finally, we investigate the extra address translation overhead of DAC over NFTL [14].

2.4.1 Improvement in RAM Cost

Table 2.1: RAM cost of different FTLs.

Capacity	NFTL	DAC	Page-level FTL	DFTL
32GB	1.5MB	4KB+S_{cache}	96MB	128KB+S_{cache}
64GB	3MB	9KB+S_{cache}	512MB	1.5MB+S_{cache}

In the design of FTL schemes, main-memory (RAM) cost and storage performance [30] are the two major considerations from the point of view of vendors. In conventional FTL schemes (NFTL and Page-level FTL), RAM cost refers to the storage cost on the address mapping table. In demand-based FTL schemes (DAC and DFTL), the address mapping table is stored in flash memory, and the RAM cost consists of two parts: the global management table size (i.e., the TPMT in the DAC scheme) and the cache size. The mapping table size and the global management table size depend on the page size and the flash capacity. Table 2.1 shows the RAM cost for different FTL schemes with 32GB and 64GB NAND flash memory, respectively. S_{cache} represents the cache size configuration. The cache size can be determined by engineers in the system design phase according to the performance requirement and the cost.

For a 32GB NAND flash memory (2KB-sized page and 128KB-sized block), the RAM footprint for NFTL is 1.5MB. Based on NFTL, our scheme uses a demand-based approach and reduces the RAM cost to 4KB (for the TPMT) plus the cache size. Since the cache size can vary according to the requirements, the RAM cost reduction varies correspondingly following the setup. We use S_{mt} and S_{gt} to represent the size of the address mapping table and the size of the TPMT, respectively. Then, the improvement in RAM cost can be calculated as follows:

$$(1 - \frac{S_{gt} + S_{cache}}{S_{mt}}) \times 100\% \tag{2.1}$$

For example, for a 32GB NAND flash, DAC shows a 95.57% and 91.41% reduction in RAM cost when the cache size is set as 64KB and 128KB, respectively.

The RAM cost for a page-level FTL is 96MB when the flash size is 32GB. Based on a page-level FTL, DFTL stores the 96MB-sized page-level mapping table in translation blocks and sets up a global translation directory in RAM, which takes up 128KB of RAM space. When the flash capacity is increased to 64GB, the global translation directory in DFTL takes up 1.5MB of RAM space. These figures indicate that the DFTL scheme will be unable to work when the RAM space is less than 1.5MB, but that DAC can work well as long as the RAM space is marginally larger than 9KB. Our scheme shows a lower RAM space requirement with better scalability.

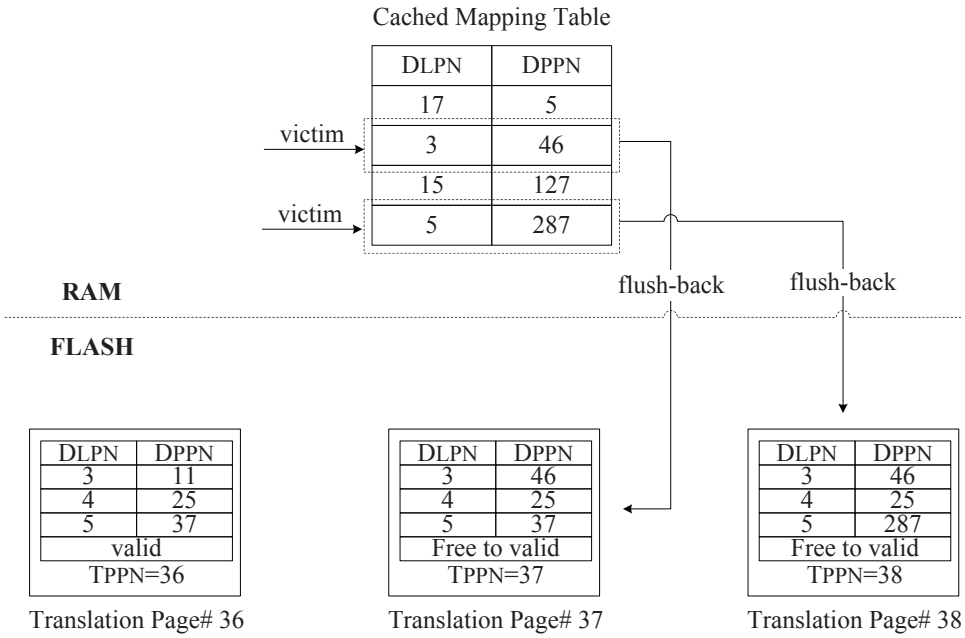
2.4.2 Improvement in Cache Hit Ratio

The cache hit ratio is critical in determining the address translation efficiency. If the requested mapping hits in the cache, the address mapping information can be obtained directly. The time overhead in this situation is the cache operations (cache read, cache write, cache searching, etc.), which is only about $5\mu s$ in real applications. If the requested mapping misses in the cache, the address mapping information needs to be read from flash memory. The time overhead is at least one flash page read operation, which is about $30\mu s$. Therefore, having a good cache management mechanism is very important.

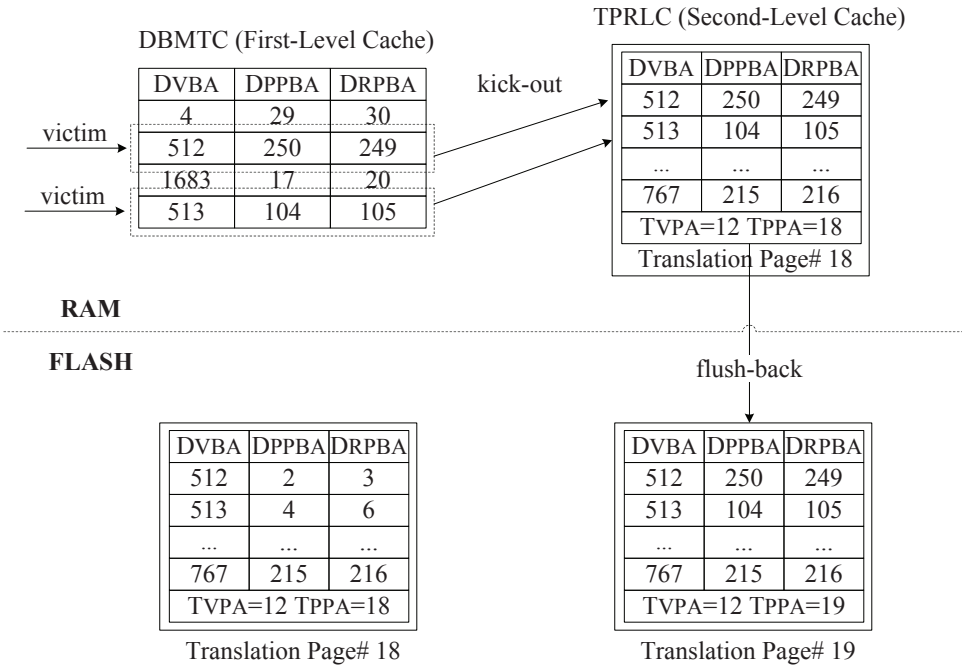
Cache size is one critical factor that has an impact on the cache performance. Given two caches with the same size and the same replacement algorithm, we can get a higher hit ratio if more cache items can be included. In the DFTL scheme, each entry is one page-level address mapping. In the DAC scheme, each entry in the first-level cache is one block-level address mapping. The block-level mapping takes up much less space than the page-level mapping. Therefore, more items can be maintained in the cache of DAC than in that of DFTL. Moreover, one page-level mapping only represents the mapping information of one page. One block-level mapping can cover the mapping information of 64 pages if one block includes 64 pages. The cache hit ratio in DAC may be 64 times higher than that in DFTL.

The replacement algorithm is another important factor affecting the cache hit ratio. A good replacement policy should not only capture the temporal locality of workloads, but also the spatial locality and access frequency of workloads. DFTL adopts a one-level cache that captures the temporal locality well. In DAC, we use a two-level caching mechanism that captures both the reference locality and the access frequency of workloads. For each fetch-in operation in DFTL, one page-level mapping is read into the cache from flash memory, with one page read time overhead. For each fetch-in operation in DAC, one block-level mapping is read into the first-level cache. The corresponding translation page will also be read into the second-level cache, and the time overhead is the same as that of DFTL. Thus, according to the spatial locality in workloads, the coming request will hit in the second-level cache in DAC, while a cache-miss may happen in DFTL. The DAC scheme should have a higher cache hit ratio than that of the DFTL scheme under the same time overhead on cache replacement.

A lower cache hit ratio leads to more translation pages being flushed from cache to flash memory. Thus, more flash space is consumed and more garbage collection will be triggered, so that more block erase operations will be performed. In the DFTL scheme, one page-level mapping table cache is adopted. One flush-back operation consumes one free translation page. However, in the DAC scheme, more than one flush-back operation may consume one free translation page. This is because the second-level cache serves as the kick-out buffer of the first-level cache, and the second-level cache can do the flush-back



(a) DFTL scheme



(b) DAC scheme

Figure 2.5: Illustration of kick-out operations in the DFTL scheme and the DAC scheme.

operation in batches. Figure 2.5 (a) shows an illustrative example of kick-out operations in the DFTL scheme. Two victim mapping entries in the CMT lead to the consumption of two free translation pages (translation page #37 and translation page #38). In Figure 2.5 (b), two victim mapping entries in the DBMTC in the DAC scheme are evicted to the TPRLC, and one free translation page (translation page #19) is used. Therefore, the DAC scheme has a higher space utilization ratio and lower translation block erase counts than the DFTL scheme.

2.4.3 Extra Address Translation Overhead

In conventional FTL schemes, where the address mapping table is stored in main memory (i.e., RAM), the address translation overhead is the time cost of the mapping table searching in RAM. However, in demand-based FTL schemes, where the address mapping table is stored in flash memory, besides the overhead of cache operations in RAM, some extra overhead on the address translation procedure are introduced if the requested mapping table is not maintained in the cache. The extra overhead is the time overhead to fetch (read) the address mapping table from flash memory as well as the potential dirty mapping entries kick-out (write-back) overhead from cache to flash memory.

In the DAC scheme, the extra overhead is the same as that of the NFTL scheme when mapping entries hit in the cache. In the cache-miss situation, the extra overhead differs according to the status of the victim translation page in cache. If the victim translation page is clean, the time overhead is one translation page read (T_{rd}) operation (for the fetch-in operation). Otherwise, if the victim translation page is dirty, the time overhead is one translation page write (T_{wr}) operation (for the kick-out operation) and one translation page read (T_{rd}) operation (for the fetch-in operation). In order to reduce the extra address translation overhead, the DAC scheme considers both the reference locality and the access frequency of workloads. Moreover, the proposed cost-aware LRU replacement algorithm gives clean pages a higher priority to be victim translation pages, so that fewer translation page write operations need to be introduced.

2.5 Evaluation

To evaluate the effectiveness of the proposed scheme, we conduct a series of experiments and present the experimental results with discussions in this section. We compare and evaluate our proposed DAC scheme over NFTL [14] and DFTL [33] in terms of address translation overhead, average system response time, and cache hit ratio. In this section, we first introduce the experimental setup. We then present the experimental results with discussions.

2.5.1 Experimental Setup

Table 2.2: Experimental setup.

Notebook Configuration	CPU	Intel Dual Core 2GHz
	Disk Space	200GB
	RAM	2GB
	DiskMon Traces	CopyFiles, DownFiles, Web Applications, Office, P2P, Media Player
Simulation Environment	OS Kernel	Linux 2.6.17
	Flash Size	32GB
	Simulator	NAND Flash Simulator

Table 4.3 summarizes the experimental setup. We developed a trace-driven NAND flash simulator under Linux 2.6.17 and implemented three schemes: DAC, NFTL, and DFTL. A 32GB NAND flash memory with 2KB-sized page and 128KB-sized block is simulated. To conduct a fair comparison with different FTL schemes, we consider only a portion of flash as the active region that stores our workloads. The remaining flash is assumed to contain cold data or free blocks, which are not under consideration. The framework of our simulation platform, as shown in Figure 4.7, consists of two modules: a NAND flash simulator module providing basic read, write, and erase capabilities; and a desired flash translation layer management scheme that can be executed on top of the NAND flash simulator. The traces,

along with various flash parameters, such as block size and page size, page read time, page write time, and so on, are fed into our simulation framework. Based on the flash memory data sheet [3], the time cost to read a physical page, write a physical page, erase a physical block, read the spare area of a physical page, and search one item in cache, are set as the time consumption for one flash page-read and one flash page-write operation, and are set as 36.6μs, 226.7μs, 2000μs, 0.8μs, and 2μs, respectively.

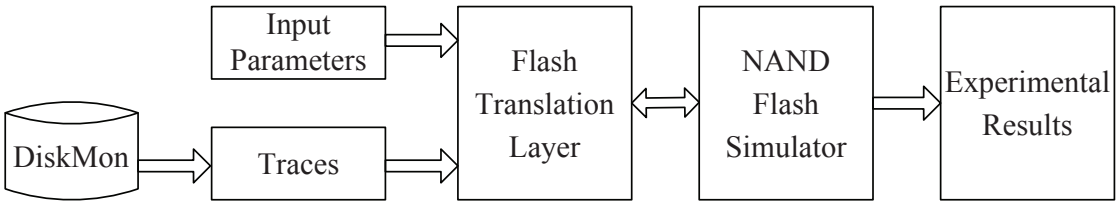


Figure 2.6: The framework of the simulation platform.

The traces with data requests used in this simulation are collected by running DiskMon [1] in Windows XP over a notebook with an Intel Dual Core 2GHz processor, a 200GB hard disk, and a 2GB DRAM. The traces reflect the real workload of the system in accessing the hard disk with applications that are used daily, such as those for web surfing, document typewriting, downloading, and playing movies and games. For each trace, the numbers and percentages of read and write operations are listed in Table 2.3. Trace 1, Trace 2, and Trace 5 are write-dominant applications, while Trace 3 and Trace 4 are read-dominant applications. Trace 6 owns similar percentages of read requests and write requests. The percentage of sequential operations indicates the access pattern of traces in terms of the arrival sequence of requests.

In the simulation, different RAM size configurations are simulated in order to explore the relationship among RAM size, cache hit ratio, and average system response time. We first consider that the three caches are all of the same size. Since the size of one flash page is 2KB, the size of each cache is initialized to 2KB, and the three caches initially take up 6KB of RAM space . The sizes of the three caches are then increased incrementally and the total cache size finally reaches 252KB. The RAM cost consequently becomes 256KB, since the translation page mapping table takes up 4KB in our scheme. In the DFTL scheme, the

Table 2.3: The characteristics of the traces.

	# of writes	# of reads	% of writes	% of reads	% of sequential operations
Trace 1	15,860,736	1,040,896	90%	10%	99.96%
Trace 2	8,198,708	2,471,266	77%	23%	53.50%
Trace 3	2,416,100	17,842,716	12%	88%	99.99%
Trace 4	639,193	8,518,437	7%	93%	50.01%
Trace 5	9,208,655	4,899,133	65%	35%	94.98%
Trace 6	8,903,616	10,906,320	45%	55%	97.91%

global translation directory takes up 128KB of RAM space. The cache size is initialized to 4KB and then increased to 128KB, which boots the RAM cost to 256KB. In the simulation, we first collect the experimental results of our scheme under eight RAM size configurations starting from 10KB to 136KB with an interval of 18KB. In order to make a comparison with DFTL, we then run the simulation with four RAM size configurations for both two schemes starting from 136KB to 256KB with an interval of 30KB.

2.5.2 Results and Discussion

Results of DAC and NFTL

In this section, we compare and evaluate our proposed DAC scheme over the representative block-level flash translation layer scheme NFTL [14] in terms of two performance metrics: the average system response time and the extra address translation overhead. We first conducted experiments to show how the RAM size influences the average system response time. The results, in which the average response time of each trace can be obtained by varying the RAM size from 10KB to 256KB, are shown in Figure 2.7.

From Figure 2.7, we can see that, although the plots for different traces are different, they all show the same trend: the average response time decreases as RAM size increases.

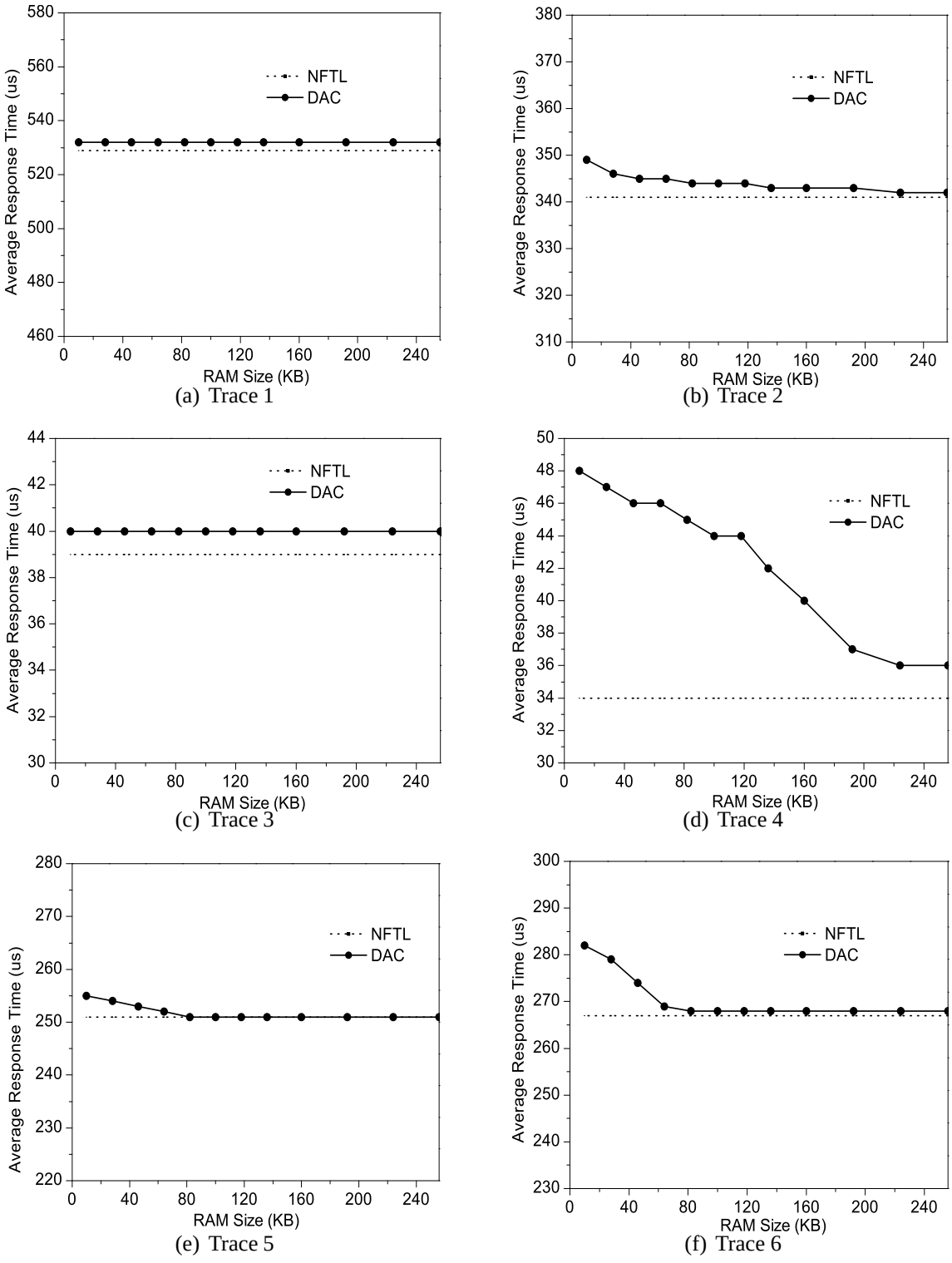
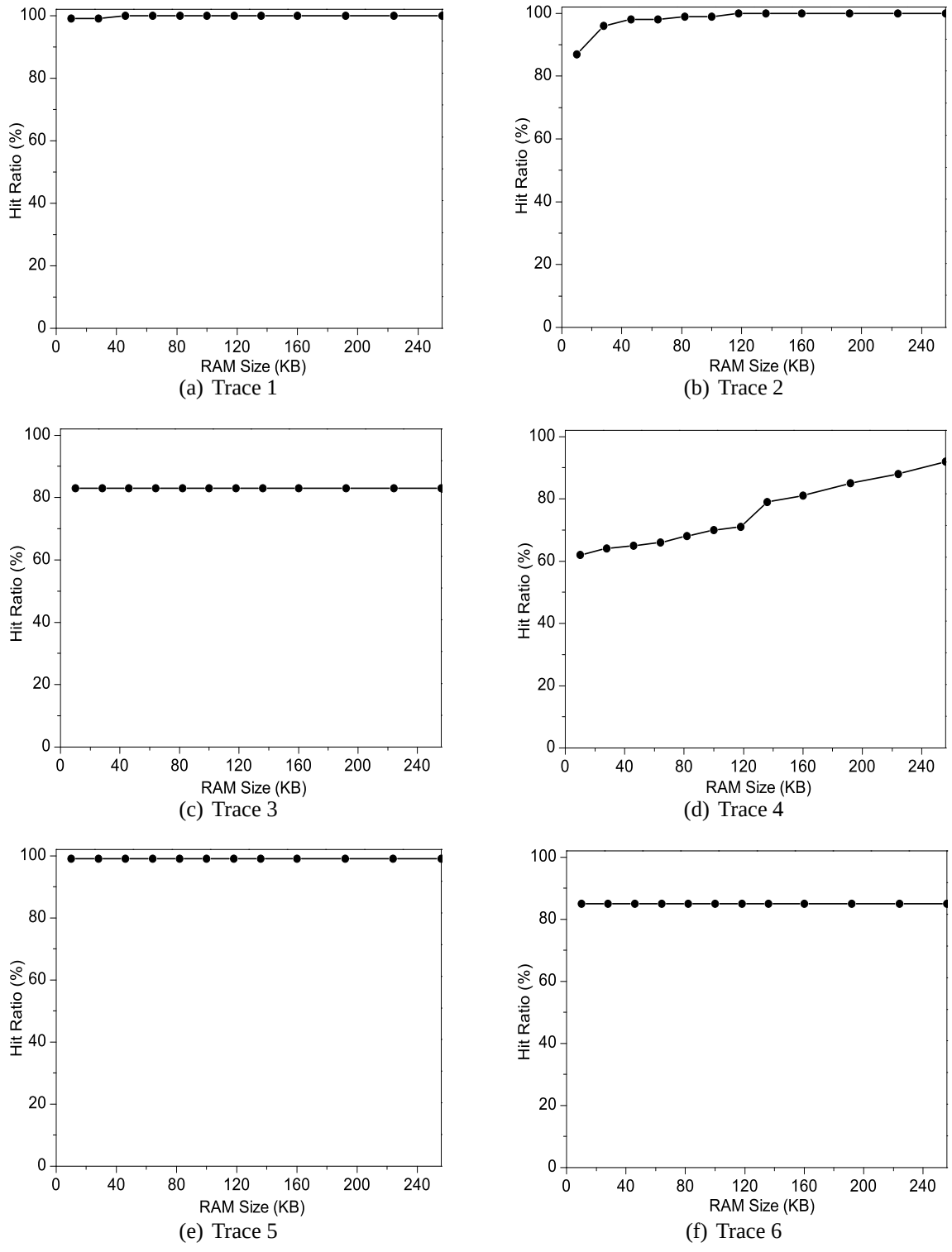


Figure 2.7: The average system response time for DAC and NFTL with different RAM size configurations over six traces from Trace 1 to Trace 6.



(a) Trace 1 (b) Trace 2

(c) Trace 3 (d) Trace 4

(e) Trace 5 (f) Trace 6

Figure 2.8: The cache hit ratio of DAC with different RAM size configurations over six traces from Trace 1 to Trace 6.

When the RAM size reaches 64KB, which takes 8.32% of the entire active mapping table size (768KB) in flash memory, the average response time of all traces are very close to or even the same as that of the NFTL scheme. Therefore, on average, DAC reduces the RAM cost by 91.68% under a 2.02% penalty to the average system response time compared with the NFTL scheme. For Trace 1 and Trace 3, the average response time shows no change when the RAM size varies from 10KB to 256KB. This is because the requests in Trace 1 and Trace 3 present much more sequential patterns. Their requested mapping can hit in a very small cache, and the response time can be significantly reduced even with a small cache size configuration. We can also see that the average response time of Trace4 in DAC is much longer than that of the NFTL scheme when the RAM size is small. This is because the response time of a read request at least doubles when a cache-miss happens. The read-dominant requests with a higher random access pattern in Trace 4 leads to a lower cache hit ratio, which increases the average response time.

Next, we present the experimental results in terms of the cache hit ratio and the extra address translation overhead. Based on the NFTL scheme, the DAC scheme introduces translation blocks and a caching mechanism for managing address mapping. Therefore, the extra overhead of DAC includes all of the operations on the translation blocks, such as the translation page read count, the translation page write count, and the translation block erase count. These extra overheads are incurred by the kick-out operation and the fetch-in operation in the cache, which are determined by the cache hit ratio.

Figure 2.8 presents the cache hit ratio in our scheme when the RAM size is limited to 256KB. The cache hit ratio is influenced by both the cache size and the reference locality of workloads. Traces with more sequential access pattern have more spatial locality. Traces with frequent update operations should present more temporal locality. This can be verified by the results of Trace 1 and Trace 5, shown in Figure 2.8. When the RAM size is set as 10KB, the hit ratio almost reaches 100% for these two traces. For Trace 2 and Trace 4, the cache hit ratio increases when the RAM size is increased. The trend of increase stops at one point and no further improvement can be achieved. This is because the reference locality has been entirely captured, and no more benefit can be obtained with the increased cache size.

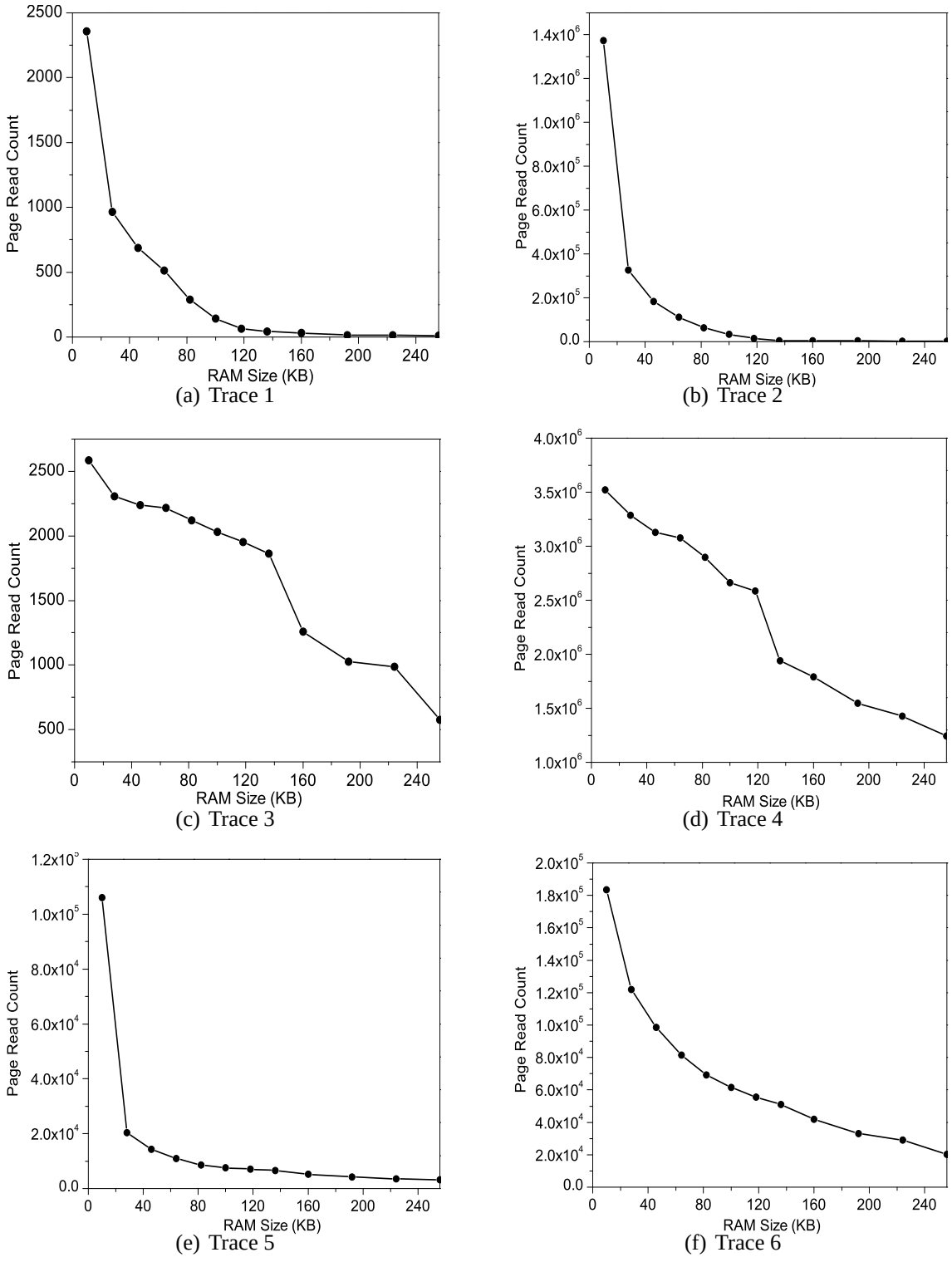


Figure 2.9: The number of translation page read in the DAC scheme with different RAM size configurations over six traces from Trace 1 to Trace 6.

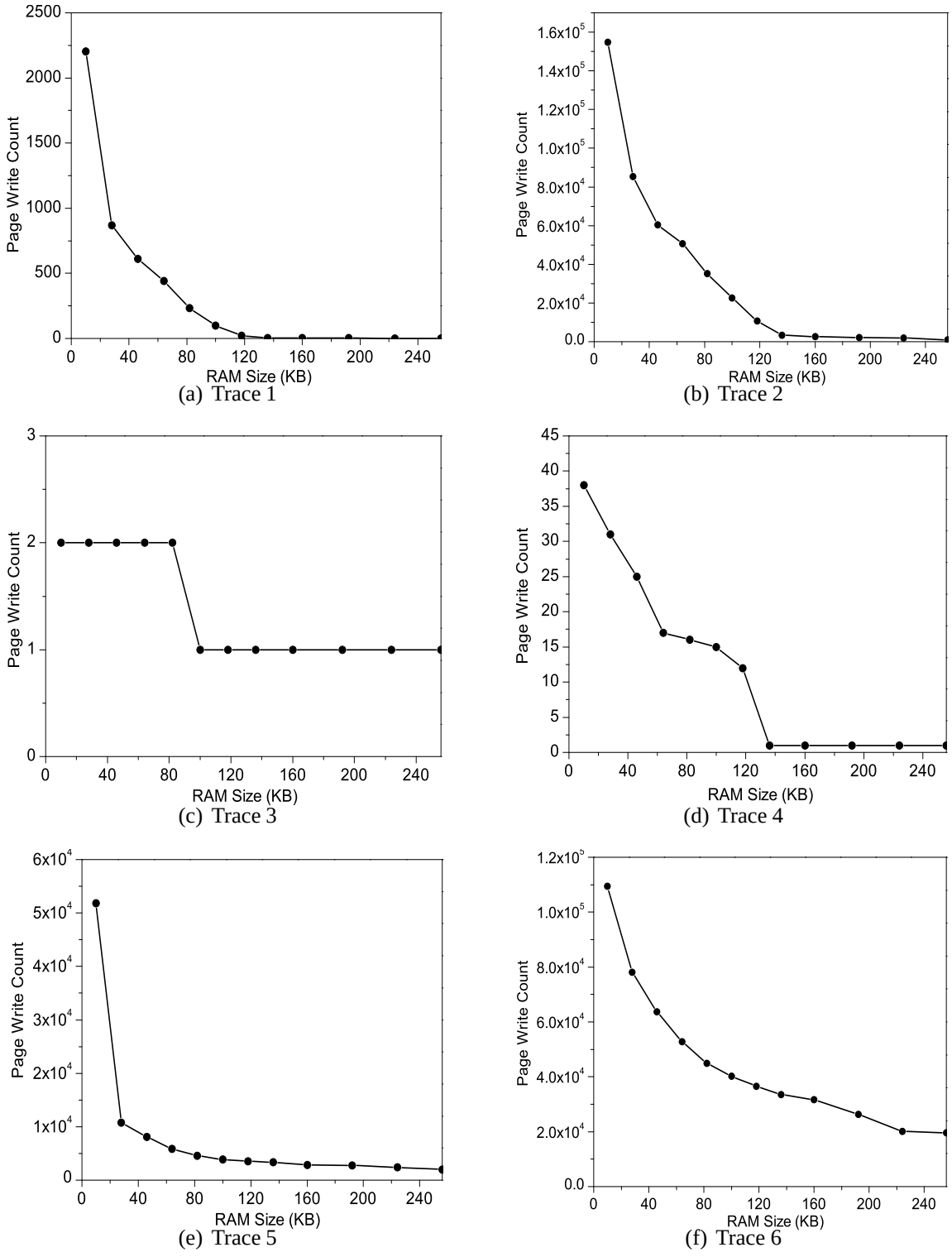


Figure 2.10: The number of translation page write in the DAC scheme with different RAM size configurations over six traces from Trace 1 to Trace 6.

A higher cache hit ratio leads to fewer fetch-in operations in cache, and fewer translation page read operations are triggered. Otherwise, a lower hit ratio will cause more translation page read operations and more translation page write operations. Figure 2.9 and Figure 2.10 show the experimental results of a translation page read count and translation page write count, respectively. Due to the increasing hit ratio in cache, both the translation page read count and the translation page write count for all of the traces are decreased. For Traces 1, 3, 5, and 6, the hit ratio is constant but the translation page read count and the translation page write count is decreased. This is because our scheme adopts a two-level caching mechanism in which three caches are involved. The second-level cache serves as the kick-out buffer for the first-level cache. When the size of the second-level cache is increased, more victim translation pages in the first-level cache are evicted to the second-level cache and fewer victim translation pages are flushed back to flash memory. Therefore, fewer translation page write operations are needed, and fewer translation pages are consumed. In particular, for the read-dominant Trace 2, only 2 page write operations are triggered, and this number is reduced to 1 with the increased cache size.

Results of DAC and DFTL

In this section, we compare and evaluate our proposed DAC over DFTL in terms of the cache hit ratio with different RAM size configurations over six traces. We also compare the average system response time and the extra address translation overhead of the two schemes under the same scenario in which a limited amount of RAM space is given.

Figure 2.11 shows the cache hit ratio for DAC and DFTL under the same RAM size configurations. In both schemes, the cache hit ratio shows the same trend of increase when the cache size varies from 10KB to 256KB. Since the GTD takes up 128KB of RAM space in DFTL scheme, the DFTL scheme cannot work when the RAM size is smaller than 128KB. In Figure 2.11, we only show the cache hit ratio of DFTL when the RAM size is larger than 128KB. From the figure, we can see that the cache hit ratio of DFTL increases when the cache size increases. However, the cache hit ratio for DAC is much higher than that of DFTL when

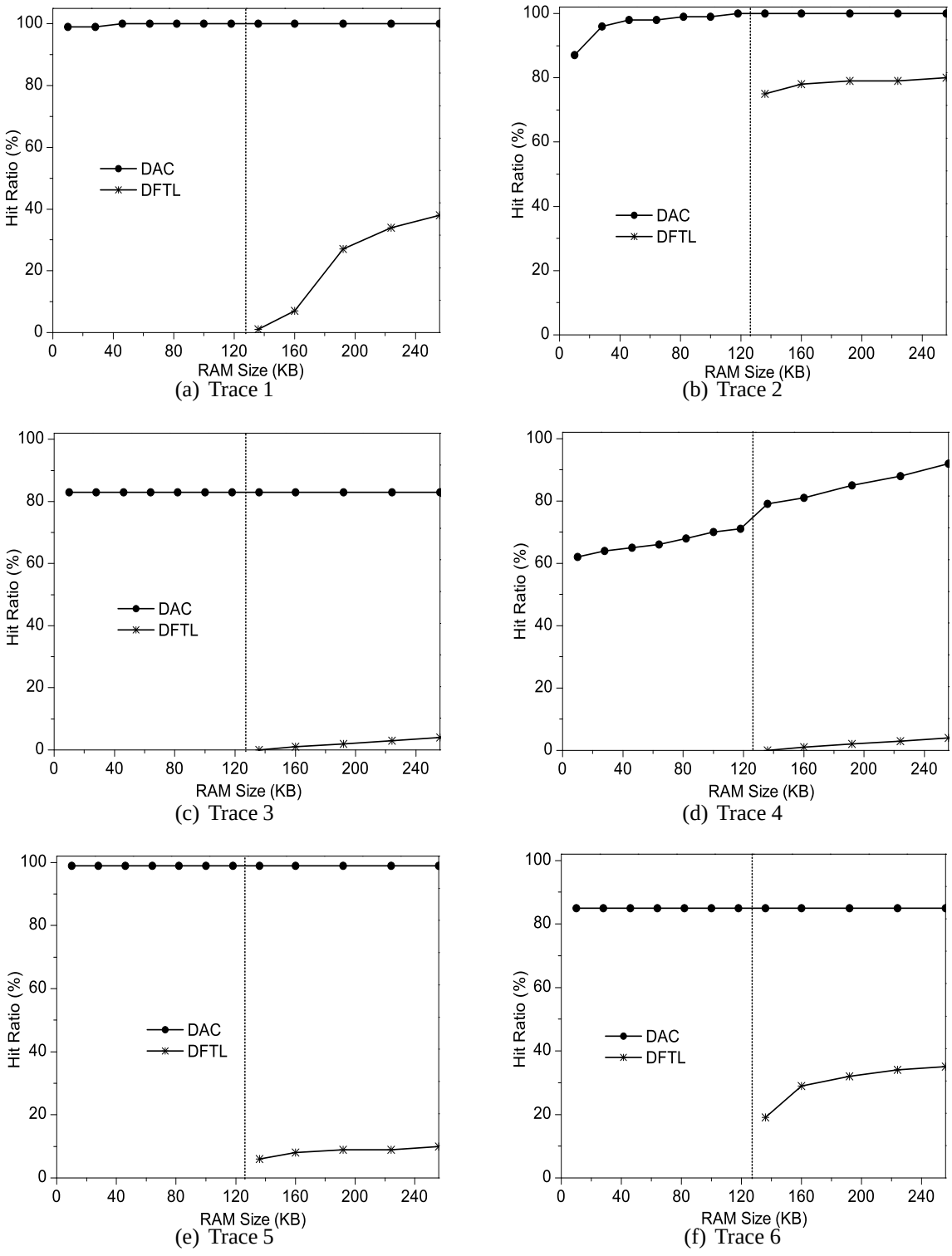


Figure 2.11: The cache hit ratio of DAC and DFTL with different RAM size configurations over six traces from Trace 1 to Trace 6.

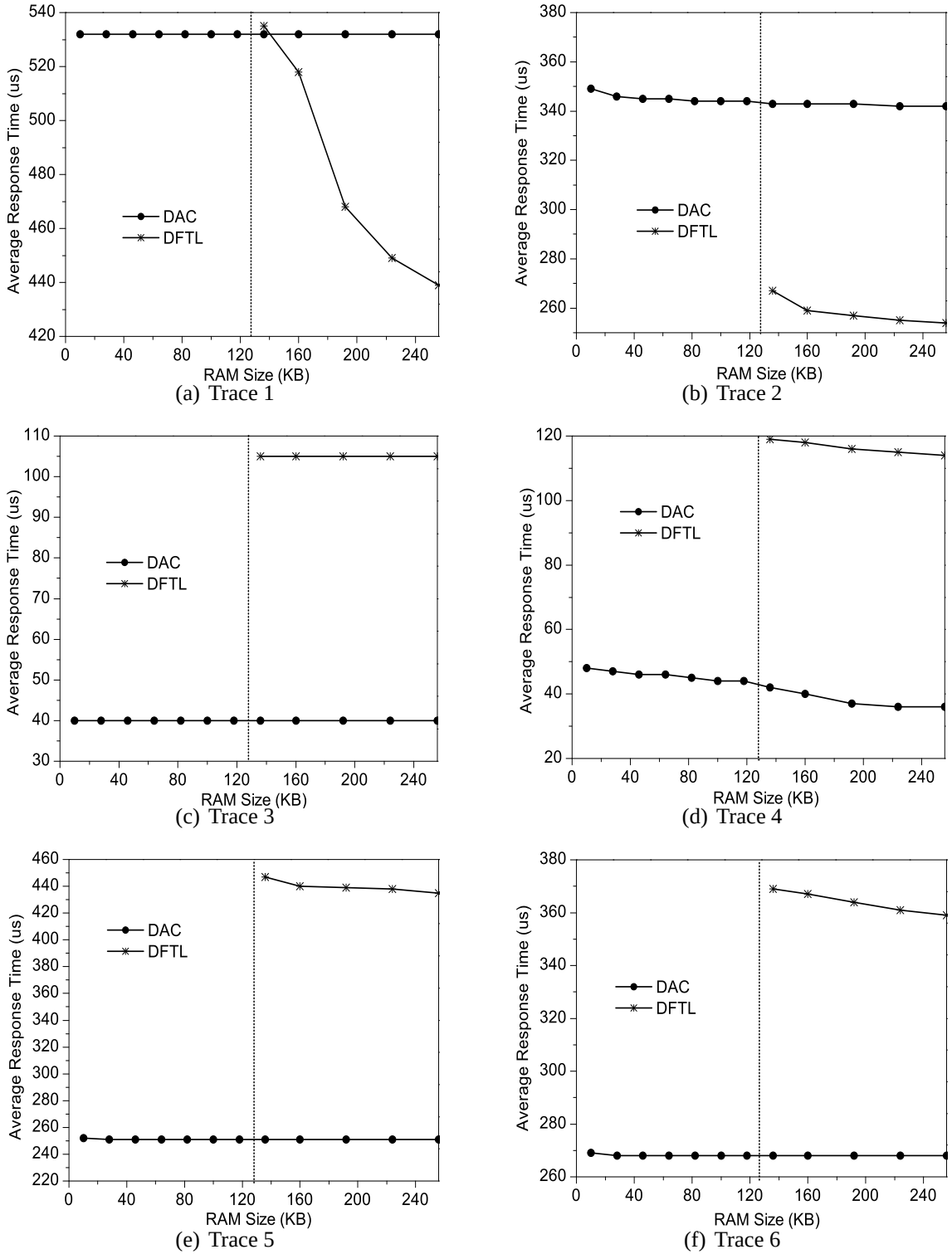


Figure 2.12: The average response time for DAC and DFTL with different RAM size configurations over six traces from Trace 1 to Trace 6.

they have the same RAM size configuration. The improvement in the cache hit ratio in our scheme comes from the larger mapping granularity and the two-level caching mechanism. In our scheme, we use the coarse-grained block-level mapping approach while DFTL adopts a fine-grained page-level mapping approach. Our scheme has a much smaller mapping table than DFTL, and more mapping items will be cached under the same cache size. Another factor that influences the cache hit ratio is the cache design. In our scheme, two-level caches are designed, and both temporal locality and access frequency are considered. In DFTL, the one-level cache only takes into account temporal locality.

Page-level FTL is considered to have a better response time than block-level FTL. We compare the average system response time of page-level mapping-based DFTL scheme with our block-level mapping-based DAC scheme. Figure 2.12 presents the average response time of these two schemes. DFTL shows a better average response time for write-dominant traces (Trace 1 and Trace 2) and a worse average response time for read-dominant traces (Trace 3 and Trace 4) compared with the DAC scheme. This is based on the fact that the page-level FTL scheme triggers garbage collection later than the block-level FTL scheme. DAC scheme can achieve benefit from the improvement in the cache hit ratio; however, the benefit cannot counteract the penalty caused by the earlier-triggered garbage collection overheads. For read-dominant traces, the higher hit ratio in DAC can reduce the average response time, since fewer block erase operations are involved in both DAC and DFTL.

Table 2.4 presents the cache hit ratio and the average response time of these two schemes when the RAM size is 132KB. From the results, we can see that the DAC scheme has a higher cache hit ratio (74.31% higher on average) than the DFTL scheme. For most of the traces, our scheme shows a faster average response time. However, it shows a 28.46% slow-down in the average response time for Trace 2 when compared with DFTL. This is because Trace 2 is a write-dominant trace that has far fewer sequential writes than other write-dominant traces (i.e., Trace 1 and Trace 5). For random requests, the block-level FTL triggers more garbage collection overhead than the page-level FTL. This can be verified from the results shown in Table 2.5. Considering all of the traces, DAC achieves a 27.65% reduction in average response time compared with the DFTL scheme. Therefore, the demand-

Table 2.4: Performance for DAC and DFTL with 132KB RAM.

	Hit Ratio (%)			Average System Response Time (μs)		
	DAC	DFTL	improvement	DAC	DFTL	improvement (%)
Trace 1	99.99	0.23	99.76	532	535	0.56
Trace 2	99.95	74.79	25.16	343	267	-28.46
Trace 3	82.75	0.02	82.73	39	105	62.85
Trace 4	78.84	0.07	78.77	42	114	63.15
Trace 5	99.58	5.83	93.75	251	435	42.29
Trace 6	84.69	18.96	65.73	268	360	25.55
Average			74.31			27.65

based block-level mapping scheme outperforms demand-based page-level mapping scheme when limited RAM size is provided. Table 2.5 shows the address translation overhead of these two schemes. In the table, columns “D.Copy,” “D.Era.,” “T.Rd,” and “T.Wr” represent the copy count of data pages in the garbage collection procedure, the data block erase count, the translation page read count, and the translation page write count, respectively. “D.Copy” and “D.Era.” indicate the garbage collection overhead on data blocks while “T.Rd,” “T.Wr” and “T.Era.” describe the address translation overhead caused by the on-demand address mapping approach. These five metrics directly determine the average system response time.

From the results shown in Table 2.5, we can see that the DFTL scheme shows much less overhead than the DAC scheme in terms of garbage collection on data blocks. For example, the data block copy count and the data block erase count in DFTL are an order of magnitude smaller than that of DAC for most traces. This is based on the fact that the fine-grained page-level mapping scheme has a higher space utilization ratio than the coarse-grained block-level mapping scheme. DFTL triggers the garbage collection of data blocks much later than the DAC scheme. However, DAC has much less overhead on the translation block operations. This can be proved by the fact that the translation block erase counts for six traces in DAC scheme are zero. This overhead reduction comes from the higher cache

Table 2.5: Overhead for DAC and DFTL with 132KB RAM.

Traces	DAC scheme					DFTL scheme				
	D.Copy	D.Era.	T.Rd.	T.Wr.	T.Era.	D.Copy	D.Era.	T.Rd.	T.Wr.	T.Era.
Trace1	8.9E6	2.8E5	43	0	0	0.1E5	1.2E5	2.0E7	0.9E7	1.E5
Trace2	4.2E6	2.1E5	5,007	0.3E4	0	3,592	9,210	0.5E7	0.2E7	0.3E5
Trace3	0	0	1,864	1	0	0	0	4.0E7	17,065	0
Trace4	94	54	9.0E5	1	0	0	0	1.8E7	10,813	0
Trace5	3.5E6	1.1E5	6,555	0.3E4	0	4.0E5	0.7E5	2.6E7	1.0E7	1.5E5
Trace6	6.8E6	2.5E5	0.5E5	3.3E4	0	9.0E5	1.1E5	3.2E7	1.1E7	1.1E5
Average	3.9E6	1.4E5	3.3E5	0.E4	0	2.2E5	0.5E5	2.4E7	0.5E7	0.7E5

hit ratio, which significantly decreases the write operations in the translation blocks. The reduced overhead enables DAC to have better system performance than DFTL when the RAM size configuration is very small.

Impact of Cache Size Allocation

For the above experiments, the size of each cache is fixed and equal to each other. To explore the impact of performing different cache size allocations performed on the system performance, we run the simulation with more cache size configurations. Since three caches are involved, we fix two cache sizes while varying the size of the other one. We assume that the whole cache size is 64KB, and the cache size of each of the two caches is 16KB, while the size of the other cache varies from 0KB to 32KB. Two traces, Trace 2 and Trace 6, are taken since they represent the random-dominant trace and the sequential-dominant trace, respectively. Figure 2.13 shows the cache hit ratio and the translation page read and translation page write overheads of our scheme. In the figure, Cache-I means the first-level cache (i.e., the DBMTC), while Cache-II(1) and Cache-II(2) represent the first cache (i.e., the TPRLC) and the second cache (i.e., the TPAFC) of the second-level cache, respectively.

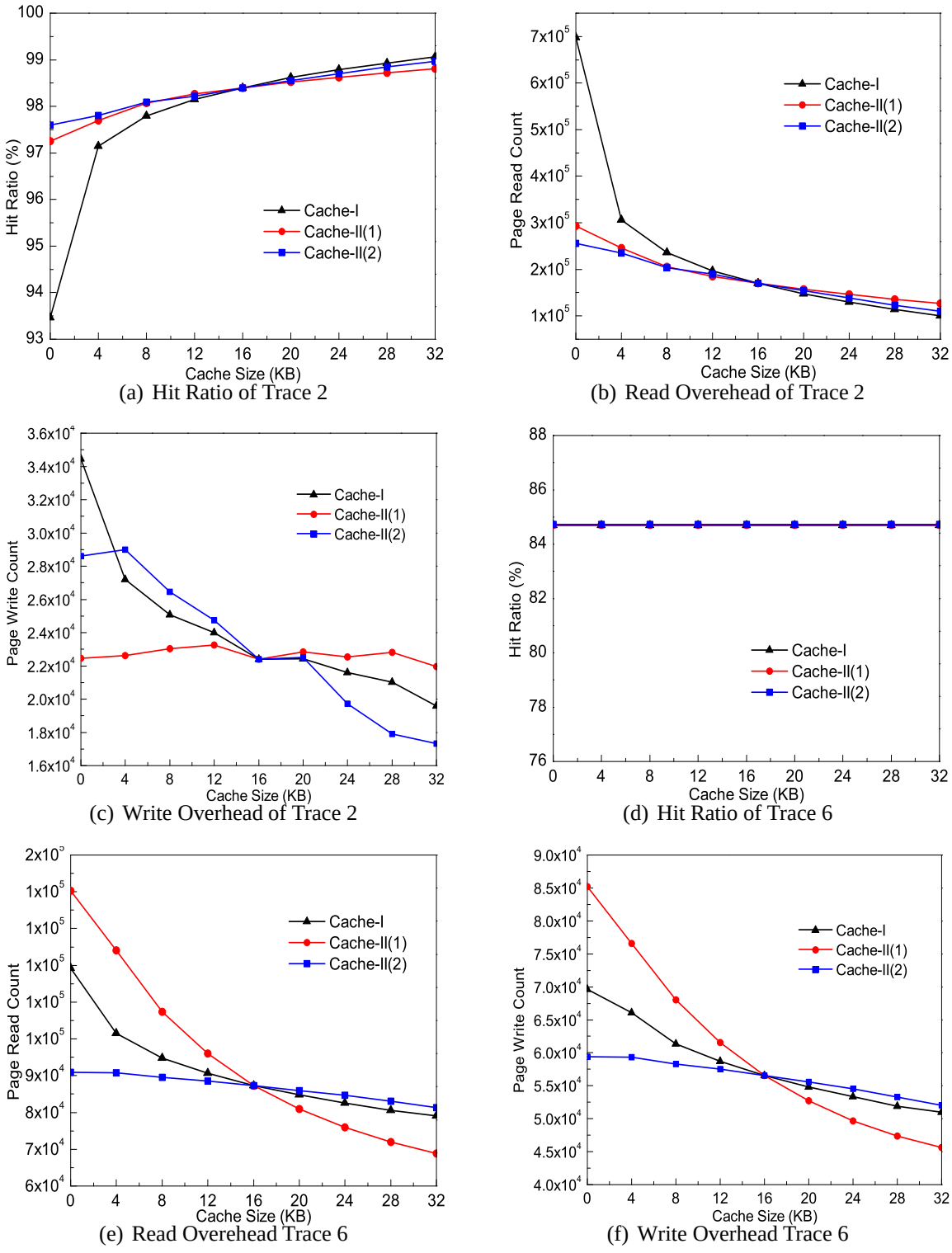


Figure 2.13: Performance of the DAC scheme with different cache size configurations over Trace 2 and Trace 6.

The X-axis in each figure means that the cache size of one cache (i.e., Cache-I) is increasing from 0KB to 32KB with an interval of 4KB while the other two caches (i.e., in Cache-II) are fixed at 16KB. The curve corresponding to each cache shows the variation in the system performance.

From Figure 2.13 (a), we can see that the cache hit ratio increases as the cache size increases. However, the first-level cache has more impact on the cache hit ratio than the other two caches. For example, the hit ratio of Cache-I grows much faster than that of the other caches with the same interval of increase. This is because the first-level cache has fine granularity and each cache line stores one single mapping entry. In the second-level cache, each cache line stores one translation page, which includes multiple mapping entries. The first-level cache is much more flexible and effective at improving the cache hit ratio, especially for traces with more random-access patterns. The improved cache hit ratio leads to less translation page read and translation page write overhead. This can be verified by the results shown in Figure 2.13 (b) and (c). Moreover, Cache-II(1) shows a lower reduction in the translation page write count than the other two caches. This is due to the fact that only 53.50% of the requests in Trace 2 are sequential. Cache-II(1) has captured all of the spatial locality with a very small configuration size, and no further benefit can be achieved when the size continues to increase. Therefore, the curve of the translation page write count for Cache-II(1) is smoother that that of other caches.

Figure 2.13 (d)-(f) show the results of Trace 6. Trace 6 is a sequential-dominant trace with 97.91% of requests accessing the disk sequentially. Figure 2.13 (d) presents the cache hit ratio of Trace 6. From the figure, we can observe that the cache hit ratio shows no big change when the cache size is increased, which is the same as the result shown in the above section. However, the constant cache hit ratio does not mean that the other two caches have no functions. As shown in Figure 2.13 (e) and (f), the translation page read count and the translation page write count are significantly reduced. This is because that we designed the second-level cache to serve as the kick-out buffer of the first-level cache. With the increase in the size of the second-level cache, more dirty pages are served in the second-level cache, and less flush-back overhead is needed. From the figure, we can also see that there is more

improvement in the kick-out overhead in Cache-II(1) than in Cache-II(2) and Cache-I. This is because Trace 6 is a sequential-dominant trace, while Cache-II is specially designed for this kind of workloads. The results shown in Figure 2.13 serve as the guidelines for allocating cache sizes for different workloads. The size of the first-level cache should be larger than that of other two caches if the application is random-dominant. Otherwise, the size of the second-level cache, especially the size of Cache-II(1), should be larger than that of the other two caches.

2.6 Summary

In this chapter, we proposed a demand-based block-level address mapping scheme with two-level caches (DAC) in large-scale NAND flash storage systems to reduce the RAM footprint without excessively compromising system performance. In DAC, block-level NFTL was adopted as the baseline scheme. Instead of the traditional approach of storing the address mapping table in the RAM, we stored the block-level address mapping table in the flash memory, and only cached the demanded mapping entries into the RAM. A two-level caching mechanism was designed to improve the cache hit ratio by together exploring the temporal locality, spatial locality and access frequency in workloads. The experimental results showed that our scheme can considerably reduce the RAM cost while keeping the average system response time well under control. In particular, on average our technique achieves a 91.68% reduction in RAM cost with only a 2.02% penalty to the average system response time compared to previous work. Moreover, our scheme showed a better cache hit ratio and faster average system response time compared with DFTL when the cache size is limited in resource-constrained embedded systems.

CHAPTER 3

MNFTL: AN MLC NAND FLASH TRANSLATION LAYER WITH POSTPONED GARBAGE COLLECTION

3.1 Overview

NAND flash memory has been widely used in various storage systems due to its unique characteristics, such as non-volatility, low power-consumption, and fast access time. In recent years, multi-level cell (MLC) NAND flash memory has become the mainstream in the market for large-scale storage systems. A new NAND flash technology, MLC technology further increases the capacity of NAND flash memory chips by storing more than one bit of data per cell instead of the traditional one bit of data per cell used in single-level cell (SLC) technology. However, this new technology also introduces two write constraints. First, the pages within a block must be programmed (written) consecutively from the least significant bit (LSB) pages to the most significant bit (MSB) pages [11]; second, partial-programming is allowed for only once [6] in one page. These two constraints pose new challenges for existing flash translation layer (FTL) schemes that were originally designed for SLC NAND flash memory. This chapter proposes a novel flash translation layer to cope with the problems caused by these two constraints in MLC NAND flash storage systems.

In the past decade, three types of flash translation layer (FTL) schemes have been proposed: page-level mapping, block-level mapping, and hybrid-level mapping. Page-level FTL can allocate the pages within a block sequentially without recording the page status (valid or invalid) in the spare area. Therefore, page-level FTL is still usable to MLC flash. However, page-level FTL is unsuitable for a large-sized MLC flash due to the large address mapping table. How to reduce the size of the address mapping table is a crucial issue. Based

on page-level FTL, the DFTL scheme [33] stores the address mapping table in flash memory and only caches a small number of active mappings in RAM. It effectively reduces the RAM cost; however, it incurs extra valid page copies when maintaining the address mapping table in the flash memory. Block-level FTL schemes [14, 83, 95] use the block offset to locate the pages within a block, and the pages may be programmed randomly within a block. Therefore, block-level FTLs may not be applicable to MLC flash.

In hybrid-level FTL schemes, physical blocks are logically partitioned into data blocks (primary blocks) and log blocks (replacement blocks) [24, 26, 28, 48, 68, 77, 99]. A data block is used to store the first written data, while the updated data are stored in log blocks. In data blocks, most of these schemes adopt the block-level mapping approach and use the block offset to locate the pages. In the GFTL scheme [28], the pages can be written sequentially within a block; however, the average system response time is slower due to the earlier-triggered garbage collection. In the superblock based FTL scheme (SFTL) [48], the garbage collection may be triggered earlier by log blocks, and extra valid page copies may be needed. We have observed that valid page copies will directly incur the garbage collection overhead. Therefore, it is necessary to design a flash translation layer that will not only be applicable to MLC flash but also reduce the garbage collection overhead.

In this chapter, we propose a novel flash translation layer (FTL) called MNFTL for MLC NAND flash memory storage systems. We analyze several fundamental problems in the design of the MLC flash translation layer, and observe that unnecessary valid page copies cause the garbage collection overhead. In order to reduce the number of valid page copies, we propose two approaches to design the flash translation layer: *concentrated mapping* and *postponed reclamation*. Since the number of valid pages within one fully occupied block depends on the address mapping approach, concentrated mapping is utilized to store the written data and its updated data in the same physical block so that the invalid pages can be concentrated closer to each other. Moreover, a valid page may become invalid if the block to which it is allocated is later to be reclaimed. Thus, postponed reclamation is adopted to postpone the time at which the garbage collection is triggered, so that the number of invalid pages within one block can be increased. Both of the two approaches reduce the number of valid

pages within the victim block that will be selected for garbage collection. In our approach, concentrated mapping uses the page-level mapping approach, so that the write constraints of MLC NAND flash can be satisfied. The corresponding page-level mapping table is stored in the spare area of the newly allocated pages, while the page mapping table indices are recorded in the RAM. Therefore, limited RAM space is used. We conduct experiments on a set of benchmarks. The experimental results show that our scheme presents a reduction of 30.09% on the average system response time compared with previous work.

This chapter makes the following contributions:

- We present for the first time a flash translation layer to hide the new write constraints of MLC NAND flash memory.
- Our scheme is the first work that reduces the garbage collection overhead by reducing the number of valid page copies in the design of the MLC flash translation layer.
- We demonstrate the effectiveness of our techniques by comparing them with some representative FTLs using a set of realistic I/O workloads.

The rest of this chapter is organized as follows. Section 3.2 shows the background and the problem analysis in the FTL design. Section 3.3 presents our proposed MLC NAND flash translation layer scheme in detail. In Section 3.4, we present the performance evaluation of our scheme. Finally, we present our conclusions in Section 3.5.

3.2 Background and Problem Analysis

In this section, we first introduce the MLC NAND flash memory that is the focus of this chapter. Then, we analyze the problems posed by MLC flash in designing the flash translation layer. Finally, we give the motivation of this chapter.

3.2.1 MLC NAND Flash Memory

For today's media-rich mobile consumer electronics, NAND flash is widely adopted as the non-volatile memory-of-choice for multimedia and Internet capability. There are two types of NAND flash memory architecture: Single-Level Cell (SLC) and Multi-Level Cell (MLC). SLC NAND flash ICs have one bit of data stored per memory cell, and two states: erased (1) or programmed (0). MLC NAND flash ICs have two bits of data stored per memory cell, and four states: erased (11), two thirds (10), one third (01), or programmed (00). Figure 3.1 (a) and (b) show the voltage references for SLC flash and MLC flash, respectively. The complex architecture of MLC NAND flash increases the capacity of the NAND flash memory chip; however, it also results in a performance disadvantage when compared to SLC NAND flash. Since MLC NAND flash has four states, it must expend more energy in managing the electrical charge during operations. Therefore, energy consumption is greater with MLC than with SLC. The program and erase operations of MLC NAND flash last 10,000 cycles, while those of SLC NAND flash last 100, 000 cycles. Moreover, the complex architecture of MLC NAND flash introduces two constraints in programming data. Random page programming within one block and multiple partial page programming within one page are no longer allowed. The two write constraints pose new challenges for its management, in particular, with regard to design of the flash translation layer.

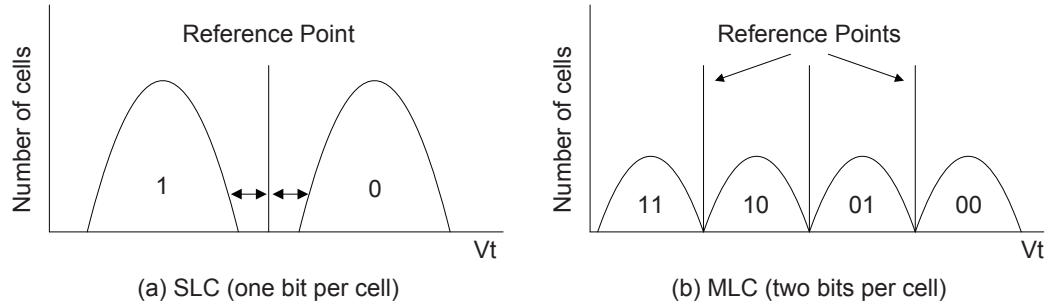


Figure 3.1: Voltage references for SLC and MLC flash cell.

3.2.2 Problem Analysis

In this section, we analyze some fundamental problems in the design of the MLC flash translation layer, taking into consideration the new write constraints of MLC flash.

Our first objective is to answer the following question: *is page-level mapping a must in the design of the MLC NAND flash translation layer?* In a block-level FTL [14], a logical page number (LPN) is divided into a logical block number (LBN) and a block offset (BO), and the logical block number is translated to a physical block number (PBN). The block offset helps to find the target page within the physical block. Given the logical page number, divided by the number of pages in a physical block, the quotient is the logical block number and the remainder is the block offset. When the block offset is used to locate the physical page, a set of consecutive pages in the logical block is usually stored in the same physical block. But the physical pages might be written randomly for the random pages in the logical block. This situation also exists in hybrid-level FTL schemes [26,68,77,99], which adopt the block offset to locate pages in their block-level mapping schemes. In a page-level FTL [13], an LPN is translated to a physical block number (PBN) and a physical page number (PPN). Since a logical page can be mapped with a physical page in any location in flash memory, sequential allocation of the pages within a block is allowed. In addition, the pages maintained in the mapping table are valid, so the page status (valid or invalid) does not need to be stored in the spare area. Therefore, the page-level mapping approach is potentially beneficial in overcoming the write constraints in MLC flash. Our observation is that the *page-level mapping approach is necessary in designing the MLC flash translation layer*.

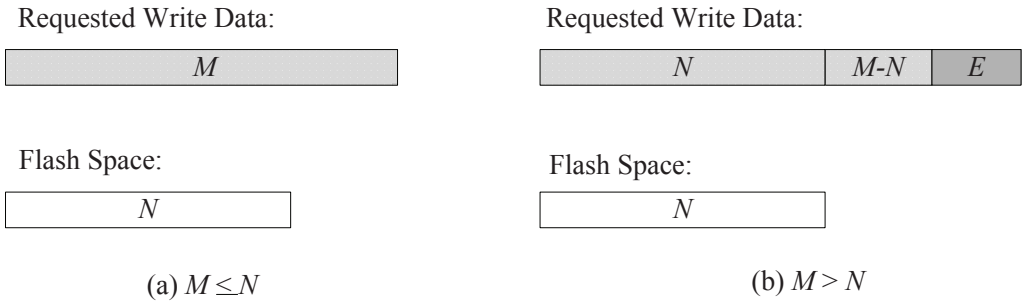


Figure 3.2: Extra overhead in garbage collection.

Another challenge we face in designing the FTL is the garbage collection overhead. Therefore, the next question we investigate is: *what is the fundamental overhead for garbage collection in NAND flash memory?* Given a set of write requests, we assume that the total amount of space required to store the requested data is *M*, and that the total amount of space that the flash memory chip can provide is *N*. If $M \leq N$, as shown in Figure 3.2(a), the flash memory chip can provide enough space to store the requested data, and no garbage collection is needed. For this case, smart FTL schemes should not incur any garbage collection. If $M > N$, as shown in Figure 3.2(b), the flash memory chip will not have enough space to service all of the requests. In order to store the *M-N* data into the flash chip, garbage collection must be performed to reclaim some obsolete space scattered over the flash chip. During the garbage collection, valid pages in the victim block need to be copied into blocks that contain free pages, which require extra space to store these valid pages. We assume that this extra space is *E*, where *E* indicates the garbage collection overhead. For this case, FTL schemes should try to minimize this garbage collection overhead. Based on this analysis, the first observation we make is that *the valid page copies cause the essential garbage collection overhead in NAND flash memory*.

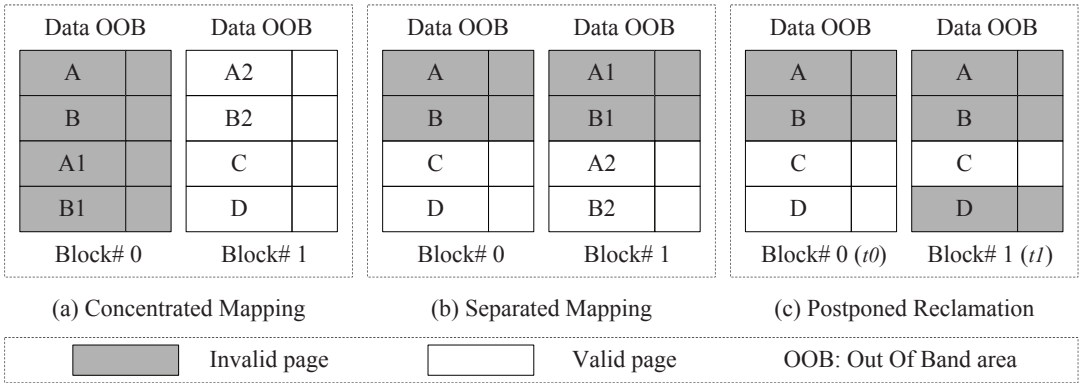


Figure 3.3: Two mapping approaches and postponed reclamation.

Since reducing valid page copies can cut down the garbage collection overhead, our next step is to explore in detail the method involved in effectively reducing the number of valid page copies in garbage collection. Two factors determine the number of valid pages in a physical block that is selected as a victim block, the distribution of write (update) operations

mapped to this block, and the time required to trigger the garbage collection to turn this block into a victim block. The first factor is based on the dedicated FTL scheme. If a write request is mapped to a physical block that contains the old version of data, the number of valid page copies may be reduced. Figure 3.3 shows an example. For the purpose of demonstration, we assume that each physical block has four pages. Given a set of write requests (*A, B, A1, B1, A2, B2, C, D*), *A1*, *A2* are updated versions of *A*, and *B1*, *B2* are updated versions of *B*. In Figure 3.3(a), *A* and *B* together with updated version *A1* and *B1* are mapped to block *0*, while *A2*, *B2*, *C*, and *D* are mapped to block *1*. All four pages in block *0* are invalid, and no valid page copy is needed when reclaiming block *0*. This mapping is called *concentrated mapping*. In the separated mapping approach shown in Figure 3.3(b), block *0* is designed to store the first version of data. When block *0* is selected as a victim block to perform garbage collection, two valid page copies (for A and B) are needed. This example shows that concentrated mapping outperforms separated mapping in reducing the number of valid page copies.

The time at which to trigger the garbage collection also affects the number of valid pages in a victim block. An example is shown in Figure 3.3(c). At time *t0*, when block *0* is selected as a victim block, two valid page copies (C and D) are needed. If the *postponed reclamation* approach is applied to postpone the time for garbage collection, the number of valid page copies may be reduced as well. At time *t1*, *t1*$>$*t0*, D is updated by the new version of the data, and only one valid page copy (C) is needed when performing garbage collection. Therefore, the second observation that we make is that *concentrated mapping and postponed reclamation are effective at reducing the number of valid page copies.*

3.2.3 Motivation

Duo to the write constraints in MLC flash, most existing FTL schemes have a limited ability to manage the MLC flash memory storage systems. Page-level FTLs [13] can be used for MLC without modification; however, the big RAM footprint is an issue for large-capacity based MLC NAND flash memory. GFTL [28], DFTL [33], and SFTL [48] can be used in

MLC flash; however, they suffer from a slow average system response time due to the earlier-triggered garbage collection. How to design an efficient FTL scheme for MLC becomes an important issue. Through an analysis of the problem in above section, we make three observations. First, the page-level mapping approach is necessary for the MLC FTL design if the two write constraints are to be overcome. Second, valid page copies are the essential garbage collection overhead. Third, concentrated address mapping and postponed reclamation can effectively reduce the garbage collection overhead. These observations provide us with insights on how to design an efficient flash translation layer for MLC flash.

3.3 MNFTL: MLC NAND Flash Translation Layer

In this section, an efficient hybrid-level MLC NAND flash translation layer, called MNFTL, is proposed. In our scheme, the page-level mapping approach is applied to each logical block in which concentrated mapping is deployed and limited RAM space is taken. In Section 3.3.1, an adaptive block-level mapping scheme is also proposed in which the postponed reclamation mechanism is implemented. In Section 3.3.2, detailed write and read operations in MNFTL are presented based on the hybrid-level address mapping scheme. In Section 3.3.3, a novel garbage collection policy is introduced to reduce the number of valid page copies and block erase counts.

3.3.1 MNFTL with Concentrated Address Mapping

In MNFTL, one logical page number is translated to one logical block number (LBN) and one block offset (BO) as shown in Figure 3.4. One logical block is mapped with M physical blocks. M is varied in an on-demand fashion. If more write (update) requests are issued to one logical block, more physical blocks will be needed, and M will be increased correspondingly. Otherwise, M will be decreased when these physical blocks are reclaimed. The block mapping table (BMT) for each logical block is represented by a linked-list. The head of a list is the logical block number (LBN) and each node in the list is one physical block

number (PBN) that is mapped to this LBN. The pages in one logical block are managed with the page-level mapping approach. Each page in one logical block can be mapped with any physical pages in its corresponding *M* physical blocks. Pages are mapped and programmed sequentially in each physical block. The page mapping table (PMT) for each logical block is divided into *N* sub-tables, and each sub-table is stored in the spare area (OOB) of the newly mapped physical page. *N* pointers are recorded in RAM as the indices of the page mapping table. The value of *N* depends on the size of the page mapping table of one logical block and the size of the spare area (OOB) of one physical page.

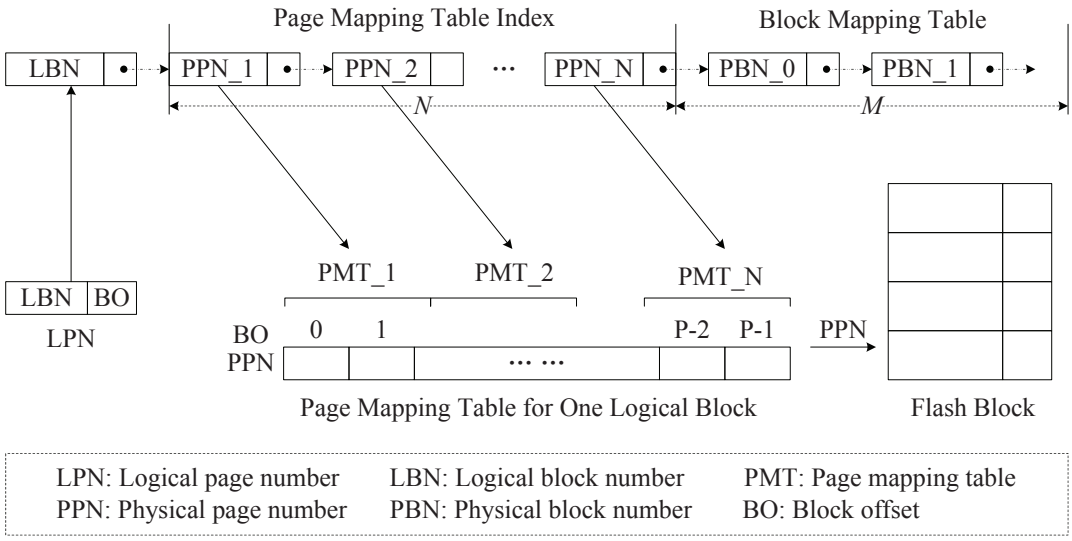


Figure 3.4: Address translation in MNFTL.

Figure 3.4 shows the block mapping table (BMT) for one LBN in RAM and the page mapping table (PMT) for one logical block. In block-level mapping, one logical block can be mapped to any physical block in the whole flash memory. The blocks mapped to one LBN form a linked-list, and the linked-list of all LBNs form a linked-list array. In page-level mapping for each logical block, one logical page can be mapped with any physical page in its corresponding physical blocks. Suppose both one logical block and one physical block include *P* pages, so that the entire page-mapping table for one logical block has *P* rows. Assume that the spare area of one physical page can store *Q* ($P \geq Q > 0$) rows of mapping slots. The whole page mapping table is then divided into *N* sub-tables according to the logical page number, where $N = \lfloor P/Q \rfloor$. One sub-table together with the requested data is written into the

spare area and the data area of the mapped physical page separately. This programming operation can be implemented in one write cycle, so it obeys the new constraints for MLC NAND flash memory [6]. Besides that, in the block mapping table (BMT), *N* pointers point to the physical pages, which store the newest version of the page mapping table. In this way, the page mapping table (PMT) can be obtained directly by reading the spare area of the physical pages while limited RAM space is taken when doing address translation.

Managed by the block-level mapping approach in MNFTL, all of the data accessing the same logical block are concentrated in one or more physical blocks. The first written data and the re-written data are consequently distributed closer to each other, which increases the possibility that an invalid page can be allocated within one physical block. The number of invalid pages within one block can be increased and the number of valid page copy operations reduced when the block is selected as the victim by the garbage collection process. Therefore, the concentrated mapping approach can reduce the garbage collection overhead.

3.3.2 MNFTL Reads and Writes

A write request issued from the file system is represented by a piece of data and a logical page number (LPN), e.g., *write(A,35)*. Given the LPN, divided by the page numbers in one logical block, the quotient is the logical block number (LBN), and the remainder is the block offset (BO). After the translation from logical page number to logical block number, the first write to a given logical page is to the first free page in a free physical block that is mapped to the logical block. Once a physical block is mapped, pages are allocated sequentially, regardless of whether the operation is a write or an update operation. After *P* writes, the physical block becomes full, and a new free physical block will be allocated to the logical block if necessary. When a new page is mapped, the newest version of the page mapping sub-table (which includes the requested block offset) will be read out from the spare area of the page pointed to by pointers in the block mapping table. The corresponding mapping slot will be updated and then written to the spare area of the new page, together with the requested data written to the data area. The pointer in the block mapping table will also

point to the new physical page. The time taken for a write request is one OOB read and one page write ($T\_rdoob + T\_wrpg$) if a free block and a free page are available.

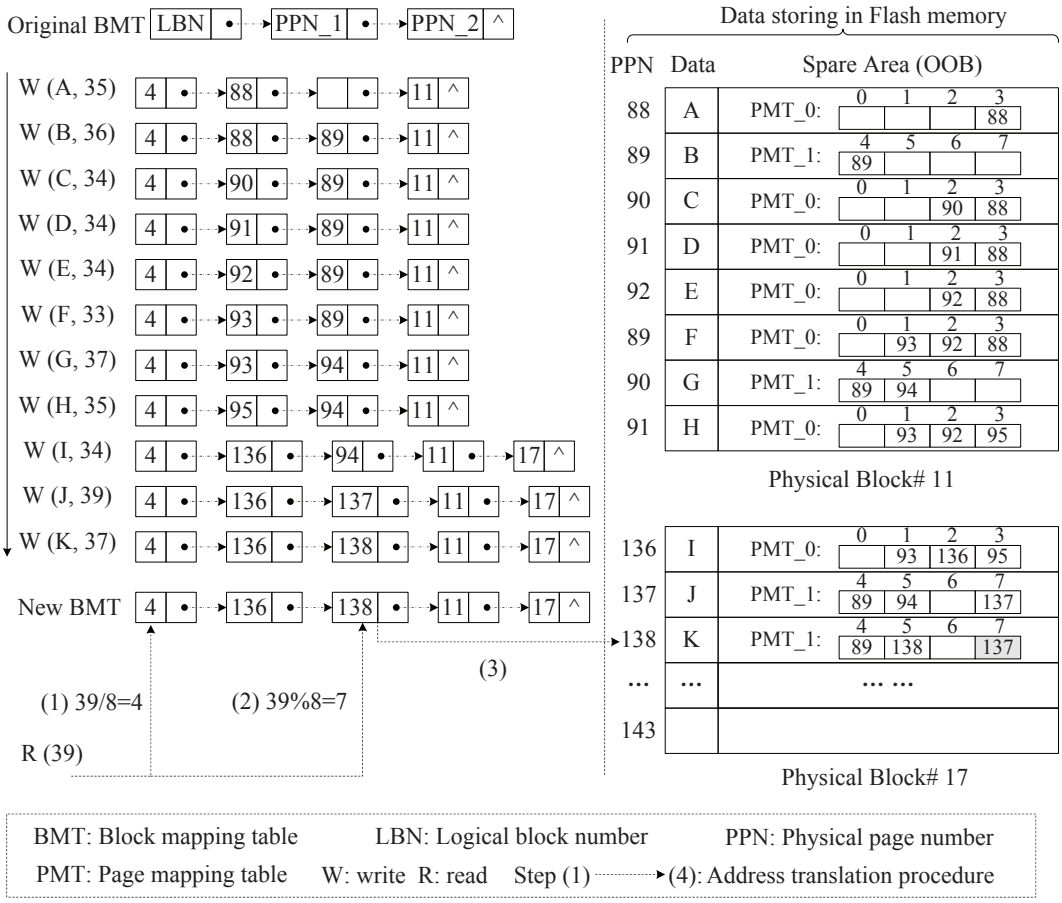


Figure 3.5: Illustration of address translation in MNFTL.

An example of a write operation in MNFTL is given in Figure 3.5. Assume that each block has eight pages, and that the page mapping table for one logical block is divided into two parts: *PMT_0 (BO:0-3)* and *PMT_1 (BO:4-7)*. The original block mapping table is free. For the first write request *write(A,35)*, the corresponding *LBN* and *BO* are *4* and *3*, respectively. A new free block *PBN=11* is allocated, and the data *A* is written into the data area of the first free page *PPN=88*. The updated mapping sub-table *PMT_0* is stored in the spare area of page *PPN=88*. The corresponding pointer *PPN_1* in the block mapping table simultaneously points to *PPN=88*. After eight writes, the physical block *PBN=11* becomes full, a new free block *PBN=17* is allocated, and the data are written sequentially into the pages. After 11 writes from the file system, the new block mapping table is given. For the

logical block *LBN=4*, two physical blocks are consumed. *PPN_0=136* and *PPN_1=138* point to the new version of the page mapping table for this logical block.

A read request issued from the file system is represented by a logical page number (LPN), e.g., *read (39)*. The corresponding LBN will first be searched in the block mapping table. Then, the page mapping sub-table for the requested BO can be obtained using the pointer in the block mapping table. From the sub-table, we can get the physical page, which stores the requested data. The time overhead for one read request is one OOB read and one page read: $T\_rdoob + T\_rdpg$. In Figure 3.5, an example is given for read request *read(39)*. In step(1)-(2), using the *LBN=4* and *BO=7*, we obtain the *PPN_1=138*, which stores the requested page mapping table. In step(3), by reading the spare area, we get the *PMT_1 (BO:4-7)* and the target page *PPN=137*. By reading the data area of target page *PPN=137*, we obtain the valid target data *J*.

3.3.3 MNFTL with Postponed Garbage Collection

The garbage collection mechanism in MNFTL aims to reduce the number of valid page copies and block erase counts. It is invoked once there are no free blocks to allocate. One fully occupied physical block with the fewest valid pages in the whole flash memory will be selected as the victim block. The valid pages in the victim block are copied to another physical block, which is mapped to the same logical block along with the victim block. Since the concentrated mapping approach is adopted in MNFTL, the number of valid page copies can be reduced. Moreover, the physical blocks are allocated in an on-demand fashion, and the garbage collection is triggered until all of the blocks are used. This is different from the address mapping approach adopted by existing FTL schemes, in which one or more physical blocks can only be mapped to specific logical block(s). The fixed mapping management triggers the garbage collection earlier, before all of the blocks are used. Therefore, the garbage collection in MNFTL actually delays the time at which to reclaim the invalid space. The delayed reclamation may enable a valid page to become invalid so that the number of invalid pages can be increased and the number of valid page copies can be reduced. In MNFTL, the

garbage collection of a victim block amounts to the following steps:

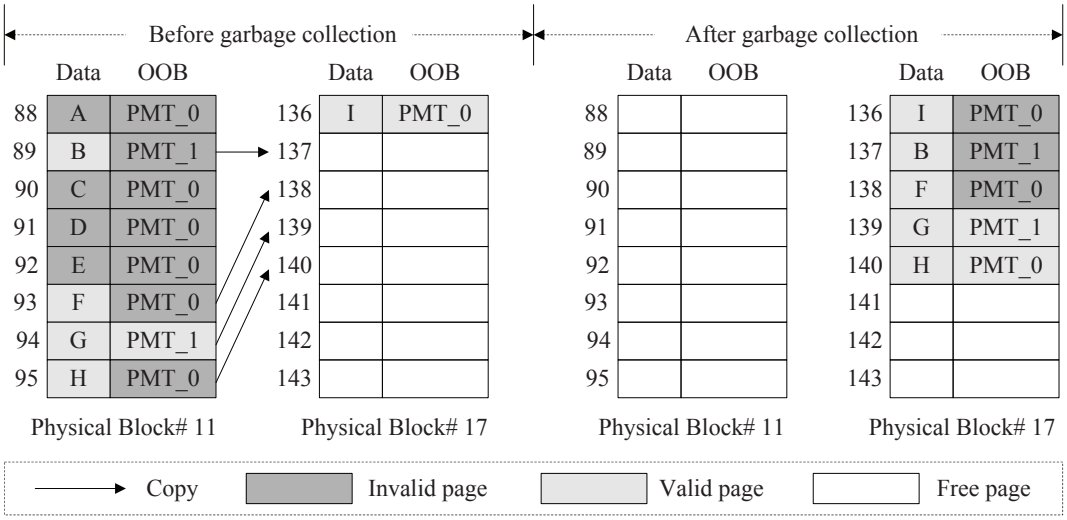


Figure 3.6: Garbage collection in MNFTL.

1. *Select the victim block*: In this step, the block with the fewest valid pages is selected as the victim block. If the pages in this block are not referenced in the page mapping table, then they are invalid, otherwise, they are valid. The time cost to identify the valid pages in the block is $N \times T\_rdoob$, where *N* is the number of sub page mapping tables for one logical block. In Figure 4.5, suppose the physical block *PBN 11* is selected as the victim block after *page 136* in *PBN 17* is written, and *PBN 17* is the new block mapped to the same logical block. In that case, victim block *PBN 11* has four valid pages which will be copied to the free pages in physical block *PBN 17*.

2. *Copy the valid pages*: The pages in the victim block can be classified into three types according to the difference in state between the data area and the spare area. (a) Full valid page: both the data area and the spare area are valid (e.g., *page 94* in Figure 4.5). (b) Full invalid page: both the data area and space area are invalid (e.g., *page 88*). (c) Partial valid page: the data area is valid and the spare area is invalid (e.g., *page 89*). When copying one valid page (regardless of whether it is a full valid page or a partial valid page) to a new block, we need to read out its mapping sub-table, and write the updated mapping sub-table as well as the data into a new free page. Assume that there are *S* valid pages in the victim block, the time overhead to copy these valid pages is $S \times (T\_rdpg + T\_wrpg)$.

3. *Erase the victim block*: The victim block (e.g., *block 11*) is erased with time overhead T_er. Figure 4.5 shows an example of the garbage collection procedure in MNFTL. The total time cost of this process is $N \times T\_rdoob + S \times (T\_rdpg + T\_wrpg) + T\_er$. Instead of fully searching all physical blocks, we first select the victim block from the logical block, which has been mapped with the maximum number of data blocks when garbage collection is triggered. If more physical blocks are mapped to one logical block, then more update operations are performed in the mapped physical blocks so that fewer valid pages can be obtained from the victim block. Let us suppose one block has *P* pages. If *P* physical blocks are mapped to the same logical block, then each physical block has at most one valid page left; if *P+1* physical blocks are mapped to the same logical block, at least one of the physical blocks will not have valid pages, which is the ideal scenario to reduce the garbage collection overhead. Moreover, there are obvious working and idle time periods in a working cycle for most real applications. In fact, we can perform reclaim operations on the logical blocks mapped with many physical blocks when the system is idle. In this way, by utilizing the idle period, more free blocks can be generated. Moreover, the wear-leveling of flash memory in MNFTL is managed automatically by locating the blocks in a round-robin approach.

3.4 Evaluation

In this section, we present the experimental setup and the experimental results with an analysis. We compare and evaluate our proposed MNFTL scheme over four representative FTL schemes: PFTL (Page-level FTL) [13], GFTL [28], DFTL [33], and SFTL [48], in terms of three performance metrics: the main-memory requirements, the average system response time, and the garbage collection overhead.

3.4.1 Experimental Setup

We developed a trace-driven MLC NAND flash simulator under Linux 2.6.17 and implemented five schemes: PFTL (Page-level FTL) [13], GFTL [28], DFTL [33], SFTL [48], and

MNFTL. To conduct a fair comparison with different FTL schemes, we consider only a portion of flash as the active region that stores our workloads. The remaining flash is assumed to contain cold data or free blocks, which are not under consideration. The framework of our simulation platform, as shown in Figure 3.7, consists of two modules: a NAND flash simulator module providing basic read, write, and erase capabilities; and a desired MLC NAND flash translation layer management scheme that can be executed on top of the NAND flash simulator. The traces along with various flash parameters, such as block size and page size, page read time and page write time, and so on, are fed into our simulation framework.

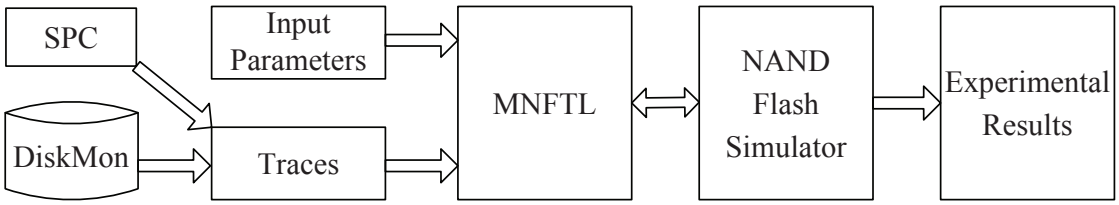


Figure 3.7: The framework of the simulation platform.

In the experiment, a 8GB MLC NAND flash memory is configured. The page size and the block size are set as 2KB and 256KB, respectively. The time cost for one OOB read, one page read, one page write, and one block erase are set as 20μs, 60μs, 800μs, and 1500μs, respectively. One access to the address mapping table in RAM is set as 5μs. In the simulation, we assume only a portion of flash as the active region that stores our test workloads. For the SFTL scheme, we set one superblock size as 6 (4 data blocks and 2 log blocks), and the total log block number is set as 256. The cache size in the DFTL scheme is set as 64KB, which is about 4% of the whole page mapping table stored in the flash memory. We use a set of benchmarks from both the real-world and synthetic traces to study the system performance for different FTL schemes. The traces used in this simulation are summarized in Table 3.1. *Financial1* and *Financial2* are I/O traces from an OLTP application running at a financial institution [5] obtained from the Storage Performance Council (SPC). *Websearch* is a read-dominant trace also made available by SPC. *Systemdisk1*, *Systemdisk2*, and *Systemdisk3* are traces that we collected from the desktop running Diskmon with Windows XP on an NTFS file system.

Table 3.1: Traces used for simulation.

Traces	Number of Requests	% of Write Requests	Average Request Size (KB)
Financial1	1,333,747	78.56	3.17
Financial2	3,699,194	17.65	2.26
Websearch	4,261,709	0.02	15.05
Systemdisk1	1,040,692	74.04	42.65
Systemdisk2	2,636,016	61.96	44.10
Systemdisk3	1,312,945	58.10	36.72

The main-memory requirement for a flash translation layer depends mainly on the size of the address mapping table. For the simulated 8GB MLC NAND flash memory chip, one physical page (block) number takes about 3 bytes (2 bytes) of RAM space, while one pointer in the linked-list requires 4 bytes of RAM space. For GFTL, DFTL, and SFTL, the address mapping table are $446KB$, $176KB$, and $62KB$, respectively. For the page-level FTL, the address mapping table takes up $12MB$ of RAM space. Our scheme applies the page-level mapping scheme in each logical block, and stores the page mapping table indices in RAM. The RAM space in our scheme is about $1.06MB$ ($32 \times 1024 \times 34B$). Our scheme results in a big reduction in RAM cost compared with the page-level FTL.

3.4.2 Results and Discussion

In this section, we show the experimental results in terms of the average system response time and the garbage collection overhead for different FTL schemes. Analysis is given to demonstrate how our MNFTL scheme outperforms other FTL schemes. Figure 3.8 shows the average system response time for different FTL schemes under the same experimental environment over six traces. In Figure 3.8, the X-axis represents the six traces and the Y-axis shows the average system response time. From the results, we can see that, our

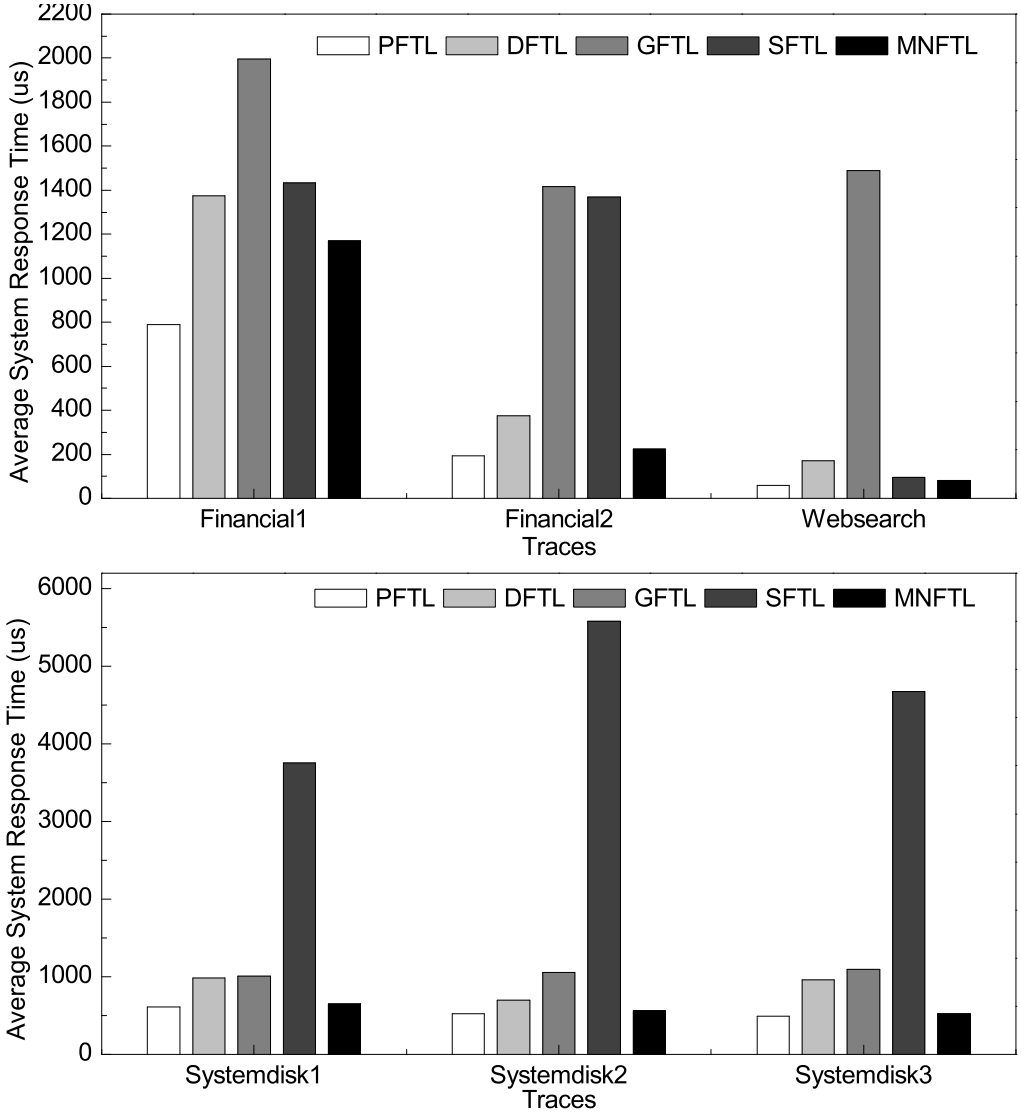


Figure 3.8: The average system response time of different FTLs over six traces.

proposed MNFTL can achieve an average reduction of 30.92% in average response time among the six traces compared with the DFTL scheme, and more improvements can be obtained compared to the GFTL scheme and the SFTL scheme. In particular, for the read-dominant trace *Websearch*, our scheme is 15μs faster than SFTL and 20μs slower than the page-level FTL scheme. This is because the page-level FTL scheme can find the requested address mapping in RAM directly, while MNFTL needs to read one OOB (T_rdoob) to get the target page mapping table. However, the SFTL scheme needs to read two OOBs ($2 \times T\_rdoob$) to obtain the requested mapping table. Therefore, if no garbage collection is

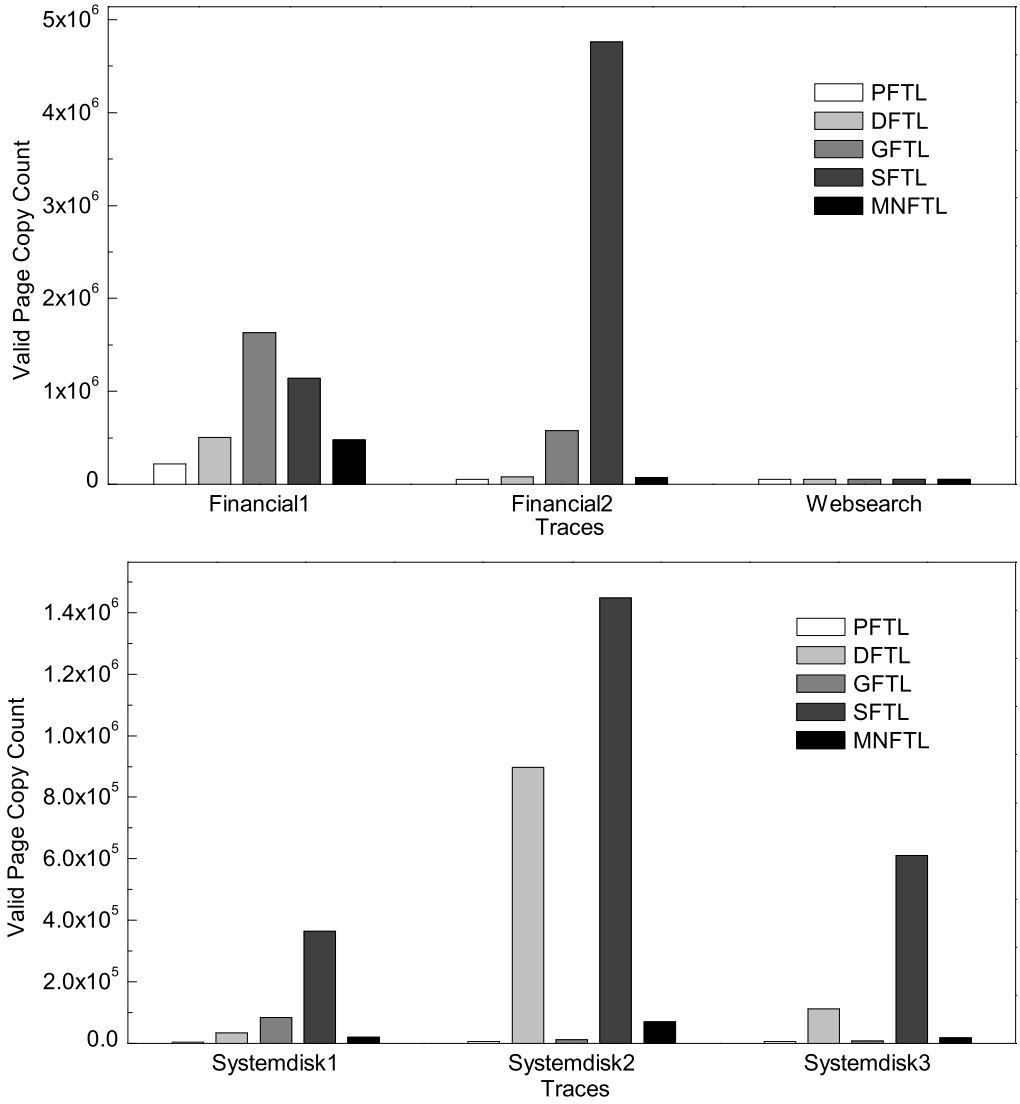


Figure 3.9: The number of valid page copy for different FTLs over six traces.

invoked, the average system response time for read requests in these schemes is a difference of about one OOB read (T_rdoob).

For write-dominant traces, we observe that MNFTL shows a much faster average response time than DFTL, GFTL, and SFTL and a slightly slower average response time than page-level FTL. This is based on the fact that, the DFTL scheme introduces translation blocks to save the address mapping table, and the GFTL scheme uses some extra blocks (about 16% of all data blocks) as the write buffer in order to guarantee the real-time performance, while the SFTL scheme introduces a small number of log blocks to store the updated data. These

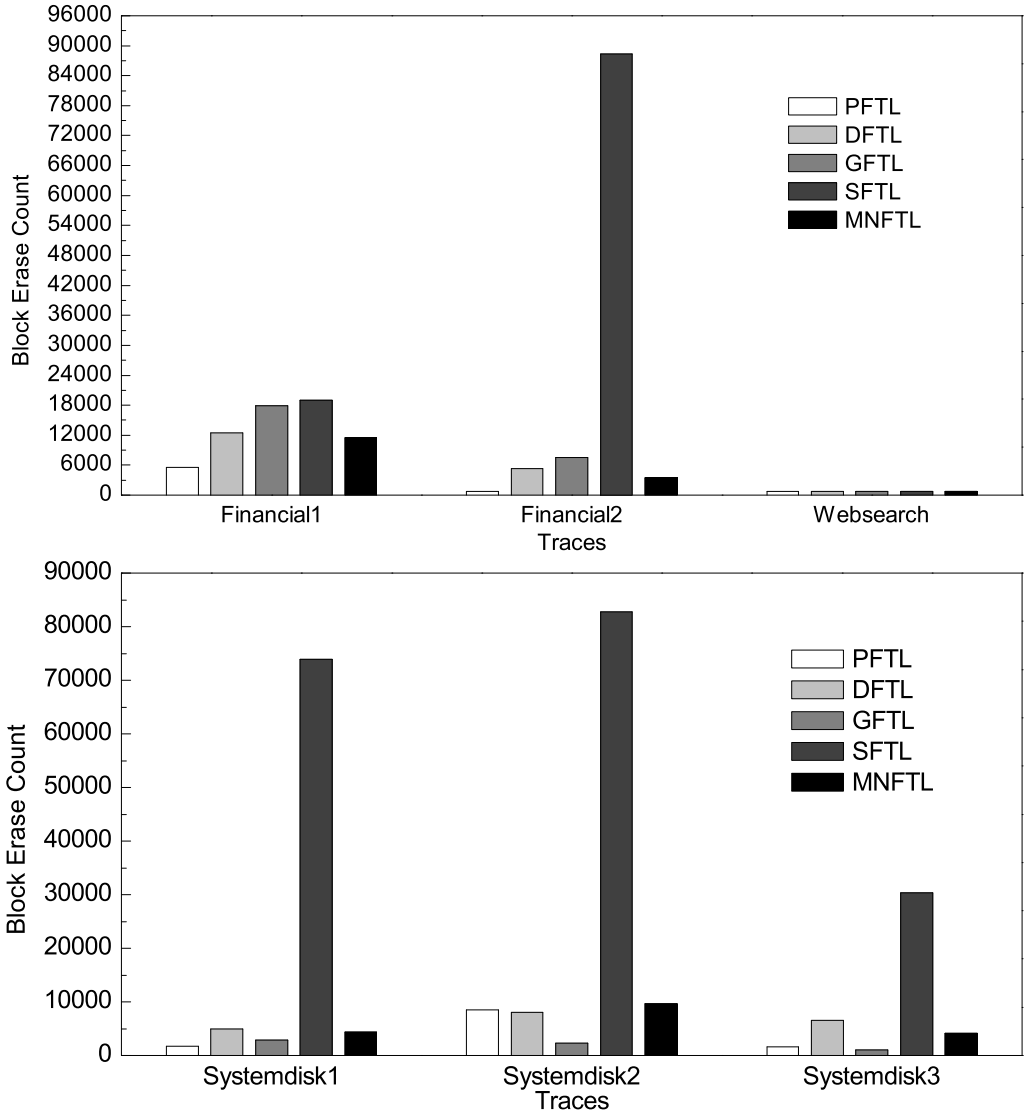


Figure 3.10: The number of block erase for different FTLs over six traces.

extra blocks led to the earlier triggered garbage collection, which resulted in more valid page copies and block erase counts. This observation is also proven by the experimental results for the valid page copy count and block erase count, which are shown in Figure 3.9 and Figure 3.10, respectively. From the results, we observe that our MNFTL scheme can achieve an average reduction of 69.78% in the number of valid page copies, and a 33.35% average reduction in the number of block erase counts compared with the DFTL scheme. For the GFTL scheme and the SFTL scheme, we find that a significant number of valid page copy and block erase operations are invoked. This is because, in the GFTL scheme, the garbage

collection is triggered once a physical block is full, and it is continually performed whenever one block exists in the garbage collection queue (GCQ). In the SFTL scheme, four data blocks in a superblock share the same two log blocks. Once the two log blocks are full or the four data blocks are full, the garbage collection will be triggered. In our scheme, no extra blocks are involved, so that block reclamation is invoked when nearly all of the data blocks are consumed. From the experimental results, we also observe that the number of valid page copies and the block erase counts for trace *Websearch* are 0. This is because 99.98% of the requests in *Websearch* are read requests, and the write requests are unable to trigger the garbage collection.

3.5 Summary

In this chapter, we studied the problem of reducing the garbage collection overhead in designing the MLC flash translation layer while satisfying the write constraints of MLC flash memory. An efficient MLC NAND flash translation layer, called MNFTL, was proposed, in which a novel address mapping scheme was adopted to fundamentally reduce the garbage collection overhead with a limited amount of RAM usage. By applying the proposed concentrated mapping and postponed reclamation, MNFTL was able to effectively reduce the number of valid page copies and block erase counts. We conducted experiments on a set of benchmarks, and the experimental results showed that our scheme can significantly improve the average system response time compared with previous work.

CHAPTER 4

RFTL: A REAL-TIME FLASH TRANSLATION LAYER WITH DISTRIBUTED PARTIAL GARBAGE COLLECTION

4.1 Overview

No matter in mission-critical hard real-time systems such as aerospace [9] and the military or in soft real-time systems such as iPads and smart phones, NAND flash memory has become essential due to its unique characteristics, such as non-volatility, low power-consumption, and fast access time. However, in NAND flash, a page once written cannot be overwritten until it is erased (out-of-place update). The erase operation can only be performed in a unit of one block (bulk-erase). These properties have caused the response time to become unpredictable. Most existing FTL schemes focus on improving the average performance, but ignore the real-time storage performance. In this chapter, we propose a real-time FTL scheme that can provide an upper bound to the worst-case system response time for I/O requests in NAND flash storage systems.

A flash translation layer is a block-device-emulation software layer that simulates NAND flash as a hard disk by hiding "out-of-place update" and "bulk-erase" properties. One function of FTL is to do address mapping between a logical address in file systems to a physical address in flash media. Another important function is to reclaim the space by erasing obsolete blocks in flash, also known as *garbage collection*. Garbage collection will be invoked if there is not enough free space to serve the requests. Given a read/write request issued from the file system, the best-case response time is constant, since no garbage collection is invoked. However, in the worst case, a request will be blocked by the time-consuming garbage collection. The request consequently suffers a long latency, which might

be intolerable for mission-critical real-time applications. Therefore, how to design a service-guaranteed FTL scheme for real-time applications has become an important problem.

In previous work, several techniques have been proposed to solve this problem. Chang et al. [22] was the first to propose real-time garbage collection for flash memory storage systems, where predictable performance is guaranteed by ensuring that enough free space is always available for write requests. Although an upper bound to the response time can be obtained, their approach suffers from a slow worst-case response time and requires extra file system support. Choudhuri et al. [28] proposed a flash translation layer called GFTL to guarantee an upper bound to the response time. GFTL reduces the upper bound by adding extra blocks as the write buffer and using a partial block cleaning policy to hide the long garbage collection latency. In order to provide enough free space to serve write requests, the full blocks are centrally organized in a garbage collection queue, and the garbage collection operations are consecutively performed as long as the queue is not empty. GFTL guarantees a worst-case response time for write requests, however, it suffers from a slower worst-case response time for read requests. Moreover, it introduces a large amount of extra page copy operations, which significantly degrade the average system response time. Since garbage collection does not occur very often, a scheme should not sacrifice too much average response time when reducing the worst-case response time. We address this problem in this chapter.

In this chapter, we propose a real-time flash translation layer, called RFTL, which provides not only an ideal upper bound to the worst-case response time but also a faster average response time. A distributed partial garbage collection policy is applied in RFTL. Different from the centralized partial garbage collection policy [9], in which all full blocks are put into a queue and garbage collection is performed in a centralized manner, garbage collection in RFTL is distributed to each logical block and a full block is reclaimed according to the arrival sequence of write requests in a distributed manner. The condition to invoke one partial step in garbage collection is when a write request arrives and the corresponding requested data block is full. Since a write request is served immediately after one partial garbage collection step, the worst-case response time of a request is only the overhead to

perform one partial step in garbage collection. Moreover, in a logical block, the garbage collection of a full block is performed only when there is a write request to the logical block; therefore, many unnecessary valid page copies and block erase operations are avoided so as to significantly improve the average system response time. Compared with GFTL, our approach does need more flash memory space; however, it effectively reduces the more valuable RAM cost. To the best of our knowledge, this is the first work to reduce both the average response time and worst-case response time by applying a distributed partial garbage collection policy in NAND flash memory storage systems.

We evaluate our scheme with a set of benchmarks running on a NAND flash memory simulator that we developed under Linux kernel 2.6.17. The experimental results show that our scheme can achieve a 36.30% improvement in the worst-case response time compared with GFTL. Moreover, we make a trade-off between the flash space and the average system response time. By doubling the flash space of GFTL, our scheme leads to a 91.79% reduction in the more valuable RAM space and a 67.06% improvement in the average system response time compared with GFTL.

This chapter makes the following contributions:

- We present for the first time a real-time flash translation layer to improve the worst-case system response time of NAND flash memory storage systems.
- We present for the first time a distributed partial garbage collection policy to enable the system to simultaneously reclaim space and serve the write requests.
- We demonstrate the effectiveness of our technique by comparing it with representative FTL schemes using a set of realistic I/O workloads.

The rest of this chapter is organized as follows. Section 4.2 shows background and motivation. Section 4.3 presents our RFTL scheme and the WCET analysis. In Section 4.4, we present the performance evaluation of our scheme, and in Section 4.5 we give our conclusions.

4.2 Background and Motivation

In this section, we first introduce the performance specifications of the NAND flash memory chip. Then, we describe the garbage collection overhead in some representative FTL schemes. Finally, we present the motivation of our work.

Table 4.1: NAND flash specifications.

Characteristics	Samsung 16MB Small Block SLC	Samsung 128MB Large Block SLC
Block size	16KB	64KB
Page size	512B	2KB
OOB size	16B	64B
Read page	36μs	25μs
Read OOB	10μs	25μs
Write page	200μs	300μs
Erase	2000μs	2000μs

4.2.1 Characteristics of Flash Memory Operations

A typical flash memory chip supports three kinds of operations: page read, page write, and block erase. The performance of the three operations is quite different, as shown in Table 4.1. A block erase takes a much longer time than a page write, which is much longer than a page read. With the propagation of writes in a flash memory chip, free space shrinks and garbage collection is invoked to regenerate some new free space for reuse. The garbage collection process may include a number of page read, page write, and block erase operations. Since garbage collection is usually considered uninterruptable, a pending write request may be blocked and the response time will largely depend on the garbage collection latency.

4.2.2 Garbage Collection Overhead

In the past decade, three kinds of FTL schemes have been proposed and different garbage collection policies adopted. In page-level FTL [13], one logical page (sector) is mapped with one physical page. The garbage collection in a page-level FTL is invoked when the NAND flash runs out of space, and each time only one victim block will be reclaimed. In general, the block with the fewest valid pages is taken as the victim block. The victim block will be erased after the valid pages are copied into a new free block. Suppose that one block consists of π pages and that the victim block has M valid pages ($\pi \geq M \geq 0$). The time overhead to reclaim the victim block is $M*(T_{rdpg}+T_{wrpg})+T_{er}$, where T_{rdpg} is the time required to read a page, T_{wrpg} is the time needed to write a page, and T_{er} is the time that it takes to erase a block.

In block-level FTL schemes [14], a logical page number (LPN) is made up of a logical block number (LBN) and a block offset (BO). One logical block is mapped with a physical block (called the *primary block*). In the case of a rewrite operation (or if the primary block is full), a new physical block (called the *replacement block*) is chosen to serve the write requests. The garbage collection in a block-level FTL is invoked once both the primary and the replacement blocks are full. Both of these blocks will be erased after being merged into a new free block. Since two blocks are involved in this process, the garbage collection latency is much longer in the worst case compared with the one in page-level FTL.

In hybrid-level FTL schemes [26, 85, 99], physical blocks are logically partitioned into data blocks (primary blocks) and log blocks (replacement blocks). A data block is used to store the first written data, while the updated data is stored in log blocks. Since one log block might be shared by more than one data block, the garbage collection needs to reclaim the data block and all associated log blocks at the same time. Thus, for a merge operation in hybrid-level FTL schemes, valid pages scattered in a data block and its corresponding log blocks are copied into more than one free block. The garbage collection latency of hybrid-level FTL tends to be much longer than that of page-level FTLs and block-level FTLs.

4.2.3 Motivation

The non-deterministic response time of requests in NAND flash memory is caused by the variable garbage collection latency. Figure 4.1 shows an illustrative example of the garbage collection (GC) process in page-level FTL schemes. For the sake of illustration, we assume that each block consists of eight physical pages. In Figure 4.1, the victim block consists of five valid pages. These valid pages are copied to a new free block. After that, all of pages in the victim block become invalid and the victim block is then erased for reuse. Based on the specifications of a small block NAND flash shown in Table 4.1, the time overhead to reclaim this block is $5*(36+200)+2000$=$3180\mu s$. Given a write request, the response time is $200\mu s$ if no garbage collection is triggered. Otherwise, the response time becomes $3380\mu s$ when the request is blocked by the garbage collection with five valid-page copy operations. Such long time latency limits the usage of NAND flash in real-time applications. Moreover, since the number of valid pages in different victim blocks is different, the time overhead to reclaim these blocks varies, which makes the response time of the requests non-deterministic. These observations motivated us to design a flash translation layer that can hide the long garbage collection latency and provide a deterministic response time.

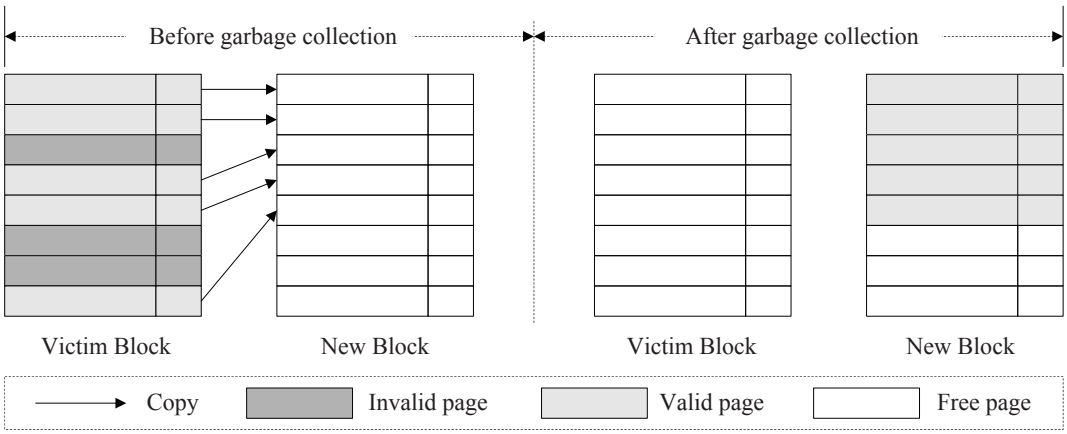


Figure 4.1: An illustration of garbage collection.

4.3 RFTL: Real-Time FTL

In this section, we describe details of the techniques for our RFTL scheme. We first propose the system architecture of a real-time flash memory storage system in Section 4.3.1. Then, we present the problem formulation and the address mapping approach for RFTL in Section 4.3.2 and Section 4.3.3, respectively. A real-time task scheduler and a new garbage collection policy are described in Section 4.3.4 and Section 4.3.5, respectively. Finally, we present the WCET analysis in Section 4.3.6.

4.3.1 Real-time Flash Memory Storage System Architecture

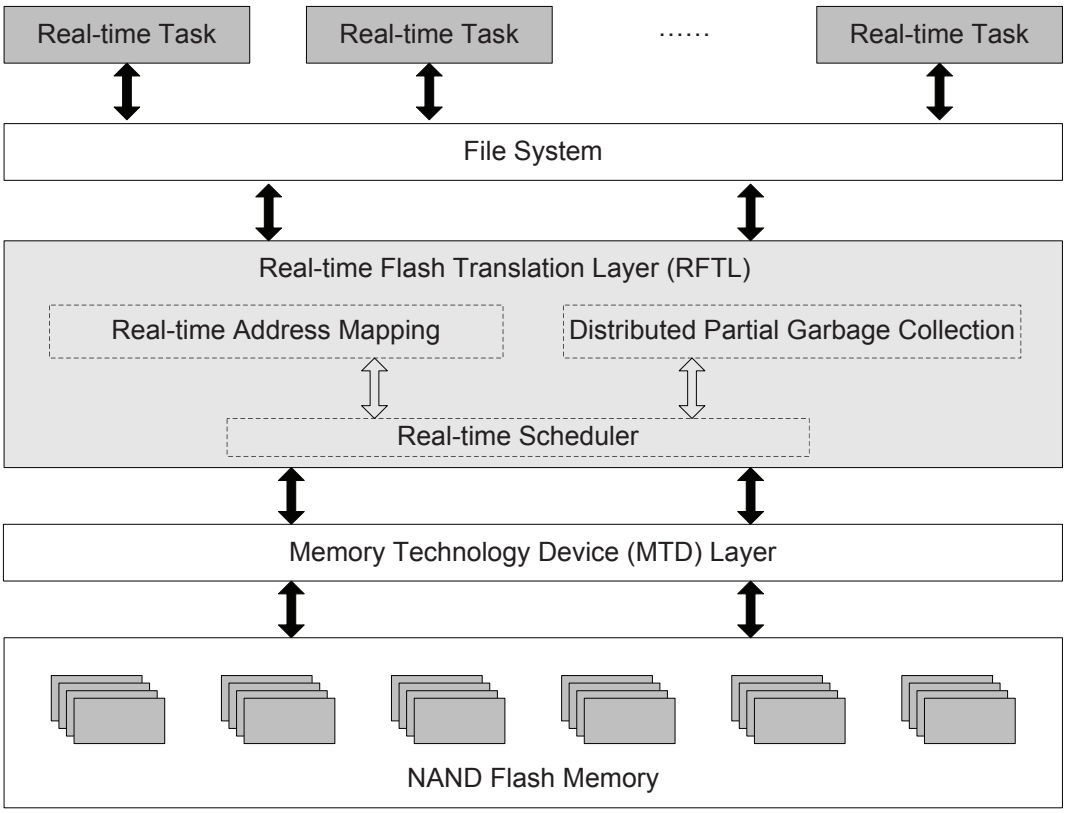


Figure 4.2: System architecture.

This section proposes the system architecture of a real-time NAND flash memory storage system, as shown in Figure 4.2. The system architecture is similar to the conventional NAND flash memory storage system shown in Chapter One, except that a conventional flash

memory storage system does not take into consideration real-time task, a real-time scheduler and a real-time garbage collection policy. We propose to support real-time services for real-time tasks by removing the unpredictability of the garbage collection overhead. A novel hybrid-level address mapping approach is designed to provide sufficient free space to serve the pending writes; meanwhile, a distributed partial garbage collection policy is proposed to reduce the worst-case block time for each write. A real-time scheduler is initiated to simultaneously serve the write and the garbage collection, while satisfying an upper bound to the response time that is close to an ideal case.

4.3.2 Problem Formulation

In order to remove the unpredictability, we model the NAND flash storage system as follows. Each I/O request issued from a file system to the FTL is modeled as an independent real-time task $T= \{p, e, d\}$, where p is the period, e is the execution time and d is the deadline. Without loss of generality, we assume that p is equal to d. Multiple I/O requests form a set of real-time tasks $V=\{T_1, T_2, ..., T_n\}$. There are two kinds of tasks in task set V: read request task $T_r=\{p_r, e_r, d_r\}$, and write request task $T_w=\{p_w, e_w, d_w\}$. p_r and p_w denote the frequency of a read or write request arriving from the file system. e_r represents the time taken to search for a target page, read the data from the page, and return a success or failure to the file system. e_w is the time overhead to search for a free page in which to store the data. The values of e_r and e_w are determined by the specific FTL. A lower bound on p (denoted as $L(p)$) gives the maximum request arrival rate that an FTL can handle. The upper bound on e (denoted as $U(e)$) shows the worst-case execution time for requests when no garbage collection is involved. From the perspective of the file system, $L(p)$ represents the worst-case response time when garbage collection is considered.

For the purpose of comparison, we first present a hypothetical ideal case as a baseline. In the ideal case, a read/write request task can be executed directly without any garbage collection involved. This is the best case scenario, and both the execution time and the response time are constant. Here, we only consider the flash operation time overhead since

the address translation overhead in RAM operations is at least an order of magnitude less than the flash operation time. The upper bounds on $U(e)$ in the ideal case are shown in Table 4.2. In the table, T_{rdoob} represents the time to read an OOB of a page. In the worst-case scenario, the execution of a read/write request task will be blocked by garbage collection. Note that, T_{er} is the longest atomic operation in flash media since the erase of one block cannot be interrupted. Therefore, T_{er} is the minimum time for which a request will be blocked and $L(p)$ should be T_{er} in the ideal case.

Table 4.2: Service guarantee bounds.

Bounds	Ideal	GFTL scheme [28]	RFTL scheme
$U(e_r)$	T_{rdpg}	$T_{rdpg}+\pi T_{rdoob}$	$T_{rdpg}+T_{rdoob}$
$U(e_w)$	T_{wrpg}	T_{wrpg}	$T_{wrpg}+T_{rdoob}$
$L(p)$	T_{er}	$T_{er}+max\{U(e_r),U(e_w)\}$	$max\{T_{er}+U(e_w),U(e_r)\}$

In this chapter, we design a real-time FTL scheme (called RFTL) that guarantees $U(e)$ for both reads and writes that are marginally T_{rdoob} larger than T_{er}. Our scheme provides service guarantees for requests that have a lower worst-case response time ($L(p)$) than GFTL [28], since $T_{rdpg}+\pi T_{rdoob}$ tends to be greater than $T_{wrpg}+T_{rdoob}$ according to the NAND flash specifications shown in Table 4.1.

Based on the model and problem analysis, we formulate the problem as follows:

Given a NAND flash memory chip and a task set $V=\{T_1, T_2, ..., T_n\}$, how can an FTL scheme be designed that can jointly schedule the requests and corresponding garbage collection operations such that a request can be executed within an upper bound $L(p)$ that is close to T_{er}?

4.3.3 Address Mapping in RFTL

In RFTL, we use a hybrid-level mapping approach. A logical page number (LPN) is divided into a logical block number (LBN) and a block offset (BO). A block mapping table is used to map a logical block with three physical blocks: the *primary block*, the *replacement block* and the *buffer block* as shown in Figure 4.3. Three indices that point to the next available page in each block are recorded in the table. The primary block is used first to serve the write requests, and the buffer block will serve the pending write requests when the primary block is full, while the replacement block provides a space to reclaim the primary block. These three blocks can periodically change their functions to provide guaranteed space for writes.

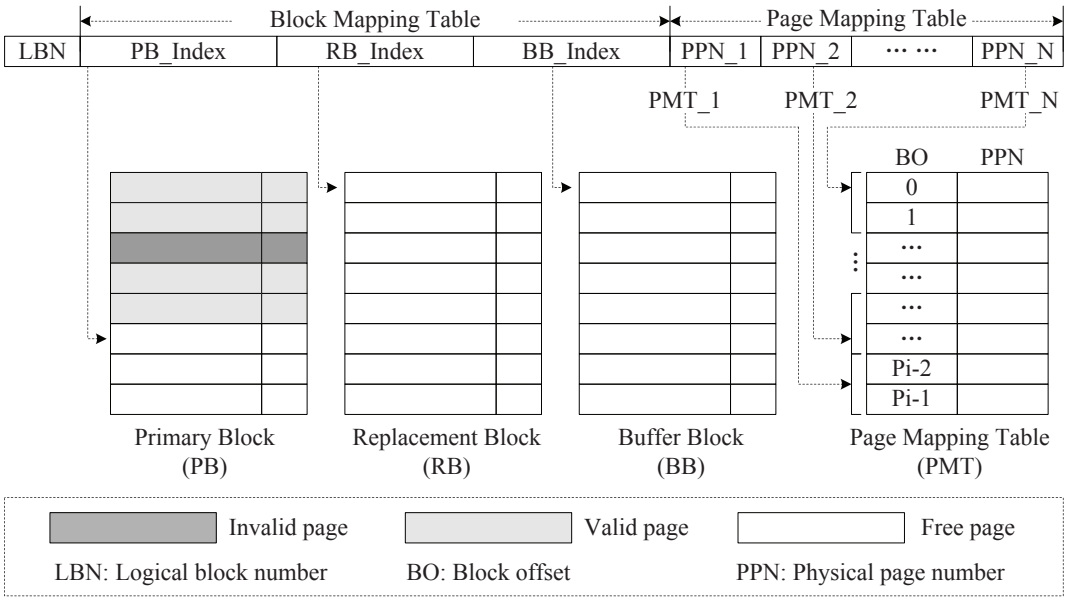


Figure 4.3: Address mapping in RFTL.

For each logical block, a page-level mapping table is used to map a logical page to a physical page that may belong to one of these three physical blocks. In order to reduce the RAM cost, the page mapping table is divided into *N* small tables, and each small table is stored in the OOB area of the newly allocated page. Suppose that each logical block and each physical block include π pages; the entire page-mapping table for a logical block then has π entries. Assume that the OOB area of a physical page can store α ($\pi \geq \alpha > 0$) entries of mapping slots; then the whole page mapping table is divided into *N* sub-tables according to

the logical page number, where $N=\lfloor \pi/\alpha \rfloor$. The N page mapping table indices are recorded in the RAM. Using the page-level mapping table indices, RFTL can obtain the address mapping information rapidly by reading one OOB.

4.3.4 Real-time Task Scheduler in RFTL

After obtaining the address mapping information, the read/write request should be serviced in three physical blocks. If no garbage collection is involved, RFTL will only execute this request in one period p. Otherwise, if the primary block is full and the garbage collection is invoked, the valid-page copy operations and the erase operation performed on the garbage collection are divided into partial steps, and the time taken to perform each step is no longer than the longest atomic operation in flash (that is the block erase operation T_{er}). In such a scenario, RFTL will first execute the request and then serve one partial garbage collection step in one period p.

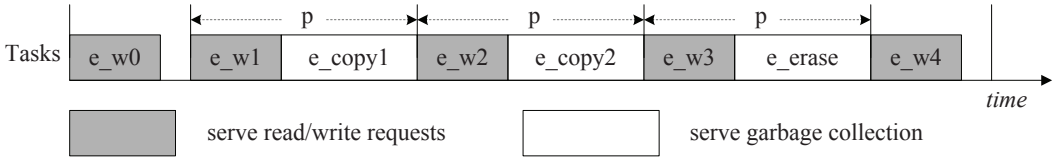


Figure 4.4: Task schedule in RFTL.

Figure 4.4 shows the task schedule policy of RFTL, in which the requests and the garbage collection can be alternatively scheduled. Five requests $w0$, $w1$, $w2$, $w3$, and $w4$ are mapped with the same primary block. $w0$ is scheduled directly since free space is available. When the primary block is full, the pending tasks are scheduled in each period p and the time cost to execute each task is e_w1, e_w2, and e_w3, respectively. In the time left for each period, the partial garbage collection operations of this primary block will be scheduled. In Figure 4.4, there are two copy operations and one erase operation. The time costs of these three operations are e_copy1, e_copy2 and e_erase, respectively. After garbage collection, the primary block becomes free and $w4$ can be scheduled.

A write request issued from the file system is represented by a data and a logical page

number (LPN), e.g., *write(D,126),* where *D* is the data and *126* is the LPN. When a write request is scheduled, the LPN is first translated to an LBN and a block offset (BO). Since three physical blocks are mapped to the logical block with LBN, the first write to the LBN is written to the first free page of the primary block, and the pages in the primary block are allocated sequentially from page 0. After π writes, the primary block becomes full, the buffer block will then serve the coming write requests, and the distributed partial garbage collection will be invoked simultaneously to reclaim the primary block. The buffer block serves as the buffer for requests from the time that the primary block becomes full until it is reclaimed. The valid pages in the primary block will be copied to the replacement block, where the copy operation can be interleaved with the requests. In the page copy process, a free page is guaranteed to be available in the buffer block to serve the requests simultaneously (to be explained in Section 4.3.6).

When a physical page is allocated to serve the write request, one mapping slot (BO, PBN) is formed. The corresponding sub-table and the data are written to the OOB area and data area, respectively. A page table index is stored in RAM to keep track of the mapping information. For a rewrite (update) operation, the out-of-date mapping slot needs to be read out from the OOB of the page pointed to by the pointers in RAM. The corresponding mapping slot will be updated and then written to the OOB of the new page. The page table index in RAM will also point to the new physical page. If a free page can always be guaranteed in the buffer block, the time to execute a write request is constant: T_{rdoob}+T_{wrpg} (one OOB read and one page write). The best-case response time is also T_{rdoob}+T_{wrpg}. In the worst case, when the partial garbage collection operation is scheduled, the worst-case response time is T_{er}+T_{rdoob}+T_{wrpg}.

A read request issued from the file system is represented by a logical page number (LPN), e.g., *read (36).* When a read request is scheduled, the LPN is first translated to an LBN and a BO. The corresponding LBN will be searched in the block mapping table in RAM. Then, the page mapping sub-table for the requested BO can be obtained using the page table index in RAM. From the sub-table, we can get the physical page that stores the requested data. Since no space is required in serving the read request, no partial garbage

collection is invoked. Therefore, the best-case response time and the worst-case response time of a read request are the same T_{rdoob}+T_{rdpg}.

4.3.5 RFTL with Distributed Garbage Collection

The garbage collection in RFTL is invoked once a primary block is full and a write request is issued to this primary block. Given a block with π pages, the garbage collection can be partitioned into k periods (steps) if all of the π pages are valid:

$$k = \lceil \pi \times (T_{rdpg} + T_{wrpg} + 2T_{rdoob}) + T_{er}/T_{er} \rceil \tag{4.1}$$

In one period *p*, the write request will first be serviced, and the execution time is e_w, where e_w=T_{wrpg}+T_{rdoob}. After the request is serviced, the time left in this period is t, where $t \geq T_e$. In time t, the garbage collection operations (valid-page copy or block erase) will be performed. For valid-page copy operations, suppose that the maximum number of pages that can be copied in this period is β, then:

$$\beta = \lfloor t/(T_{rdpg} + T_{wrpg} + 2T_{rdoob}) \rfloor \tag{4.2}$$

Figure 4.5 gives an example of the garbage collection process in RFTL. We assume that β=4 and k=3, which means that four valid-page copies can be finished in one period p and three periods are needed in the worst case. In Figure 4.5 (a), the primary block is full and garbage collection is triggered. Write request $w0$ is serviced in the first page of the buffer block; meanwhile, four valid pages in the primary block are copied to the replacement block after $copy0$, as shown in Figure 4.5 (b). After $w1$ is serviced, all of the valid pages in the primary block are copied into the replacement block by the $copy1$ operation. The primary block is erased after the write request $w2$ is serviced.

Exchange Operation After the primary block is reclaimed, an exchange operation is performed to change the position of the primary block and the replacement block as shown in Figure 4.5 (d). The new primary block will serve the coming requests if free space is

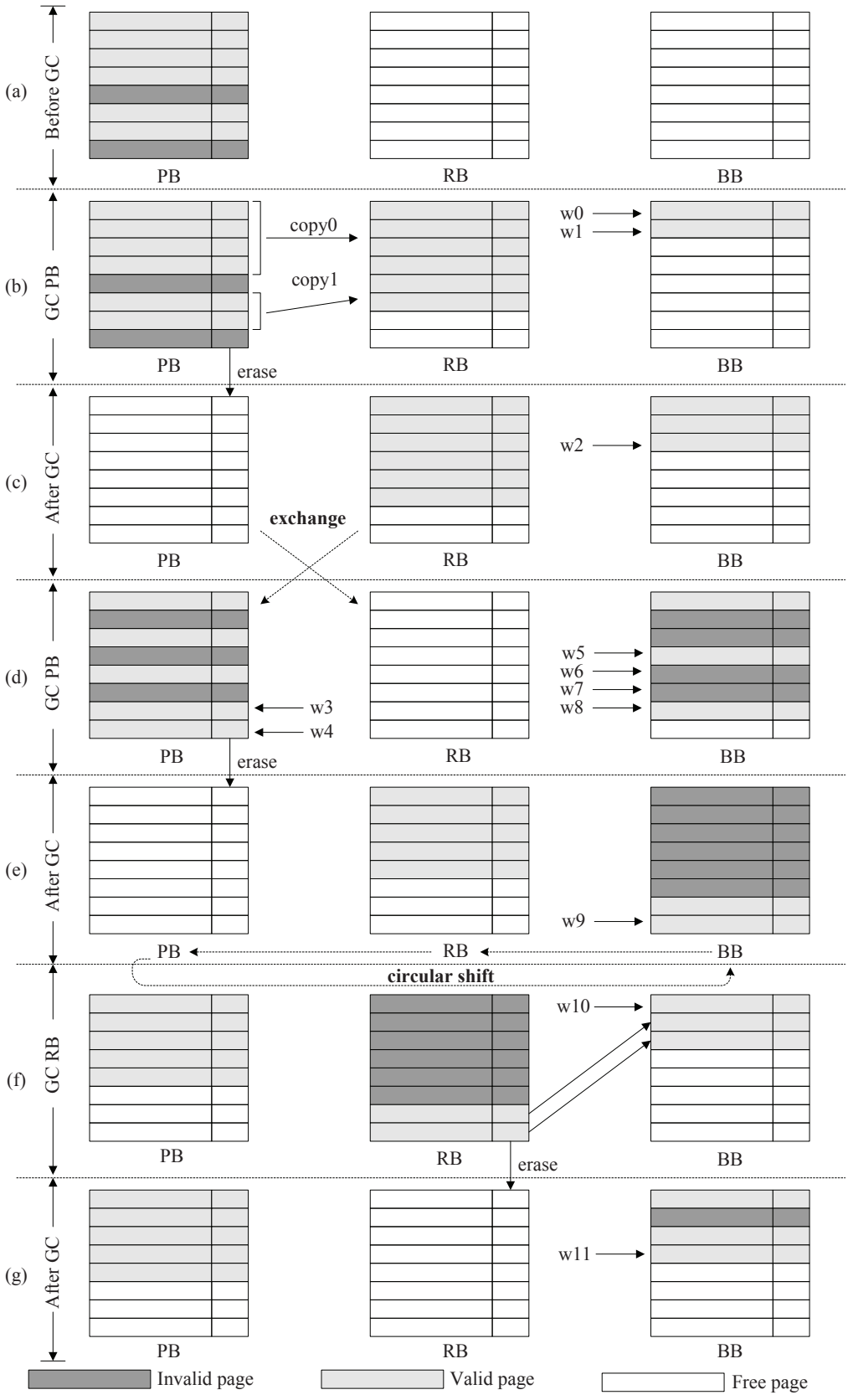


Figure 4.5: Garbage collection in RFTL.

available (i.e.,$w3$ and $w4$). After the primary block is full, the coming requests are written to the buffer block (i.e.,$w5$ and $w6$). When the buffer block has only k (i.e., k=3) free pages left, the partial garbage collection of the primary block is triggered again. The replacement block will store the valid pages from both the primary block and the buffer block. The partial garbage collection is interleaved with pending requests served in the buffer block (i.e.,$w7$, $w8$ and $w9$). After the buffer block is full, the primary block is free, as shown in Figure 4.5 (e).

Circular Shift Operation After the buffer block is full, a circular shift operation is taken to change the position of the three blocks. The free primary block will be reallocated as a buffer block, and the original buffer block is transferred to a new replacement block. The original replacement block will serve as the new primary block, as shown in Figure 4.5 (f). Partial garbage collection for the replacement block is triggered. Since the replacement block has k valid pages, the garbage collection can be split into j partial steps:

$$j = \lceil k \times (T_{rdpg} + T_{wrpg} + 2T_{rdoob})/T_{er} + 1 \rceil \tag{4.3}$$

Figure 4.5 (g) shows an example of the reclamation of the replacement block when j equals to two. The replacement block becomes free after two write requests $w10$ and $w11$ are served in the buffer block. The primary block can serve the requests again if free pages are included. A new garbage collection will be invoked if the new primary block is full and a new request wants to access this block.

In RFTL, garbage collection of one physical block is partitioned into multiple independent steps, and each step is triggered by one request. If the requests arrive and want to access the same logical block, the partial steps are performed consecutively within the physical blocks mapped to the same logical block. Otherwise, if the requests want to access different logical blocks, the garbage collection operations are correspondingly distributed to different logical blocks. In Figure 4.5, the garbage collection of the primary block or replacement block is triggered and finished by consecutive requests, which are mapped to the same logical block.

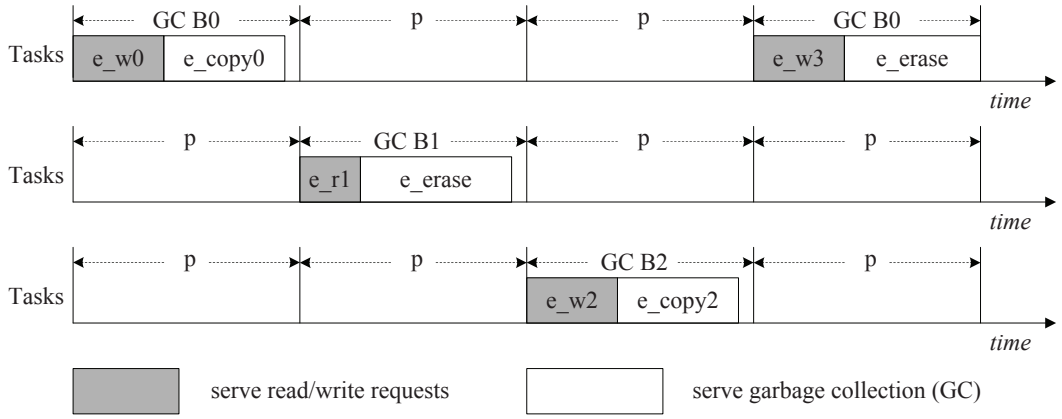


Figure 4.6: Distributed garbage collection in RFTL.

Figure 4.6 gives an example of garbage collection distributed to different logical blocks by the requests mapped to different logical blocks. We suppose that four requests $w0$, $w1$, $w2$, and $w3$ arrive sequentially. Write requests $w0$ and $w3$ are mapped to primary block $B0$, and $w1$ is mapped to primary block $B1$, while $w3$ is mapped to primary block $B2$. In the first period, the garbage collection of $B0$ is performed in which a valid-page copy, $copy0$, is executed after the schedule of request $w0$. In the second period, primary block $B1$ is reclaimed since the request $w1$ is mapped to it, and block erase operation $erase$ is executed. In period 4, primary block $B0$ is reclaimed again since the garbage collection is not finished in the first period. Two benefits can be achieved by distributed partial garbage collection. First, the long garbage collection latency can be fundamentally hidden, such that the worst-case response time of requests can be reduced to $L(p)$, where $L(p)$=$max\{T_{rdpg}+T_{rdoob}$, $T_{er}+T_{wrpg}+T_{rdoob}\}$. Second, the garbage collection overhead can be reduced since the valid page numbers in one block may decrease when the garbage collection is distributed. In other words, the change from reclaiming one block to a new block postpones the garbage collection of the old block. The postponed reclamation of the old block may reduce the number of valid page numbers within it, since a later rewrite operation may make the original valid page invalid. The average system response time is consequently reduced due to the decreased garbage collection overhead.

4.3.6 WCET Analysis in RFTL

Based on the distributed garbage collection policy, we can obtain the worst-case response time for requests in RFTL is $L(p)$ if enough free space can be guaranteed. In order to verify that the block management in RFTL can provide enough space for all requests, we present the worst-case analysis and give one theorem. The theorem gives the sufficient condition for a write request to be deterministically serviced.

Theorem 4.3.1. *The sufficient condition for providing a deterministic service for each request is that at least one free block and k free pages should be reserved when the distributed partial garbage collection is triggered.*

Proof. In the worst case, all pages in the victim block are valid pages. If the space reserved is less than one free block, there is no place to store at least one of the valid pages in the victim block. If fewer than k free pages are provided, at least one pending write will be blocked. □

Based on Theorem 4.3.1, we can get two lemmas for our scheme. The first lemma shows the sufficient condition for guaranteeing a deterministic service when doing partial garbage collection for one block with k valid pages. The second lemma presents the minimum number of blocks that are needed to guarantee the deterministic service.

Lemma 4.3.1. *Given a victim block with k valid pages, the sufficient condition for partial garbage collection to work is that at least k+j free pages should be reserved.*

Proof. In the worst-case scenario, enough free space should be guaranteed to store the k valid pages and the j pending writes that are interleaved with the partial garbage collection. Therefore, if less than $k + j$ space is provided, at least one valid page or one pending write will be blocked. □

In RFTL, the partial garbage collection of the replacement block is triggered after the circular shift operation. In the worst case, k valid pages need to be copied into buffer block. Since the buffer block can provide at least $2k+j$ free pages, the partial garbage collection can be guaranteed according to Lemma 4.3.1.

Lemma 4.3.2. *When distributed partial garbage collection is applied in block-level mapping schemes, the minimum number of blocks to guarantee deterministic service is 3.*

Proof. If one logical block is mapped to one physical block, no free space is provided to do a partial garbage collection. This violates the sufficient condition in Theorem 4.3.1. If one logical block is mapped to two physical blocks, only one free block is provided. This also violates the sufficient condition in Theorem 4.3.1. Therefore, in order to provide a deterministic service with distributed partial garbage collection, at least three blocks are needed. □

In RFTL, we adopt a block-level mapping approach in which one logical block is mapped to three physical blocks. Lemma 4.3.2 provides the guidelines on how to design a deterministic FTL scheme with a block-level mapping approach.

4.4 Evaluation

To evaluate the effectiveness of the proposed RFTL, we conduct a series of experiments and present the results with an analysis in this section. We compare and evaluate our proposed RFTL scheme over a well-known block-level FTL scheme (NFTL) [14], and a hybrid-level FTL scheme (GFTL) [28], in terms of the best-case system response time and the worst-case system response time. Besides, the distribution of the average system response time is also evaluated.

Table 4.3: Experimental setup.

Hardware	CPU	Intel Dual Core 2GHz
	Disk Space	200GB
	RAM	2GB
Simulation Environment	OS Kernel	Linux 2.6.17
	Flash Simulator	NAND flash simulator
	Flash Size	128/256/512MB

4.4.1 Experimental Setup

In the experiments, we developed a trace-driven NAND flash simulator under Linux kernel 2.6.17 and implemented three FTL schemes: GFTL [28], NFTL [14], and RFTL. The NFTL scheme is a general purpose block-level FTL scheme. GFTL is a representative deterministic FTL scheme. Therefore, we compare our scheme with NFTL and GFTL. Table 4.3 summarizes our experimental setup. Three NAND flash memory chips with a capacity of 128MB, 256MB, and 512MB, respectively are simulated. To conduct a fair comparison with different FTL schemes, we consider only a portion of flash as the active region in which our workloads are stored. The remaining flash is assumed to contain cold data or free blocks that are not under consideration. The framework of our simulation platform, as shown in Figure 4.7, consists of two modules: a NAND flash simulator providing basic read, write, and erase capabilities; and a desired flash translation layer management scheme that can be executed on top of the NAND flash simulator. The traces, along with various flash parameters such as block size and page size, page read time and page write time, and so on, are fed into our simulation framework. We can get the simulation results after running the NAND flash simulator. The parameters in our simulation are based on the flash memory data sheet values shown in Table 4.1.

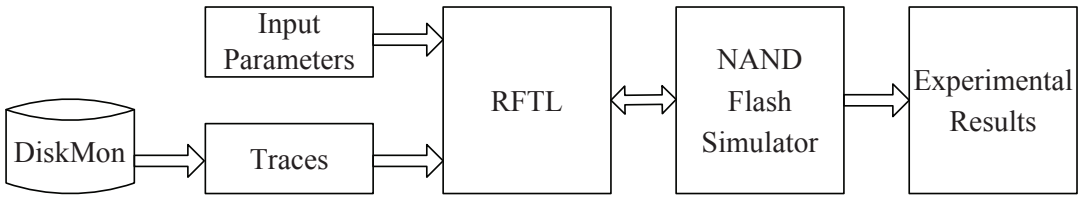


Figure 4.7: The framework of the simulation platform.

We use the following benchmarks from both the real-world and the synthetic traces to study the system performance for different FTL schemes. *Multimedia* is a real-world trace that we obtained from a notebook with Windows XP on an NTFS file system downloading and playing multimedia files (e.g., Movie, MP3). It consists of 1,633,269 write requests and 1,002,748 read requests. *Financial* is a well-known, write-dominant I/O trace obtained from an OLTP application running at a financial institution [5]. It consists of 4,099,354 write requests and 1,235,633 read requests. In order to perform a rigorous evaluation of different schemes, each read/write request in the traces is simulated with a periodicity of $L(p)$ without any idle period involved.

4.4.2 Results and Discussion

In this section, we present the simulation results of the proposed RFTL scheme, GFTL scheme, and NFTL scheme in terms of real-time and average performance as well as the space overhead (RAM cost and flash memory cost).

Table 4.4 presents the best-case and the worst-case system response time of the RFTL scheme for the two traces based on varying flash utilizations (%) and numbers of pages per block (π). The first two columns under R_{best} and R_{worst} denote the best-case and the worst-case response time for read requests, respectively. The next two columns, W_{best} and W_{worst}, represent the best-case and the worst-case response time, respectively, for write requests. Based on Table 4.4, we can observe that the worst-case response time for a read request is 50μs, which is equal to $T_{rdoob}+T_{rdpg}$. For a write request, the worst-case response time is 2325μs, which is equal to $T_{er}+T_{rdoob}+T_{wrpg}$. The worst-case response time for a read

request and write request is independent of the flash utilization and the flash size. It presents no variation when the flash utilization and the page size per block (π) vary. This observation shows that our scheme can provide a guaranteed service for different flash specifications and different traces.

Table 4.4: Best-case and worst-case system response times for RFTL.

Benchmarks	%	π	R_{best} (μs)	R_{worst} (μs)	W_{best} (μs)	W_{worst} (μs)
Multimedia	50	32	50	50	325	2,325
	50	64	50	50	325	2,325
	50	128	50	50	325	2,325
	100	32	50	50	325	2,325
	100	64	50	50	325	2,325
	100	128	50	50	325	2,325
Financial	50	32	50	50	325	2,325
	50	64	50	50	325	2,325
	50	128	50	50	325	2,325
	100	32	50	50	325	2,325
	100	64	50	50	325	2,325
	100	128	50	50	325	2,325

Table 4.5 shows the average system response time for the RFTL scheme under varying flash utilization ratios (%) and numbers of pages per block (π). Columns under R_{avg} and W_{avg} represent the average system response time for a read request and write request, respectively. The average response time for all requests, the total number of valid-page copy operations, and the total number of erase operations are also measured, and are denoted as T_{avg}, Σ_{cp}, and Σ_{er}, respectively. From the results, we can see that, the average response

Table 4.5: Average system response time for RFTL.

Benchmarks	%	π	R_{avg} (μs)	W_{avg} (μs)	T_{avg} (μs)	Σ_{cp}	Σ_{er}
Multimedia	50	32	50	400	335	137,630	69,142
	50	64	50	359	298	66,508	33,296
	50	128	50	339	280	32,414	16,208
	100	32	50	419	341	270,903	205,297
	100	64	50	375	303	131,281	99,021
	100	128	50	353	285	64,295	48,367
Financial	50	32	50	389	274	31,822	26,943
	50	64	50	354	248	15,445	13,049
	50	128	50	338	236	7,488	6,285
	100	32	50	390	271	68,687	79,720
	100	64	50	355	245	33,409	38,714
	100	128	50	337	232	16,381	18,812

time for read requests is close to the best-case response time, and the average response time for write requests is close to the worst-case response time. This is because that few valid-page copy operations or block erase operations are involved in one period p. This verifies that the distributed garbage collection can provide enough space to serve the continuous incoming requests. The average response time for each trace is decreased, while the number of valid-page copy and block erase operations are reduced as the flash size increases (e.g., as π increases from 32 to 128). This is based on the fact that more free flash space will lead to less garbage collection when the same number of requests are serviced. Moreover, the valid-page copy and block erase operation are increased when the flash utilization is increased for a fixed flash size. This is due to the fact that the amount of free space shrinks when the flash

continually serves the write request. More garbage collection will be invoked to reclaim the obsolete pages, which increases the average system response time.

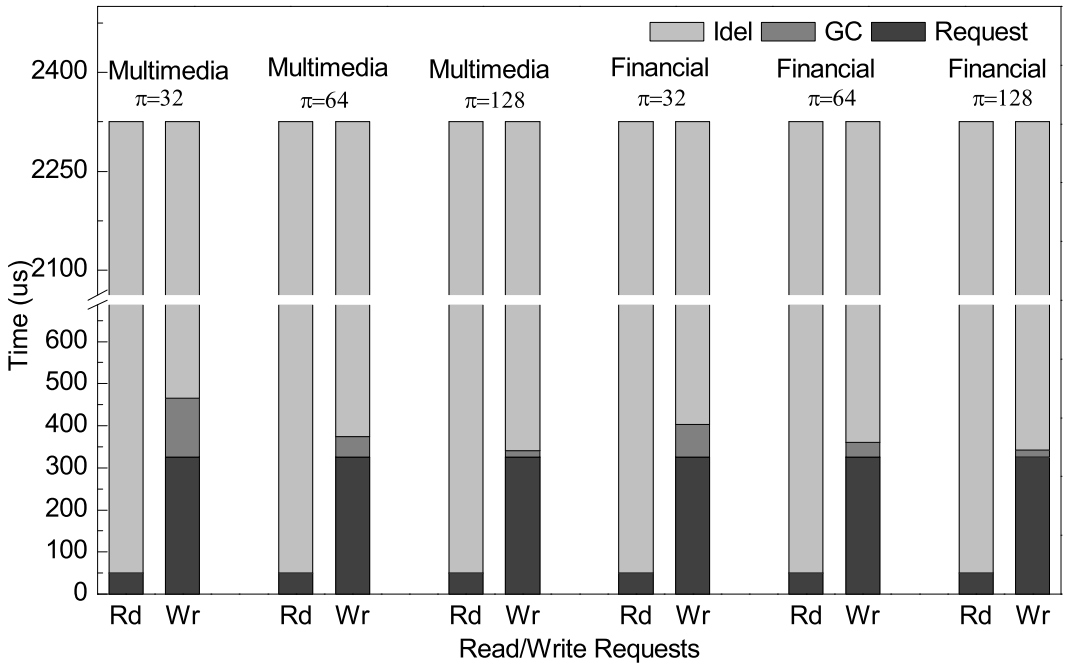


Figure 4.8: Average time distribution per period in RFTL.

Figure 4.8 shows the distribution of the request service time and the garbage collection (GC) overhead in one period p. The total length of a bar represents the upper bound of the response time, which is $L(p)$ as mentioned in Table 4.2. The "Request" bar denotes the execution time of the request, and the "GC" bar represents the average time cost in garbage collection, which includes a series of valid-page copy and block erase operations. The time left is the idle time. Since the total length of the bar is calculated under the worst-case scenario, from the results we can see that a large amount of time is idle in one period. The idle time increases when the value of π is increased. This is because more space is provided, leading to less garbage collection overhead. Note that in case of a read request, much more idle time is left than in the case of a write request for both traces. This is because the time cost to execute a page read is less than that for a page write. In particular, we find that the garbage collection time is zero in one period for all read requests. This is because no partial garbage collection is scheduled when a read request is executed. This schedule policy can

delay the garbage collection time. Thus, the number of invalid pages may be increased and the number of valid pages within the victim block is reduced correspondingly, allowing the average system response time to be improved. From the figure, we can also observe that the idle time for the trace $Financial$ is longer than that of the trace $Multimedia$. This is because the trace $Financial$ follows a higher temporal locality and more update operations occur, resulting in fewer valid-page copy operations in garbage collection.

Table 4.6: Performance for RFTL, GFTL and NFTL.

Traces	Metrics	FTL Schemes		
		RFTL scheme	GFTL scheme	NFTL scheme
Multimedia	T_{worst} (μs)	2,325	3,650	4,335
	T_{avg} (μs)	303	525	321
	Σ_{cp}	1.31e5	5.38e5	3.95e5
	Σ_{er}	0.99e5	1.29e5	0.48e5
	Σ_{oob}	0.05e8	0.29e8	1.37e8
	$L(p)$ (μs)	2,325	3,650	4,335
Financial	T_{worst} (μs)	2,325	3,650	4,557
	T_{avg} (μs)	245	2,997	522
	Σ_{cp}	0.03e6	7.65e7	0.38e7
	Σ_{er}	0.38e5	6.60e5	1.23e5
	Σ_{oob}	0.02e8	0.40e8	2.86e8
	$L(p)$ (μs)	2,325	3,650	4,557

Table 4.6 compares the system performance of RFTL, GFTL, and NFTL under the same flash size and space utilization ratio. Both RFTL and GFTL show great improvement in

the worst-case response time compared with the NFTL scheme, which is a general purpose FTL. In the NFTL scheme, two block merge operations are involved and the blocked time of each request in the worst case is at least 2*T_{er}. In the GFTL scheme and RFTL scheme, one victim block reclamation is needed and the garbage collection is partitioned into multiple small steps. Therefore, they have lower worst-case response time than the NFTL scheme. Note that RFTL achieves a 36.30% improvement in the worst-case response time compared to GFTL, which means that RFTL can accept requests at a higher arrival rate while providing read/write service guarantees. This is based on the fact that in GFTL it is necessary to search all of the OOB area of one block in order to read the valid page. But in RFTL, the address mapping information can be obtained by reading one OOB area.

The RFTL scheme a shows better average response time than the NFTL scheme, while GFTL has the longest average response time. In order to provide enough space to serve the real-time requests, the GFTL scheme invokes the garbage collection once a block becomes full. The reclamation of one block is performed in a concentrated manner, which incurs many unnecessary valid page copies and unnecessary block erase overhead. As shown in Table 4.6, this extra overhead significantly increases the average response time compared with NFTL and RFTL. In the RFTL scheme, the partial garbage collection is distributed to each logical block in an on-demand fashion. The valid page copy and block erase operations are performed only when needed. The delayed reclamation reduces the number of valid page copies and block erases. Therefore, RFTL achieves a 67.06% improvement in average response time compared with GFTL.

4.4.3 Overhead

In order to provide a deterministic service, both GFTL and RFTL introduce extra flash space to serve as the write buffer for partial garbage collection. The number of buffer blocks required for GFTL is the same as that of data blocks, while RFTL needs double the number of data blocks to serve as replacement blocks and buffer blocks. Although RFTL has more overhead in terms of flash space, it shows a great reduction in the much more valuable RAM

space. Given a large-block based 128MB NAND flash with 64 pages per block, RFTL requires 16KB (16B*1024) of RAM space to store the block mapping table and page mapping table index. For the GFTL scheme, the RAM cost is 195KB, which consists of three parts: the block level mapping table for data blocks (3KB), the page mapping table for buffer blocks (64KB), and one block buffer in RAM (128KB). RFTL shows a 91.79% reduction in RAM cost compared with GFTL.

4.5 Summary

In this chapter, we proposed a real-time flash translation layer (called RFTL) for NAND flash memory storage systems, which can provide real-time service guarantees by hiding the long garbage collection latency. To achieve this, a novel hybrid-level address mapping approach was designed to provide enough free space to serve the pending writes. Meanwhile, a distributed garbage collection policy was introduced to reduce the worst-case response time. A real-time scheduler was in charge of coordinating the writes and the garbage collection so that an upper bound to the response time could be obtained. In order to evaluate the system performance of our scheme, we conducted a series of experiments. The experimental results showed that our scheme can achieve a 36.30% improvement in the upper bound of the worst-case response time for requests compared with GFTL. Moreover, we achieved a 67.06% reduction in average system response time and a 91.79% reduction in RAM cost compared with GFTL.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

NAND flash memory has been widely adopted in storage systems for various embedded systems and consumer-electronic products, due to its advantages in non-volatility, shock-resistance and low-power consumption. With the fast development of flash memory technology, the capacity of flash memory chips has been increasing dramatically, particularly with the advent of MLC NAND flash memory technology. The increased capacity of the NAND flash memory chip poses new challenges for its management. Flash translation layers suffer from a large RAM footprint problem for address management. Moreover, most existing FTLs are designed for SLC flash, and become inapplicable or inefficient when applied to MLC flash. In this thesis, we investigated several challenging issues in designing the FTL schemes for NAND flash memory storage system in resource-constrained embedded systems. In particular, we proposed three techniques to optimize the system performance from three aspects including the RAM cost, garbage collection overhead, and real-time storage performance.

- First, we proposed a demand-based block-level address mapping scheme with a two-level caching mechanism, named DAC, to reduce the RAM footprint on address mapping management for large-scale NAND flash memory storage systems in resource-constrained embedded systems. In our DAC, the large address mapping table is stored in the flash memory chip and only a small number of active mapping entries are cached in RAM so that the RAM footprint can be reduced. In order to reduce the extra address

translation overhead caused by the on-demand address mapping scheme, a two-level caching mechanism was proposed to improve the cache hit ratio, in which the reference locality and the access frequency of workloads were explored. Different cache replacement policies are initiated for different caches, so that a higher cache hit ratio and lower kick-out overhead can be achieved. We conducted experiments on a set of traces collected from real workloads. The experimental results showed that our technique can achieve a 91.68% reduction in RAM cost, while the average response time presents an average degradation of 2.02% compared with previous work.

- Second, we proposed a novel flash translation layer (FTL) called MNFTL for MLC NAND flash memory storage systems, to reduce the garbage collection overhead while hiding the new write constraints in MLC flash. We analyzed several fundamental problems in the design of the MLC flash translation layer, and observed that valid page copies cause the garbage collection overhead. In our MNFTL, the garbage collection overhead reduction is achieved by concentrating the invalid pages closer to each other, while postponing the time to do the block reclamation. In this way, the valid page numbers within a victim block can be minimized and the number of valid page copies can be reduced. In our approach, concentrated mapping uses the page-level mapping approach, so the write constraints of MLC NAND flash can be satisfied. The corresponding page-level mapping table is stored in the spare area of the newly allocated pages, while the page mapping table indices are recorded in the RAM. Therefore, limited RAM space is used. We conducted experiments on a set of benchmarks. The experimental results showed that our scheme presents a reduction of 30.09% in the average system response time compared with previous work.

- Third, we proposed a real-time flash translation layer, named RFTL, to reduce the worst-case system response time and the average system response time of NAND flash memory storage systems in real-time embedded systems. In RFTL, the improvement in performance is achieved by cutting the long garbage collection process into small partial steps and interleaving each small step with the pending write requests. Through

the proposed distributed partial garbage collection policy, the response time of a pending write request is decreased and an upper bound to the worst-case response time that close to being an ideal case is obtained. Meanwhile, the average system response time is also reduced due to the postponed reclamation introduced by the partial garbage collection. The experimental results showed that our scheme can achieve a 36.30% improvement in the upper bound of the worst-case response time for requests compared with GFTL. Moreover, we achieved a 67.06% reduction in average system response time and a 91.79% reduction in RAM cost compared with GFTL.

5.2 Future Work

The work presented in this thesis can be extended in different directions in the future.

- First, the two-level caching mechanism proposed in this thesis mainly focuses on block-level FTL designs, and we can further apply it to hybrid-level FTLs. Compared with block-level FTL, hybrid-level FTLs have better address mapping efficiency and flexibility. However, they have a much larger RAM footprint than block-level FTL. Applying the demand-based address mapping scheme to these schemes can reduce the RAM cost and further improve the address mapping flexibility as well as the average system response time. Moreover, the proposed two-level caching mechanism can also be used to overcome the drawbacks in the demand-based page-level DFTL scheme. As discussed in this thesis, the one-level cache design in DFTL suffers from a lower cache hit ratio and more expensive overhead on translation block management. How to design a two-level caching mechanism in the DFTL scheme to improve the system performance is a future endeavor.

- Second, the power failure problem was not studied in this thesis. As the address mapping table is working and maintained in the RAM when a flash memory chip is in normal working mode, we may lose the most-updated mapping entries when a power failure occurs. A promising main memory alternative, Phase Change Memory (PCM),

can provide a non-volatile storage service, which can overcome the power failure problem. Therefore, we can further explore the possibility of storing the address mapping table in PCM. The endurance of PCM and the implementation of two-level caching with PCM are issues we need to address.

- Third, this work only focuses on the optimizing techniques for SLC or MLC NAND flash memory storage systems. Since SLC and MLC flash have different properties and distinct performances, an SLC/MLC hybrid-architecture NAND flash memory storage system may provide better storage performance after adopting the advantages of the two technologies. Therefore, it is interesting to extend our techniques to optimizing the hybrid-architecture storage system.

- Finally, a possible research direction is to use main memory data compression techniques to manage the large address mapping table in RAM, so that the RAM cost can be reduced.

REFERENCES

[1] DiskMon for Windows. *http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx*.

[2] Intel Corporation. Understanding the flash translation layer (FTL) specification. *http://developer.intel.com*.

[3] Micron 32GB Mass Storage. *http://www.micron.com/products/partdetail?part=MT29F32G08CBABAWP*.

[4] NAND flash scalability. *http://www.wikipedia.org/wiki/Flash-memory/cite-note-29*.

[5] OLTP trace from umass trace repository. *http://traces.cs.umass.edu/index.php/Storage*.

[6] Samsung Electronics. K9LBG08U0M(v1.0)-32GB DDP MLC data sheet. *http://www.samsung.com*.

[7] SAMSUNG NAND flash. *http://www.samsung.com/global/business/semiconductor*.

[8] Websearch trace from umass trace repository. *http://traces. cs.umass.edu/index.php/Storage/Storage*.

[9] Airlines electronic engineering committee (AEEC). *ARINC Specification*, 651, 1991.

[10] Memory Technology Device (MTD) Subsystem for Linux. *http://www.linux-mtd.infradead.org/*, 2009.

[11] Samsung Electronics. Page program addressing for MLC NAND application note. *http://www.samsung.com*, 2009.

[12] *http://www.pcmag.com/article2/0,2817,2397287,00.asp*, 2011.

[13] Amir Ban. Flash file system. *US patent 5,404,485*, April 4, 1995.

[14] Amir Ban. Flash-memory translation layer for NAND flash (NFTL). *M-systems*, 1998.

[15] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *SIGOPS Operating System Review*, 41(2):88–93, 2007.

[16] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *SIGPLAN Note*, 44(3):217–228, March 2009.

[17] 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, 2007.

[18] Li-Pin Chang. Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASP-DAC '08)*, pages 428–433, 2008.

[19] Li-Pin Chang and Tei-Wei Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '02)*, pages 187–196, 2002.

[20] Li-Pin Chang and Tei-Wei Kuo. A real-time garbage collection mechanism for flash-memory storage systems in embedded systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA' 02)*, pages 409–430, 2002.

[21] Li-Pin Chang and Tei-Wei Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *ACM Transactions on Storage*, 1(4):381–418, 2005.

[22] 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 Transactions on Embedded Computing Systems*, 3(4):837–863, 2004.

[23] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *Proceedings of the 44th Annual Conference on Design Automation (DAC '07)*, pages 212–217, 2007.

[24] Yuan-Hao Chang and Tei-Wei Kuo. A commitment-based management strategy for the performance and reliability enhancement of flash-memory storage systems. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*, pages 858–863, 2009.

[25] 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, 2009.

[26] 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 393 –398, 2009.

[27] Siddharth Choudhuri and Tony Givargis. Performance improvement of block based NAND flash translation layer. In *Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pages 257–262, 2007.

[28] Siddharth Choudhuri and Tony Givargis. Deterministic service guarantees for NAND flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '08)*, pages 19–24, 2008.

[29] 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, 2009.

[30] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[31] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: a hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*, pages 664–469, 2009.

[32] Cagdas Dirik and Bruce Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*, pages 279–289, 2009.

[33] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 229–240, 2009.

[34] Long-zhe Han, Yeonseung Ryu, Tae-sun Chung, Myungho Lee, and Sukwon Hong. An intelligent garbage collection algorithm for flash memory storages. In *Proceedings of the 6th international conference on Computational Science and Its Applications - Volume Part I (ICCSA '06)*, pages 1019–1027, 2006.

[35] Jen-Wei Hsieh, Li-Pin Chang, and Tei-Wei Kuo. Efficient on-line identification of hot data for flash-memory management. In *Proceedings of the 2005 ACM symposium on Applied computing (SAC '05)*, pages 838–842, 2005.

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

[37] Jen-Wei Hsieh, Tei-Wei Kuo, Po-Liang Wu, and Yu-Chung Huang. Energy-efficient and performance-enhanced disks using flash-memory cache. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, pages 334–339, 2007.

[38] Jingtong Hu, Chun Jason Xue, Wei-Che Tseng, Yi He, Meikang Qiu, and Edwin H.-M. Sha. Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation. In *Proceedings of the 47th Design Automation Conference (DAC '10)*, pages 350–355, 2010.

[39] Po-Chun Huang, Yuan-Hao Chang, Tei-Wei Kuo, Jen-Wei Hsieh, and Miller Lin. The behavior analysis of flash-memory storage systems. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08)*, pages 529–534, 2008.

[40] Yazhi Huang, Tiantian Liu, and Chun Jason Xue. Register allocation for write activity minimization on non-volatile main memory. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC '11)*, pages 129–134, 2011.

[41] Seunghwan Hyun, Sehwan Lee, Sungyong Ahn, Hyokyung Bahn, and Kern Koh. Vector read: Exploiting the read performance of hybrid nand flash. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*, pages 177–184, 2008.

[42] Soojun Im and Dongkun Shin. Storage architecture and software support for SLC/MLC combined flash memory. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*, pages 1664–1669, 2009.

[43] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*, pages 31–42, 2002.

[44] Yongsoo Joo, Youngjin Cho, Donghwa Shin, and Naehyuck Chang. Energy-aware data compression for multi-level cell (MLC) flash memory. In *Proceedings of the 44th annual Design Automation Conference (DAC '07)*, pages 716–719, 2007.

[45] Yongsoo Joo, Yongseok Choi, Chanik Park, Sung Woo Chung, EuiYoung Chung, and Naehyuck Chang. Demand paging for oneNAND flash execute-in-place. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS '06)*, pages 229–234, 2006.

[46] 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, 2007.

[47] Sanghyuk Jung, Jin Hyuk Kim, and Yong Ho Song. Hierarchical architecture of flash-based storage systems for high performance and durability. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*, pages 907–910, 2009.

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

[49] 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 Transactions on Computers*, 58(6):744–758, 2009.

[50] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES '06)*, pages 103–112, 2006.

[51] 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, 2008.

[52] 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 1–14, 2008.

[53] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Disk schedulers for solid state drivers. In *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09)*, pages 295–304, 2009.

[54] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[55] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[56] Jinkyu Kim, Hyunggyu Lee, Shinho Choi, and Kyoung Il Bahng. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT '08)*, pages 31–40, 2008.

[57] Sungchan Kim, Chanik Park, and Soonhoi Ha. Architecture exploration of NAND flash-based multimedia card. In *Proceedings of the conference on Design, automation and test in Europe (DATE '08)*, pages 218–223, 2008.

[58] Tei-Wei Kuo, Yuan-Hao Chang, Po-Chun Huang, and Che-Wei Chang. Special issues in flash. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '08)*, pages 821–826, November 2008.

[59] Ohhoon Kwon and Kern Koh. Swap-aware garbage collection for NAND flash memory based embedded systems. In *Proceedings of the 7th IEEE International Conference on Computer and Information Technology (CIT '07)*, pages 787–792, 2007.

[60] Ohhoon Kwon, Jaewoo Lee, and Kern Koh. Ef-greedy: A novel garbage collection policy for flash memory based embedded systems. In *Proceedings of the 7th international conference on Computational Science, Part IV: (ICCS'07)*, pages 913–920, 2007.

[61] Ohhoon Kwon, Yeonseung Ryu, and Kern Koh. An efficient garbage collection policy for flash memory based swap systems. In *Proceedings of the 2007 international conference on Computational science and its applications - Volume Part I (ICCSA'07)*, pages 213–223, 2007.

[62] Chul Lee, Sung Hoon Baek, and Kyu Ho Park. A hybrid flash file system based on NOR and NAND flash memories for embedded devices. *IEEE Transactions on Computers*, 57(7):1002–1008, July 2008.

[63] Jongmin Lee, Sunghoon Kim, Hunki Kwon, Choulseung Hyun, Seongjun Ahn, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Block recycling schemes and their cost-based optimization in NAND flash memory based storage system. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT '07)*, pages 174–182, 2007.

[64] Kwangyoon Lee and Alex Orailoglu. Application specific low latency instruction cache for NAND flash memory based embedded systems. In *Proceedings of the 2008 Symposium on Application Specific Processors (SASP '08)*, pages 69–74, 2008.

[65] Kwangyoon Lee and Alex Orailoglu. Application specific non-volatile primary memory for embedded systems. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '08)*, pages 31–36, 2008.

[66] 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, 2009.

[67] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, pages 1075–1086, 2008.

[68] 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 Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[69] Adam Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[70] Han-Lin Li, Chia-Lin Yang, and Hung-Wei Tseng. Energy-aware flash memory management in virtual memory system. *IEEE Transactions on Very Large Scale Integration Systems*, 16(8):952–964, August 2008.

[71] Yu Li, Jianliang Xu, Byron Choi, and Haibo Hu. StableBuffer: optimizing write performance for DBMS applications on flash devices. In *Proceedings of the 19th ACM international conference on Information and knowledge management (CIKM '10)*, pages 339–348, 2010.

[72] Duo Liu, Yi Wang, Zhiwei Qin, Zili Shao, and Yong Guan. A space reuse strategy for flash translation layers in SLC NAND flash memory storage systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–14, 2011.

[73] Zhanzhan Liu, Lihua Yue, Peng Wei, Peiquan Jin, and Xiaoyan Xiang. An adaptive block-set based management for large-scale flash memory. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*, pages 1621–1625, 2009.

[74] Sai 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, 2009.

[75] Sai Tung On, Yinan Li, Bingsheng He, Ming Wu, Qiong Luo, and Jianliang Xu. FD-buffer: a buffer manager for databases on flash disks. In *Proceedings of the 19th ACM international conference on Information and knowledge management (CIKM '10)*, pages 1297–1300, 2010.

[76] Veera Papirla and Chaitali Chakrabarti. Energy-aware error control coding for flash memories. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*, pages 658–663, 2009.

[77] 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 Transactions on Embedded Computing Systems*, 7(4):1–23, 2008.

[78] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim, and Bumsoo Kim. Cost-efficient memory architecture design of NAND flash memory embedded systems. In *Proceedings of the 21st International Conference on Computer Design (ICCD '03)*, pages 474–483, 2003.

[79] Dongchul Park, Biplob Debnath, and David Du. CFTL: a convertible flash translation layer with consideration of data access patterns. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS' 10)*, pages 365–366, 2010.

[80] Kwanghee Park, Junsik Yang, Joon-Hyuk Chang, and Deok-Hwan Kim. Anticipatory I/O management for clustered flash translation layer in NAND flash memory. *Electronics and Telecommunications Research Institute Journal*, 30(6):790–798, 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, 2006.

[82] Youngwoo Park, Seung-Ho Lim, Chul Lee, and Kyu Ho Park. PFFS: a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*, pages 1498–1503, 2008.

[83] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. Demand-based block-level address mapping in large-scale NAND flash memory storage systems. In *Proceedings of the 8th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '10)*, pages 173–182, 2010.

[84] 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 *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*, pages 157–166, Apr. 2011.

[85] 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 Annual Design Automation Conference (DAC '11)*, pages 17–22, 2011.

[86] Liang Shi, Chun Jason Xue, and Xuehai Zhou. Cooperating write buffer cache and virtual memory management for flash memory based systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*, pages 147–156, 2011.

[87] Hyungkeun Song, Sukwon Choi, Hojung Cha, and Rhan Ha. Improving energy efficiency for flash memory based embedded applications. *Journal of Systems Architecture*, 55(1):15–24, January 2009.

[88] Jinsun Suk and Jaechun No. Performance analysis of NAND flash-based SSD for designing a hybrid filesystem. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications (HPCC '09)*, pages 539–544, 2009.

[89] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Xie Yuan, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA '10)*, pages 1–12, 2010.

[90] Sheng-Jie Syu and Jing Chen. An active space recycling mechanism for flash storage systems in real-time application environment. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '05)*, pages 53–59, 2005.

[91] Yi-Lin Tsai, Jen-Wei Hsieh, and Tei-Wei Kuo. Configurable NAND flash translation layer. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC' 06)*, pages 118–127, 2006.

[92] Hung-Wei Tseng, Han-Lin Li, and Chia-Lin Yang. An energy-efficient virtual memory system with flash memory as the secondary storage. In *Proceedings of the 2006 international symposium on Low power electronics and design (ISLPED '06)*, pages 418–423, 2006.

[93] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 35th SIGMOD international conference on Management of data (SIGMOD '09)*, pages 59–72, 2009.

[94] Yi Wang, Duo Liu, Zhiwei Qin, and Zili Shao. An endurance-enhanced flash translation layer via reuse for NAND flash memory storage systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '11)*, pages 1–6, Mar. 2011.

[95] 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 2010 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '10)*, pages 163–172, 2010.

[96] Chin-Hsien Wu. A time-predictable system initialization design for huge-capacity flash-memory storage systems. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '08)*, pages 13–18, 2008.

[97] Chin-Hsien Wu. An energy-efficient i/o request mechanism for multi-bank flash-memory storage systems. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):6:1–6:25, January 2009.

[98] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient B-Tree layer for flash-memory storage systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA '03)*, pages 409–430, February 2003.

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

[100] Chin-Hsien Wu, Tei-Wei Kuo, and Chia-Lin Yang. Energy-efficient flash-memory storage systems with an interrupt-emulation mechanism. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04)*, pages 134–139, 2004.

[101] Chin-Hsien Wu, Tei-Wei Kuo, and Chia-Lin Yang. A space-efficient caching mechanism for flash-memory address translation. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06)*, pages 64–71, 2006.

[102] 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 507 –512, 2009.

[103] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design Test of Computers*, 28(1):44–51, Jan.–Feb. 2011.

[104] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row buffer locality-aware data placement in hybrid memories. *SAFARI Technical Report No. 2011-005*, September 2011.