การทำแคชให้เหมาะที่สุดสำหรับแฟลชทรานสเลชันเลเยอร์

นายพีระ ต้นธีรวงศ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2557

OPTIMIZING CACHE FOR FLASH TRANSLATION LAYER

Mr.Peera Thontirawong

A Dissertation Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2014

Thesis Title	OPTIMIZING CACHE FOR FLASH TRANSLATION LAYER
By	Mr.Peera Thontirawong
Field of Study	Computer Engineering
Thesis Advisor	Professor Prabhas Chongstitvatana, Ph.D.
Thesis Co-advisor	Assistant Professor Mongkol Ekpanyapong, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

..... Dean of the Faculty of Engineering
(Professor Bundhit Eua-arporn, Ph.D.)

THESIS COMMITTEE

..... Chairman
(Associate Professor Somchai Prasitjutrakul, Ph.D.)

..... Thesis Advisor
(Professor Prabhas Chongstitvatana, Ph.D.)

..... Thesis Co-advisor
(Assistant Professor Mongkol Ekpanyapong, Ph.D.)

..... Examiner
(Assistant Professor Setha Pan-ngum, Ph.D.)

..... Examiner
(Assistant Professor Thanarat Chalidabhongse, Ph.D.)

..... External Examiner
(Assistant Professor Kemathat Vibhatavanij, Ph.D.)

พีระ ต้นธีรวงศ์: การทำแคชให้เหมาะที่สุดสำหรับแฟลชทรานสเลชันเลเยอร์. (OPTIMIZING CACHE FOR FLASH TRANSLATION LAYER) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ศ. ดร. ประภาส จงสถิตย์วัฒนา, อ.ที่ปรึกษาวิทยานิพนธ์ร่วม : ผศ. ดร. มงคล เอกปัญญาพงศ์, 110 หน้า.

ภาควิชา ..วิศวกรรมคอมพิวเตอร์..	ลายมือชื่อนิสิต .....
สาขาวิชา ..วิศวกรรมคอมพิวเตอร์..	ลายมือชื่ออ.ที่ปรึกษาวิทยานิพนธ์หลัก .....
ปีการศึกษา .....2557.....	ลายมือชื่ออ.ที่ปรึกษาวิทยานิพนธ์ร่วม .....

5171821021: MAJOR COMPUTER ENGINEERING
KEYWORDS: FLASH MEMORY / FLASH TRANSLATION LAYER / ASYMMETRICAL ACCESS TIME / CACHE MEMORY / ADDRESS TRANSLATION / GARBAGE COLLECTION / CACHE REPLACEMENT POLICY / SPATIAL LOCALITY

PEERA THONTIRAWONG : OPTIMIZING CACHE FOR FLASH TRANSLATION LAYER. ADVISOR : PROF. PRABHAS CHONGSTITVATANA, Ph.D., CO-ADVISOR : ASST. PROF. MONGKOL EKPANYAPONG, Ph.D., 110 pp.

Flash memory is ubiquitous and it can be found in many devices. It surpasses hard disk in various ways. However, flash memory is not perfect; it contains several limitations owing to its characteristics. The most important limitation is that it is not suitable for in-place update. In order to overcome many limitations, a flash memory device employs a flash translation layer (FTL) to manage data locations. The main component in the FTL is a mapping table as it handles out-of-place update by performing address translation. The concept of the address translation is comparable to the virtual memory but the sizes of both address spaces managed by the FTL are identical. With limited spatial resource, cache is mandatory for fine-grained mapping table. Nevertheless, the characteristics of flash memory directly influence cache performance, for example, asymmetrical access times between read and write. The optimization is therefore required. In this dissertation, the novel cache techniques for FTL will be presented. The optimization will be focused on asymmetrical access times and the main goal will be address translation time while reducing spatial overhead.

Department : Computer Engineering	Student's Signature .....
Field of Study : Computer Engineering	Advisor's Signature .....
Academic Year : 2014	Co-advisor's Signature .....

Contents

Chapter Page

List of Tables

List of Figures

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

In computer systems, there are three main components: processor, memory, and I/O interface. Since the beginning of computer era, the number of transistors on a dense integrated circuit, e.g., processor, is double every two years according to Moore's law**?**, and the processor performance is expected to double every 18 months **?**.

On the contrary, the growth of hard disk drive (HDD), the ferromagnetic-materials-based memory, performance is outpaced by this growth and the gap is widening . One reason is the limitation of mechanical movement. However, the advent of flash memory has broken this performance lock. Flash memory is a non-volatile memory based on semiconductor technology; thus, its growth is applicable to Moore's law.

As flash memory outperforms ferromagnetic materials on shock resistance, power consumption, and, of course, access latency, it is more preferred for data storage of many computer systems. NAND flash memory is welcomely accepted in mobile computer systems and portable storage devices, for examples, smartphones, tablets, USB drives, and memory cards, owing to its small area and high durability. In personal computers (PCs) and enterprise servers, NAND flash memory was adopted in form of solid-state drive (SSD). SSDs is gradually replacing ferromagnetic hard disk drives (HDD). For examples, SSDs are selected as the solely secondary storage in the Macbook Air, the lightweight laptops, in 2010 and the Macbook Pro with retina display, the high performance laptops, in 2012 from Apple Inc,. Furthermore, Amazon, Google, and Microsoft began offering their cloud computing services with options of SSDs as back-end storages in 2014.

However, NAND flash memory also has it own technological limitations that make itself more complicated to utilize than the traditional storage, e.g., HDD. In-place update is infeasible for NAND flash memory due to the erase-before-program requirement. Out-of-place update has to be employed. Moreover, the lifetime of a NAND flash memory cell is decided by program/erase (P/E) cycles. Frequently erasing a particular part increases its bit error rate, and eventually the whole memory will become unusable.

Amid these problems, a software named flash translation layer (FTL) is implemented for the simplification. FTL handles the out-of-place update and emulates the traditional block-interfaces for a flash memory device. Other issues related to the management of the flash memory device are also in charge of FTL.

One of the main modules of FTL is address translation. Its function is to convert logical addresses, which are used by a file system, to physical addresses in the flash memory device. Since the conversion is mandatory, the address translation is directly influence the response time of the flash memory device. In other words, the flash memory device needs a well-suited FTL to exploit its potential performance.

There are many studies for enhancing an FTL, and one of the famous types of FTL is demand-based. The demand-based FTL **?** is well-known for its speed and little SRAM overhead, and the general idea behind every demand-based FTL is caching. In order to efficiently utilize flash memory, a fine-grain mapping table with a lot of information is required. The fine-grain mapping table allows the address translation to freely select any location for storing data. However, the fine-grain mapping table heavily demand large SRAM capacity. Instead of storing the large mapping table inside SRAM, a demand-based FTL opts for keeping the table in the flash memory device and caching only essential information in SRAM. Therefore, the fast response time and low SRAM overhead are achieved.

The performance of a demand-based FTL depends on its cache system. Exploiting spatial locality is very popular for increasing a demand-based FTL performance **????** since the spatial locality can be found in sequential accesses. With the advancement of NAND flash memory technology, the smallest accessible unit of NAND flash memory is very large; blindingly exploiting spatial locality may not increase the performance as much as expected. On the contrary, the performance might be decreased due to insufficient cache size.

Besides, the key of any caches is to hide the back-end storage latency. However, the service times of flash memory operations are asymmetrical. The programming time of a page is substantially longer than the reading time. In addition, the longest operation is erasure, which is operated on block basis. Therefore, applying typical cache technique to a demand-based FTL may not achieve sublime performance, the cache system of a

demand-based FTL is needed to be optimized.

In this dissertation, the asymmetrical access time of the flash memory will be put in the spotlight. Since the programming time is several times slower than reading time, the penalty of evicting a modified cache entry is significant. To be precise, its can be more than the cost of evicting an unmodified cache entry and then reading back. Consequently, not only the cache hit ratio, but the number of evicted modified cache entries also affects FTL performance.

Finally, we proposed two novel demand-based FTLs with optimized cache system. The optimization is mainly based on asymmetrical access time of NAND flash memory. The first FTL is named SCFTL. Its cache is optimized for exploiting spatial locality while minimizing cache write-back. The cooperation between the spatial locality exploitation techniques and the novel replacement policy estimates cost and benefit of cache entry fetching and takes action accordingly.

Then, 3DFTL, the second FTL, completely eliminates cache write-back by embedded a modified cache entry with the data that have to be written. The compression technique is also employed in order to embedded many mapping entries into each. In addition, the accelerator, ILog, is also proposed. ILog employs another small cache that help 3DFTL avoiding unnecessary flash memory accesses. Moreover, it also guarantees the recovery time of 3DFTL.

1.1 Design Goals for Optimizing Cache for Flash Translation Layer

Aside from exploiting asymmetrical access time, we first set our three goals that will be used to design and optimize the cache of mapping table in a demand-based FTL:

1. be efficient, the proposed FTL should have fast system response time.
2. be economical, the proposed FTL should not occupy large SRAM capacity.
3. be adaptable, the proposed FTL should be able to work with other FTL functions, e.g., garbage collection, wear leveling, and load balancing.

1.2 Scope

The scope of this dissertation is limited to the following:

- This dissertation is mainly focused on the address translation.
- This dissertation considers only address caching or the cache of mapping table. Data caching and instruction caching are not covered.
- The proposed FTLs was experimented and evaluated on the simulated 8GB MLC NAND flash memory, which is described in Chapter 4.
- The performance of the proposed FTLs was evaluated by the selected workload traces from SPC **?** and MSRC **?**.
- The FTLs that are the outcome of this dissertation are compared against DFTL **?** and CDFTL **?**.

1.3 Dissertation Organization

In this dissertation, the two novel FTLs and one additional technique would be described. This chapter has introduced the importance of FTL, the motivation of this dissertation, the design goals, and the scope of this dissertation. The rest of the dissertation is organized as follows. Chapter 2 will give more details of NAND flash memory and FTL. The characteristics, limitations, and constraints of NAND flash memory are also included. Then, the related works and the state-of-the-art FTLs are in Chapter 3. Chapter 4 will be the performance evaluation methodology. The performance metrics that are used by this dissertation, the information and assumptions of the simulator, and the descriptions of benchmarks are provided. This dissertation works are begun in Chapter 5. The design and performance of SCFTL, a demand-based FTL with efficient caching strategy, is presented. Next, Chapter 6 is the details about of 3DFTL. 3DFTL is a demand-based FTL with zero cache write-back. After that, the scalabilities of SCFTL and 3DFTL are discussed in Chapter 7. Then, ILog, a garbage collection and recovery accelerator for 3DFTL, is proposed in Chapter 8. Finally, Chapter 9 will conclude this dissertation. The contributions is provided and the directions of future works are briefly discussed.

CHAPTER II

BACKGROUND

2.1 NAND Flash Memory

Flash memory is a non-volatile memory. The technology behind the flash memory is derived from MOSFET. As shown in Figure 2.1, a cell of flash memory has an additional floating gate sit below a control gate. The floating gate can trap electrons and acts as a control gate; hence, the cell can be use as a data storage. NAND flash memory is one type of the flash memory that the cells are connected as arrays of NAND gates.

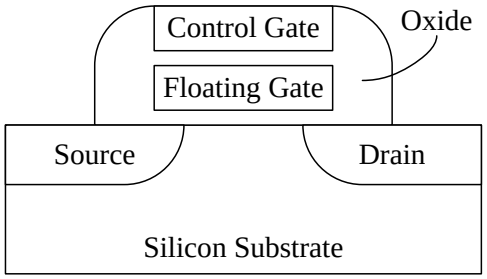


Figure 2.1: A flash memory cell

2.1.1 Principle Operations

The NAND flash memory has three basic operations: program, read, and erase. A page is the smallest unit for reading and programming. A group of consecutive pages called a block, and it is the smallest unit for erasure.

1. **Program** In order to program a flash cell, very strong voltage is applied to the control gate. The very strong voltage makes some electrons passing through layers of insulator. The passed electrons will be trapped inside the floating gate and therefore cause a similar effect as constantly applying current to the control gate.

2. **Read** The stored data can be read out by disconnecting the control gate and letting the floating gate controls the current. A cell that has electrons trapped inside its floating gate will let the current pass through, and vice versa.

3. **Erase** Finally, the trapped electrons can be removed by applying very strong voltage to the substrate while holding the control gate to ground. As a result, the electrons are pulling out of the floating gate, and the current has to be controlled by the control gate.

2.1.2 Organization

As illustrated in Figure 2.2, several arrays of cells are grouped together as a block owing to the physical organization of NAND flash memory. In the block, a group of cells that have the same position in arrays is called a page. Thus, the number of pages in a block is determined by the length of arrays. A page is the smallest unit for reading and programming. A group of consecutive pages called a block, and it is the smallest unit for erasure.

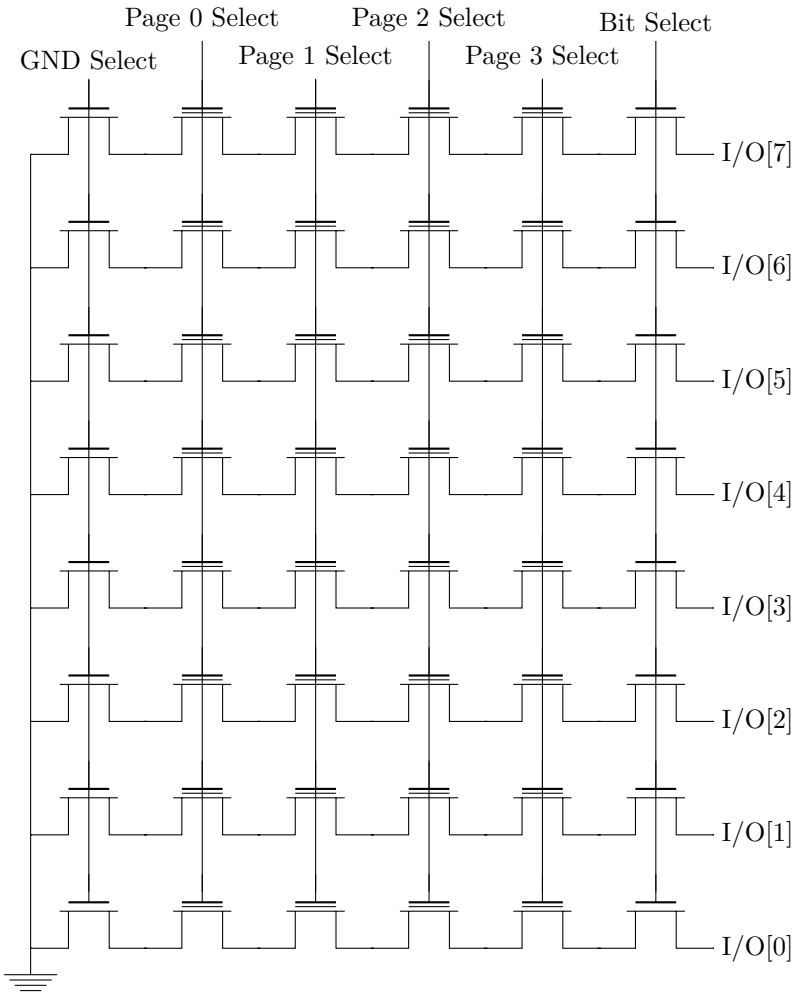


Figure 2.2: A block of four-page NAND flash memory

Additionally, as blocks are grouped together and form a plane. Each plane has buffer registers with total size of one page. Since reading or programming a page has to be done as a whole, but the whole page is too large to be simultaneously transferred via databus, the registers are served as a buffer for data transfer. In addition, each plane can be executed in parallel, i.e., planes of a NAND flash memory device are working together like RAID. Lastly, each die of NAND flash memory contains several planes; however, dies are sharing the interconnection. An illustration of NAND flash memory organization is shown in Figure 2.3.

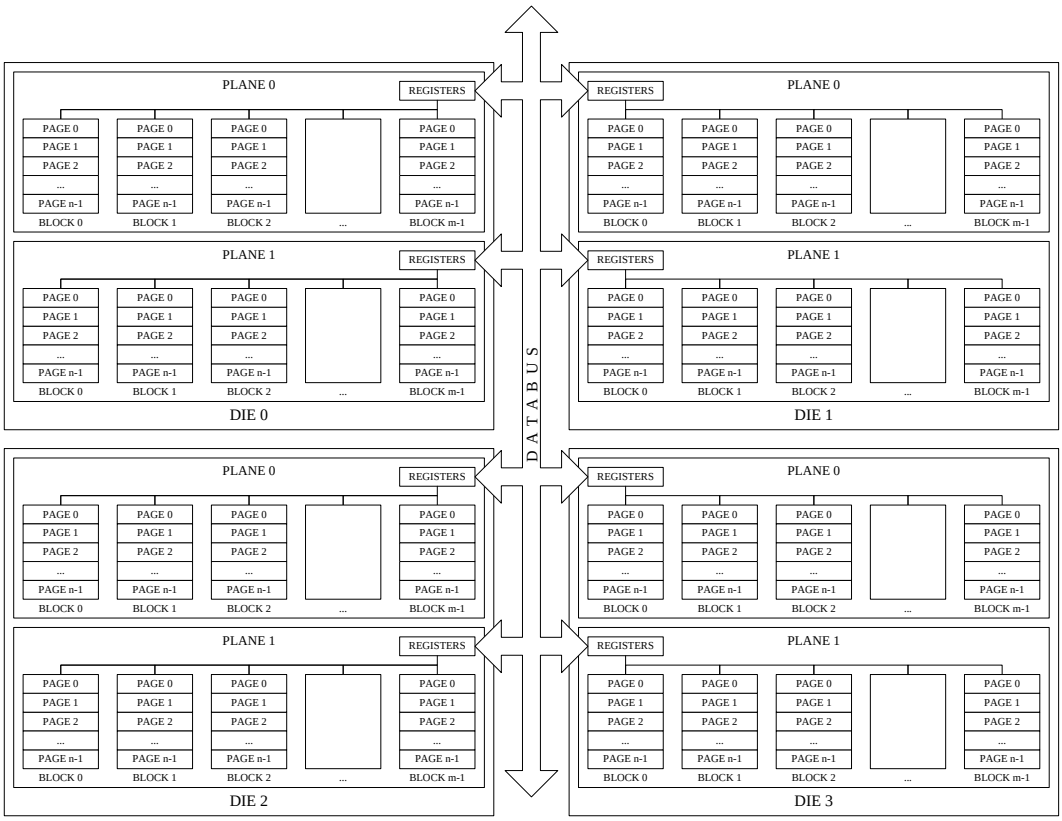


Figure 2.3: An orgainization of NAND flash memory

2.1.3 Characteristics

1. **No In-Place Update** There are two different operations needed for bringing electrons into and out of the floating gate. In addition, the smallest programming unit is a page, but the smallest erasure unit is a block; erasing one page will force the entire block to be erased. As adjacent pages may still have valid data, several operation will be incurred in order to prevent data loss. Consequently, in-place update is not feasible for NAND flash memory.

2. **Limit Number of Program/Erase (P/E) Cycles** Programming and erasure needs applying strong voltage to flash cells that may cause deformation. The error rate will be increasing after several program and erase operations, and the cells will be worn-out, eventually.

3. **Asymmetrical Read/Write Speed** Generally, Programming a page needs significantly longer time than reading due to its process, which may include verification and validation. Furthermore, a block erasure also typically takes longer time than a page programming.

2.1.4 Single-Level Cell (SLC) and Multi-Level Cell (MLC)

Due to the fact that a flash cell actually stores electrons, the stored data have to be interpreted from the level of voltage readout. Basically, only one level of voltage is used to classify the value; therefore, each cell value can be either ”1” or ”0” and represents one bit of data. This type of flash memory is called Single-Level Cell or SLC. Another type is Multi-Level Cell or MLC. Rather than using only one level, MLC uses at least two levels to classify the value. In case of two levels, the voltage can be fall into four thresholds. Hence, two bits of data can be stored. However, MLC is likely to have more errors than comparable SLC because of smaller thresholds. Consequently, the endurance of MLC is lower.

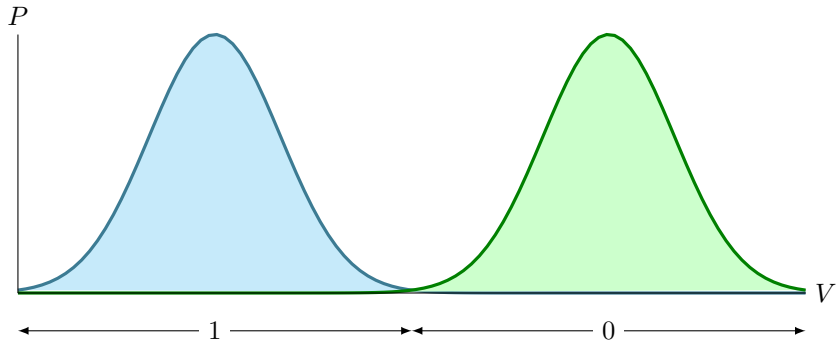


Figure 2.4: Probability distribution of SLC voltage

2.1.4.1 MLC Constraints

Due to the nature of NAND flash memory that cells are connected as an array and control gate of cells of the same page are sharing the wire, programming – or even reading

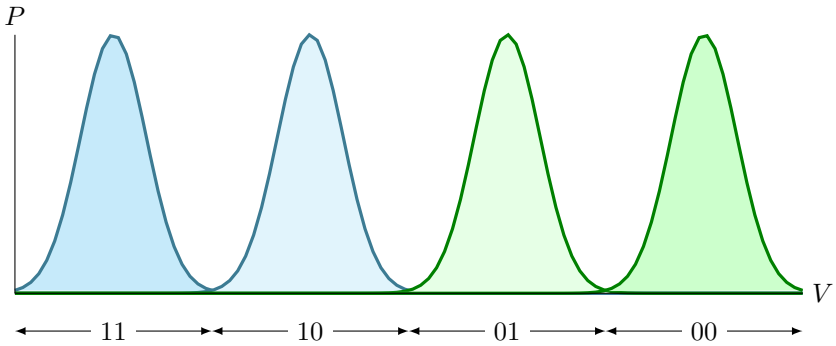


Figure 2.5: Probability distribution of MLC voltage

– a cell can disturb nearby cells. In MLC, every cell is very sensitive because of narrower thresholds; a small disturbance can easily cause an error. Programming an MLC page has to be more restrict. As a consequence, two more constraints have been added.

1. **No Partial Page Programming** Every page allows to be programmed only once. It has to be erased before it is reprogrammed.
2. **Sequential Page Programming** Every page in each block has to be sequentially programmed in order to minimize the effect of disturbance on nearby programmed pages.

2.2 Flash Translation Layer

Flash translation layer or FTL is software that manages the flash memory. It emulates the simpler I/O interfaces, i.e., read and write, and hides the complication of flash memory management from the upper level as shown in Figure 2.6. FTL handles read request or write request from the file system and issues read operation, program operation, or erase operation to NAND flash memory controller. Also, the logical block address (LBA), which is normally used by many file system for referring daat location, is translated to a physical address, e.g., physical page number (PBN) for reading or programming, or physical block number (PBN) for erasure.

FTL has as many as five main functions because of the characteristics of flash memory.

1. **Address Translation** Due to the limitation of flash memory, out-of-place update is

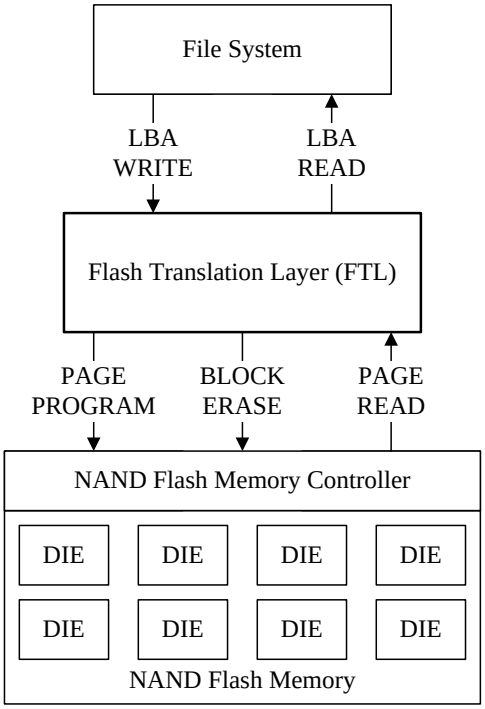


Figure 2.6: An overview of FTL

necessary. Consequently, the actual address or physical address of data is different from its request address or logical address, and translation between these logical address and physical address is needed. Generally, address translations are achieved by looking up and updating the mapping tables according to requests.

2. **Garbage Collection** When the freshly updated data is stored in a new location, the obsoleted pages are not automatically erased from the flash memory. Since out-of-place updates keep consuming free pages, a garbage collection is required for erasing and recycling the outdated pages. As the erase is done on block level, The function of garbage collection also include moving valid pages to new places before claiming the block.

3. **Wear Leveling** The error of each flash cell is higher when its number of P/E cycles increased. As the name implied, wear leveling avoids overuse or underuse any particular cell by cycling data around the flash memory. Ideally, the objective of wear leveling is making every cell worn-out at the same time and therefore yielding the longest possible flash memory lifetime.

4. **Parallelization and Load Balancing** A flash device usually contains several flash memory dies. These dies can handle operations concurrently; hence, parallelization

and load balancing is necessary for maximizing the performance.

5. **Worn-Out Block Management** Since each flash cell varies in endurance, some cells may worn-out easily than the others. Moreover, some cells might be already worn-out even though it was just out-of-the-box. As the worn-out cell has higher error rate, it has to be handle separately.

The request handling flowcharts of a typical FTL are shown in Figure 2.7 and 2.8. In the beginning, the I/O interface puts the request in command queue (CMD Q) while keeps the data in the DRAM buffer. When FTL is available, it will retrieve the request from the command queue and process accordingly. the requested logical block address (LBA) and the requested size (SIZE) is therefore converted to several consecutive logical page numbers (LPNs). Then, the LPNs are mapped to PPNs by address translation. During the mapping, a free physical page number (PPN) will be allocate in case of write request. After the access to PPN was done, the related metadata are updated, and then the I/O interface responses to the requester.

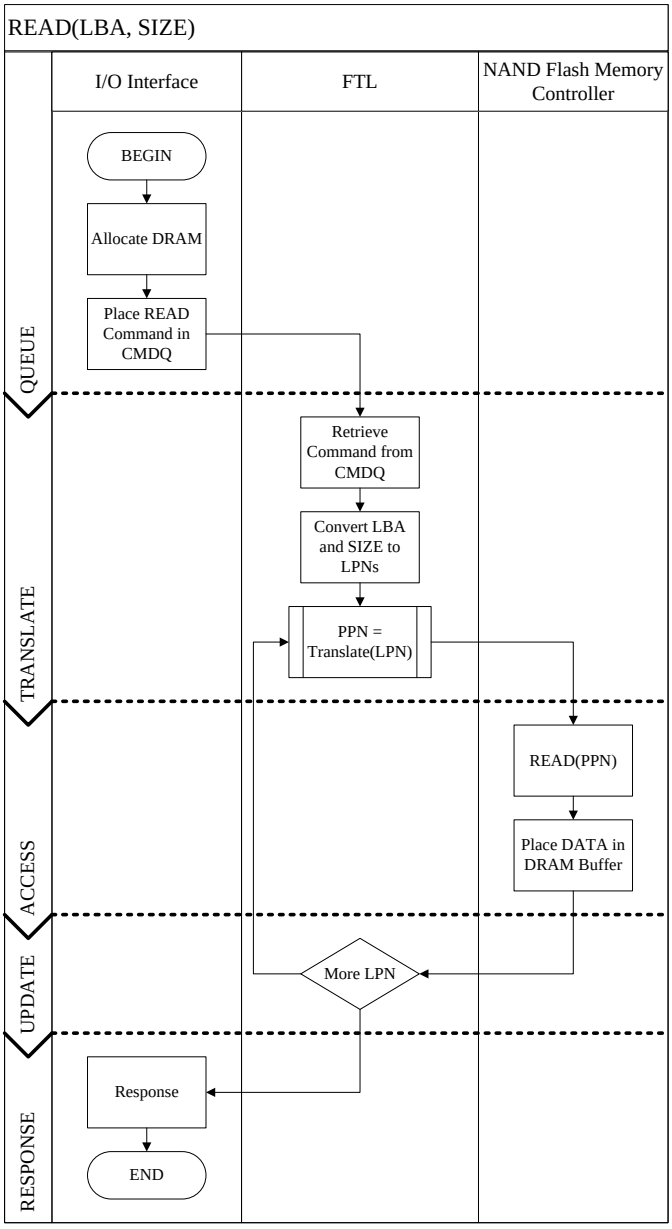


Figure 2.7: A flowchart of read request handling

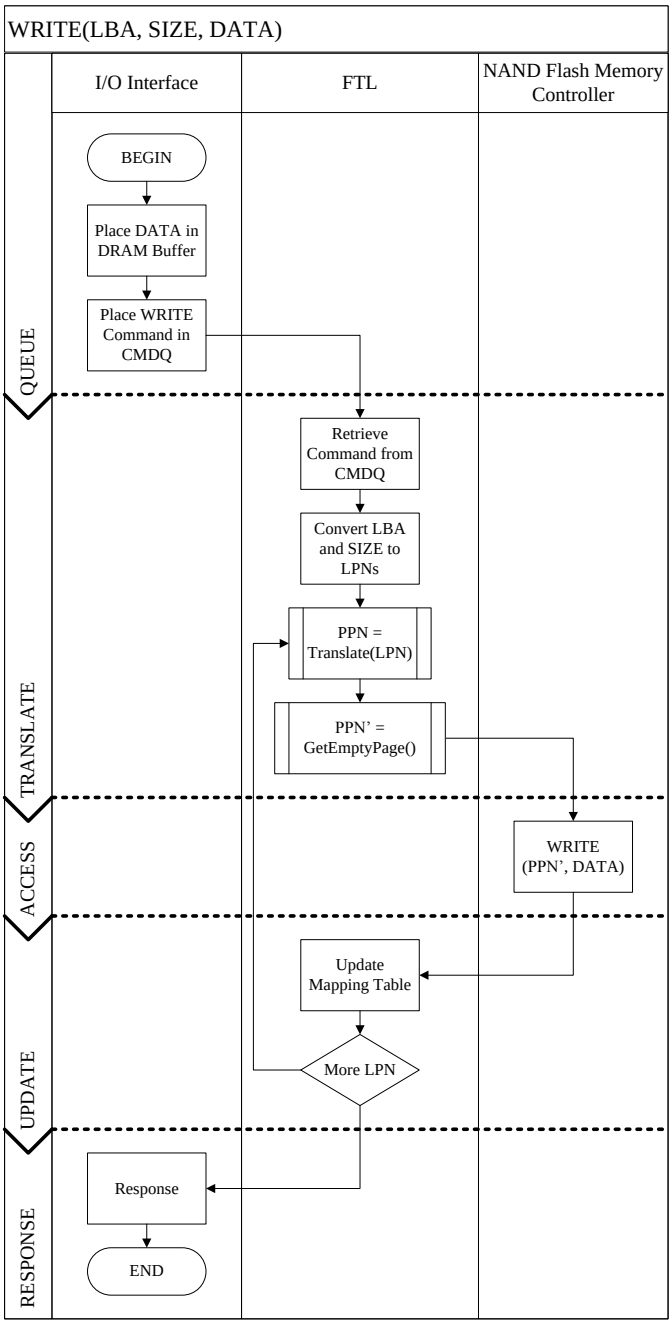


Figure 2.8: A flowchart of write request handling

CHAPTER III

LITERATURE REVIEWS

Owing to the characteristic of the flash memory, updates of data have to be done out-of-place. However, an FTL hides the actual location of data from the upper-level device to reduce the complication. Hence, an address translation is fundamental, as it is used for locating the data by mapping the logical address to the physical address.

3.1 Block-Level FTL and Page-Level FTL

There are two basic ways to map addresses. One is at the page level and the other is at the block level. Assuming a logical page has the same size as a flash page. The page-level address translation **?** will maintain a mapping relationship between pages. In other words, an FTL with the page-level address translation one-to-one map a logical page to a physical page.

On the other hand, block-level address translation **?** keeps the memory requirement low by employing block-to-block mapping. It can assign a logical block to a free physical block while leaving their least significant bits of addresses or page numbers untouched. In order to maintain one-to-one mapping property, not only the updated page is going to be written in to a new block but the valid pages in the same block also have to be copied in to a new block. In consequence, block-level address translation suffers from numerous valid page copied, which is also called high write amplification.

The tangible benefit of the page-level address translation is its flexibility. Its mapping constrains are more relax than the block-level address translation, and therefore high performance can be achieved easily. However, the trade off is the spatial requirements of the mapping table. The mapping table of a page-level address translation is several magnitudes larger than the mapping table of a block-level address translation.

To be compromised, hybrids of these two address translations are invented. Hybrid address translation combine page-level address translation and block-level address trans-

lation together. Even though, hybrid FTLs consist the page-level address translation, the block-level address translation limits the mapping flexibility degree.

3.2 Log Buffer-Based FTL

The log buffer-based scheme is the widely adopted scheme for hybrid FTLs **??????????**. The log-buffer-based divides flash blocks into data blocks and log blocks. The data blocks are using block-level mapping, while log blocks, which are fewer, are mapped at the page-level. The log buffer-based scheme stores the updated pages in log blocks. The copying of valid pages in the block that contains obsoleted pages is delayed until log blocks are full. This process of reclaiming log blocks is called merge operation.

There are three types of merge operation: switch merge, partial merge, and full merge. They are illustrated in Figure 3.1. A switch merge has the lowest operation cost. All pages in a log block are sorted. The log block can be converted to a data block instantaneously. In case of a partial merge, every written page in a log block has to be sorted, but the log block is not fully written. Hence, only unmodified pages have to be copied to the log block before it can be converted to a data block. In other cases, a log block has to be full merge that requires allocating new blocks and copying several pages.

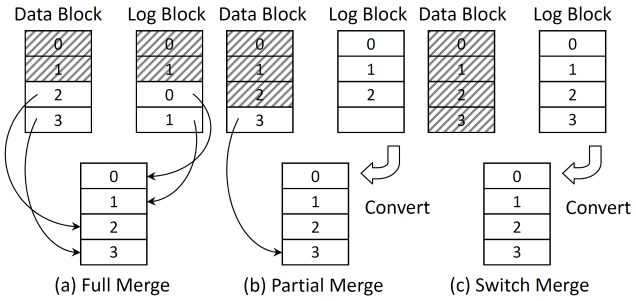


Figure 3.1: Three types of merge operation **?**

The first log buffer-based scheme is BAST **?**. The limitation of BAST is that pages in each log block are tied to only pages from the same data block. FAST **??** elevated this limitation by allowing mapping of a log page to any data page. However, FAST suffers from lengthy processing time of a full merge operation because many data blocks are involved in each log block.

Accordingly, several FTLs aim for reducing the number of data blocks associated

with a log block. SAST **?** groups consecutive data blocks into sets. Every set has it own log blocks, and its management is similar to FAST. A-SAST **?** is an improve SAST that a number of data blocks in each set can be adjusted on the fly. Another approach, which is very simple, is KAST **?**. KAST is FAST with the limited number of associated data block.

There is also an FTL that combines BAST and FAST together. LAST **?** classifies write requests into two types: sequential write requests and random write requests. The sequential write requests have to be specially taken care in order to increase the possibility of switch merge and partial merge operations. They are handled by BAST managed log blocks, while FAST managed log blocks deals with the random write requests.

An extreme log buffer-based FTL is Superblock-based FTL **??**. A part of its idea is rather similar to SAST. Blocks are grouped and named a superblock. However, superblock does not have data blocks; every block is treated as a log block. Since log blocks are managed at the page-level, the mapping table this scheme is huge. It has to be divided into hierarchy, and some are offloaded to the flash memory.

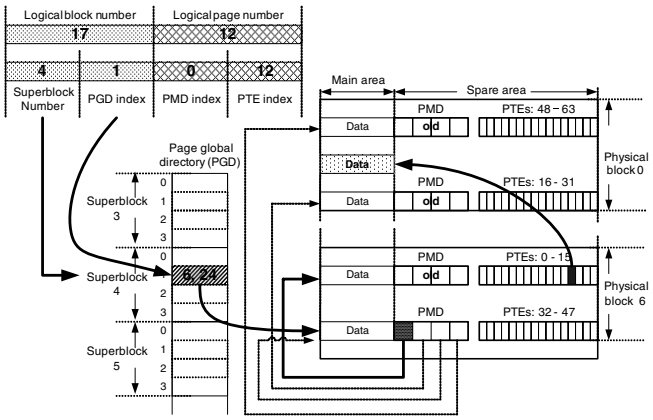


Figure 3.2: An example of address translation in Superblock-based FTL **?**

Although log buffer-based FTLs provide good trade-off between performance and memory space, merge operations, which reclaim disorder log blocks, cause high P/E cycles. Therefore, they are not suitable for MLC NAND flash memory, which has low endurance. MNFTL **?** is an FTL designed for MLC NAND flash memory. MNFTL has two-step address translation. The first step is looking for the index of the corresponding page mapping table (PMT) in RAM. The second step reads the PMT from spare area, and

then maps the logical page to the physical page according to the PMT entry. MNFTL employs concentrate-mapping technique to reduce write amplification caused by erase operation. It uses the block mapping table (BMT) to conduct out-of-place update within the same block as the replaced one. However, the BMT also limits the number of logical blocks that can be mapped to a physical block. Therefore, MNFTL is also a hybrid FTL.

3.3 Demand-Based FTL

Though the page-level FTL has substantial advantages, its overwhelmingly large mapping table makes it infeasible to be implemented. The flexibility of page-level address translation obeys in sequential programming order constraint of MLC NAND flash memory easily. Moreover, page-level FTL allows higher capacity utilization, which leads to low P/E cycles. Consequently, several page-level FTLs were invented **???????**.

In order to implement the page-level address translation in limited RAM space, DFTL **?** stores the enormous page-mapping table in several pages of the flash memory. The pages are called translation page and can be located by a small mapping table in RAM. Since retrieving and updating a mapping entry need to access the flash memory, keeping the mapping table in the flash memory drastically burdens the performance. For these reasons, DFTL exploits the temporal locality by caching some mapping entries in RAM. Hence, the number of flash page read operations decreases and translation page updating can be postponed until a modified mapping entry is evicted. Furthermore, the postponed updating allows the modified mapping entries of the same translation page to combine and write together when one of them is evicted. However, the hit ratio of DFTL is not high because it takes little advantage of the spatial locality.

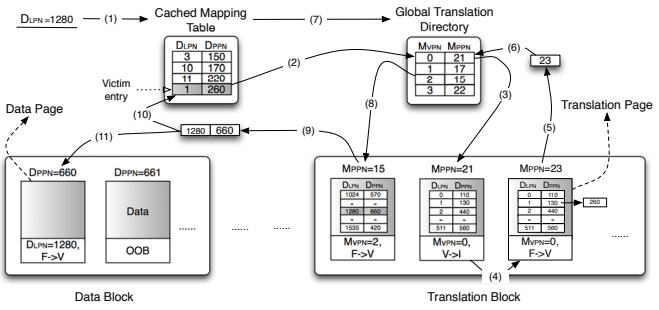


Figure 3.3: An example of address translation in DFTL **?**

In order to increase the hit ratio, CAST **?** adds a consecutive field into their

caches. Consecutive logical addresses that are mapped to consecutive physical addresses are grouped into single cache block. The example of consecutive field is illustrated in Figure 3.4. As the physical addresses of the logical addresses 10-12 are consecutive, they can be kept together in the same cache block by setting the consecutive field (C) to two.

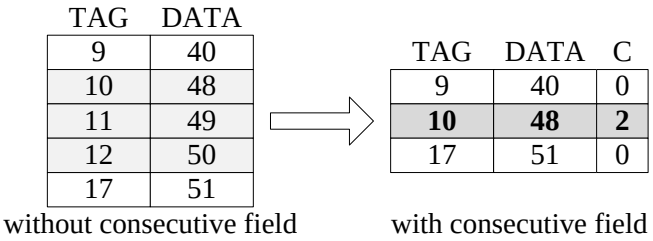


Figure 3.4: An example of cache with consecutive field.

Owing to the nature of sequential write operations, their physical addresses are more likely to be consecutive. Hence, adding the consecutive field improves overall performance. In addition, CAST biases its physical address selection to prefer contiguous data locations and therefore increases the chance of consecutive addresses.

An idea of consecutive field is also presented in a hybrid FTL named CFTL **?**. CFTL employs LFU cache replacement policy; frequently accessed pages are having better performance. To accelerate the address translation time of infrequently accessed pages, CFTL implements a mechanism to detect and convert infrequently accessed page-mapping entries into block-mapping entries.

To further exploit the spatial locality, S-FTL **?** caches a whole translation page as a single cache block. It also reduces the RAM space needed by compressing cached translation pages. In contrast, CDFTL **?** enhances DFTL by adding a second level cache. The first level cache is similar to DFTL while the second level cache stores entire translation pages. Hence, the temporal locality is exploited on the first level cache, while the spatial locality is handled by the second level cache. Even though caching full translation pages can guarantee the spatial locality exploitation, it is not suitable for a device with small RAM capacity because each cached translation page is huge.

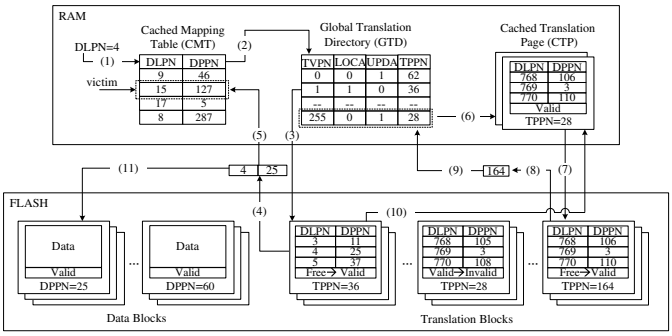


Figure 3.5: An example of address translation in CDFTL **?**

CHAPTER IV

PERFORMANCE EVALUATION METHODOLOGY

Before detailing this dissertation works, we would like to describe about the performance evaluation methodology that will be used throughout this dissertation. This chapter begins with the performance metrics and follows by the assumptions and constraints of the simulation. In the end of the chapter, the characteristics of the benchmarks is described.

4.1 Performance Metrics

Although the main objective of this dissertation is to optimize the cache for a demand-based FTL, directly measuring the performance of a cache may not reflect the actual FTL performance. For example, a demand-based FTL that has very high cache miss ratio but very low cache miss penalty may perform better than a demand-based FTL that has very low cache miss ratio but very high cache miss penalty. Consequently, the evaluation will measure not only the cache performance but also the FTL performance.

4.1.1 Cache Performance

Generally, the performance of a cache system is measured by the hit ratio or miss ratio. The cache hit ratio is the ratio of cache hit per cache access:

$$CacheHitRatio = \frac{num_{CacheHit}}{num_{CacheAccess}}. \tag{4.1}$$

The cache miss ratio is, on the other hand, the ratio between the number of cache misses and the number of cache accesses:

$$CacheMissRatio = \frac{num_{CacheMiss}}{num_{CacheAccess}}. \tag{4.2}$$

The hit or miss ratio can perfectly show the efficiency of cache system; A cache system that can effectively exploit the access localities of workloads will have higher

cache hit ratio or lower cache miss ratio. However, the overall performance, especially in terms of time, may not be able to predict by only cache hit or miss ratio. The reason is that the penalty of each cache miss can be differed. There are several research works that proposed cache performance metrics that are more accurate **?**.

Since this dissertation are focusing on the performance of FTL, the complex metrics are not suitable for analyzing the effect of cache over FTL. Hence, the cache hit or miss ratio will be used. Moreover, a cache miss will be categorized by it miss penalty. Thus, there are four classes of cache access:

1. cache hit, A cache hit is a read or write access while its data are being cached;
2. cache miss without penalty, A cache miss without penalty is an access that does not hit and require neither fetching nor writing-back. A cache write that replaces unmodified cache entry, for example;
3. cache miss with fetching penalty, A cache miss with fetching penalty is an access that does not hit; thus, the data have to be fetched from the back-end storage but will not replace a modified data;
4. cache miss with writing-back penalty, A cache miss with writing-back penalty is an access that does not hit and evicts a modified cache entry; therefore, the modified data have to be written back to the back-end storage.

Among these four, a cache hit has the best performance then a cache miss without penalty. In NAND flash memory, a page reading time is only a tenths of a page programing time. Therefore, a cache miss with write-back penalty is considered for having the worst performance out of four classes.

4.1.2 FTL Performance

Since an FTL has many functions, as mentioned in Chapter 2, its performance can be measured in several aspects. There are five main categories of FTL performance **?**: SRAM overhead, translation performance, block utilization, garbage collection performance, and recovery performance.

4.1.2.1 SRAM Overhead

Generally, the data of FTL, for instance the mapping table, are kept in SRAM, which is a valuable resource. Thus, several FTLs also use the flash memory itself to store FTL data and keep only essential data in SRAM. Making the most out of this valuable resource is desirable. Since a SRAM overhead of a demand-based FTL can be configured by adjusting its cache size, the SRAM overhead has to be controlled in each experiment.

4.1.2.2 Translation Performance

One of the principle functions of FTL is address translation that is responsible for finding data location and also assigning new data location. The translation performance is determined by how many look-ups or updates required and how long do these operations taken since they may access the flash memory. Therefore, the translation performance mostly depends on the choice of algorithm, data structure, and storage.

The translation performance can be measured by average system response time and average access time. The system response time is the total time from the very beginning til finished, while the access time is begin when the FTL starts servicing the I/O request. Therefore, the difference is the I/O queuing time. The average system response time is the better metric for measuring the overall performance, as it include the impact of request frequency.

Since the average system response time is workload dependent, the value can be much varied between benchmarks. In this dissertation, the translation performance is measured by a normalized average system response time. The normalized average system response time (NT) is calculated from dividing the measured average system response time of evaluating FTL (t_{FTL}) by the average system response time of the original page-level FTL (t_{PFTL}), which keeps the entire page-level mapping table in SRAM:

$$NT = \frac{t_{FTL}}{t_{PFTL}}. \tag{4.3}$$

Therefore, the normalized average system response time is comparable between benchmarks. The value lower than 1 means an improvement from the baseline; hence, the lower value is the better translation performance.

4.1.2.3 Block Utilization

Block utilization is a number of pages in a block that was used before the block is erased. The block utilization of a flash memory can be also measured by a number of block erasures because an FTL with lower block utilization usually runs out of free pages and triggers it's garbage collection more often. Consequently, the FTL that has better block utilization tends to have longer lifespan. The block utilization (BU) can be calculated from:

$$BU = \frac{w_{IO}}{e}, \tag{4.4}$$

where w_{IO} is the number of pages that was requested to be written by I/O and e is the number of pages erased, which is the number of block erased multiply by the number of pages in block.

4.1.2.4 Garbage Collection Performance

The performance of garbage collection is determined by the number of valid pages moved during each garbage collection. The lower number often results in higher performance because of fewer pages written and blocks erased. The garbage collection performance is defined by valid pages move rate (MR):

$$MR = \frac{m}{e}, \tag{4.5}$$

where m is the number of valid pages moved by the garbage collection and e is the number of pages erased. The higher value of MR means the lower garbage collection performance.

4.1.2.5 Fault Tolerance

Flash memory is widely used in embedded and mobile systems. A sudden power loss, which is unexpected, can be occurred anytime. Since the crucial FTL data is usually stored in SRAM, mechanisms, such as backup and recovery, for handling data loss have to be taken in consideration for these systems in order to prevent data corrupted. As the recovery involves reading many pages in order to reconstruct the latest state of mapping table, the recovery performance (RP) can be measured by the number of pages read per

total number of pages in the recovering flash memory:

$$RP = \frac{r_{recovery}}{N}, \tag{4.6}$$

where $r_{recovery}$ is the number of pages read during recovery and N is the total number of pages in the flash memory.

4.2 Simulator

Every experimental result in this dissertation was simulated by the customized simulator based on FlashSim **?**. FlashSim is widely adopted by many research works . The customized simulator is consists of four main components: I/O interface, FTL, NAND flash memory controller, and NAND flash memory, as shown in Figure 4.1.

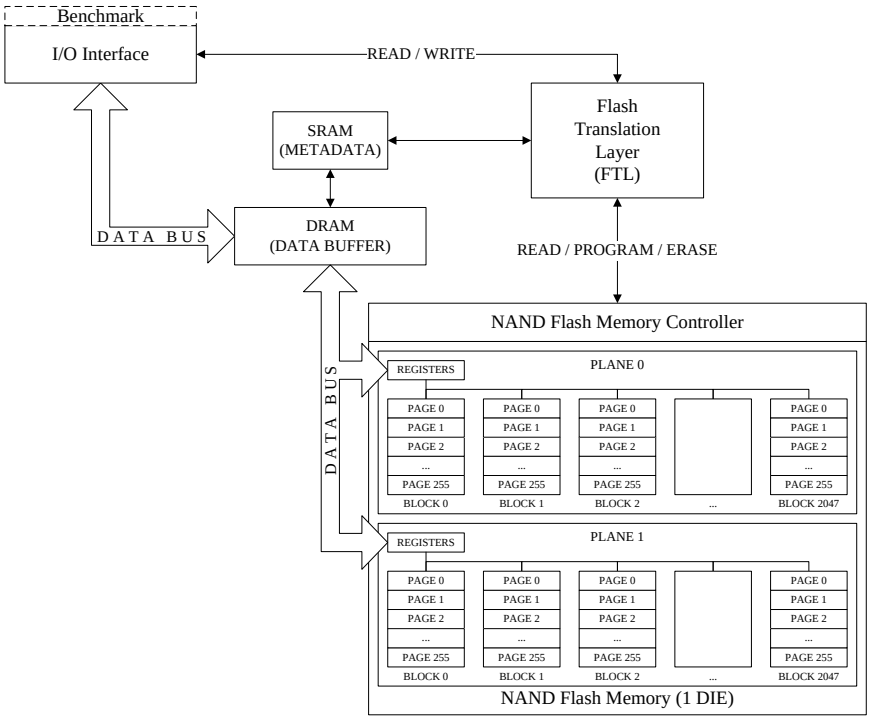


Figure 4.1: An overview of the simulator

1. I/O interface

   The I/ O interface is assumed as a black-box. It generates I/O requests according to the I/O trace. The generated request will be immediately put on the command queue, while data is put in the allocated space in the DRAM buffer.

2. FTL

The FTL response to queuing I/O requests by converting a request, read or write, into several commands, read, program, or erase. Then, the command is issued to the NAND flash memory controller. Also, it has SRAM for temporarily storing necessary metadata, for instance, mapping table, status of pages and blocks.

3. NAND flash memory controller

   The NAND flash memory Controller accepts the command from the FTL. Then, it accesses the NAND flash memory according to the command and records the calculated execution time. The data can be transferred directly from and to DRAM buffer; however, the metadata are passed to the FTL.

4. NAND flash memory

   The simulated NAND flash memory is modeled according to the 8GB MLC NAND flash memory specifications obtained from Micron Technology, Inc **?**. The summary of specifications is provided in Table 4.1. The register in each plane acts as a reading and programing buffer since the page has to be read or programed as a whole.

Table 4.1: 8GB MLC NAND Flash Memory Specifications **?**

Specifications	
Die Size	8GB (4,096 blocks)
Die Configuration	2 planes (2,048 + 2,048 blocks)
Plane Configuration	2,048 blocks with 1 8,640-byte register
Block Configuration	256 pages
Page Size	8,192 + 448 bytes (data + spare area)
Page Read Time	75µs
Page Program Time	1,300µs
Block Erase Time	3,800µs
Transfer Rate	50MB/s
Endurance	3,000 P/E cycles
Minimum ECC Requirement	24-bit ECC per 1,080 bytes

4.2.1 Assumptions and Constraints

Since this dissertation is concentrated on applying cache for address translation performance and SRAM overhead, several assumptions have been made and some variables have been controlled in order to exhibit the impact of the cache on FTL performance.

The assumptions and constraints are as follows.

- There are only two types of I/O request: READ and WRITE. Special types of request, for example TRIM , are not allowed.
- The command queue is a simple FIFO queue. Reordering or aggregating requests are not allowed.
- The sizes of command queue and DRAM buffer are infinite.
- There is no delay for I/O interface. However, the queuing time depends on availability of the FTL and NAND flash memory controller.
- The garbage collection is only running on-demand, and the triggering condition is the same for every FTL.
- The garbage collection selects a recycling block by greedy selection based on the number of valid pages in the block.
- Wear leveling is disable.
- No parallelization and load-balancing.
- Any special features of the NAND flash memory, e.g., copy-back, two-plane access, is disable.
- Only one die of the NAND flash memory is simulated.
- the over-provisioning ratio or the ratio of reserved capacity to total capacity is set to 1/32. the user accessible is 31/32 of total capacity.
- the user accessible pages are 100% filled before starting the simulation.

4.3 Benchmarks

The performance of FTLs will be evaluated by executing several workload traces selected from two publicly accessible sources: Storage Performance Council (SPC) **?** and Microsoft Research Cambridge (MSRC) **?**.

4.3.1 SPC Benchmarks

The SPC benchmarks consists of five I/O traces. Two are the I/O traces of OLTP applications and three traces are obtained from search engines. The summarized statistics of benchmarks are in Table 4.2.

Table 4.2: Summarized Statistic of SPC Benchmarks

Benchmark	Number of Requests	Write Request (%)	Average Size (KB)	Average Read Size (KB)	Average Write Size (KB)
Financial1	5,334,984	76.84	3.38	2.24	3.72
Financial2	3,699,195	17.65	2.39	2.28	2.92
WebSearch1	227,623	0.05	17.16	17.17	8.00
WebSearch2	964,410	0.05	17.26	17.26	8.00
WebSearch1	899,121	0.07	17.78	17.76	44.63

The request frequency histograms of each benchmark are shown in Figure **??**.

4.3.2 MSRC Benchmarks

For MSRC benchmarks, the traces from the enterprise data centers running various applications are selected. The details of these traces can be found on their publication **?**.

MSRC benchmarks **?** are I/O traces collected from 13 servers in Microsoft data center running various enterprise workloads, as shown in Table 4.3, for seven days. Each server has one OS volume and several data volumes. The total is 36 volumes. However, only 25 volumes, 13 OS volumes and 12 data volumes, are selected as benchmarks. The summarized statistics of the selected traces are described in Table 4.4.

As shown in Table 4.4, all of OS volumes and all of their first data volumes are selected, except `ts` and `wdev`. `ts` does not have any data volumes because it is a terminal server. On the contrary, $\texttt{wdev}_2$ is selected instead of $\texttt{wdev}_1$ since $\texttt{wdev}_1$ has very few I/O requests.

The request frequency histograms of each benchmark are shown in Figure **??**.

Table 4.3: Description of MSRC servers

Server	Description
hm	Hardware monitoring
mds	Media server
prn	Print server
proj	Project directories
prxy	Firewall/web proxy
rsrch	Research projects
src1	Source control
src2	Source control
stg	Web staging
ts	Terminal server
usr	User home directories
wdev	Test web server
web	Web/SQL server

Table 4.4: Summarized Statistic of MSRC Benchmarks

Benchmark	Number of Requests	Write Request (%)	Average Size (KB)	Average Read Size (KB)	Average Write Size (KB)
OS Volumes					
hm_0	2,610,531	67.07	8.75	6.79	9.72
mds_0	1,191,594	88.43	8.80	22.59	7.00
prn_0	2,474,978	83.02	8.98	21.89	6.34
$proj_0$	2,805,838	85.40	30.34	14.80	33.00
$prxy_0$	11,822,230	97.79	4.43	6.43	4.38
$rsrch_0$	1,051,948	96.59	8.30	11.40	8.19
$src1_0$	3,348,043	21.78	9.68	5.01	26.45
$src2_0$	1,429,295	90.33	6.56	6.66	6.54
stg_0	1,887,372	86.41	11.06	25.09	8.86
ts_0	1,633,021	85.37	8.94	14.72	7.95
usr_0	1,622,780	56.62	25.66	45.82	10.20
$wdev_0$	1,022,921	87.11	7.58	7.18	7.64
web_0	1,486,663	78.13	12.28	28.42	7.77
Data Volumes					
hm_1	609,192	4.65	15.16	14.93	19.99
mds_1	166,627	48.61	25.42	37.71	12.43
prn_1	2,296,084	39.80	7.25	6.54	8.32
$proj_1$	238,890	18.56	36.61	42.72	9.77
$prxy_1$	43,563,576	62.59	9.16	12.28	7.31
$rsrch_1$	9,303	99.94	8.13	4.00	8.13
$src1_1$	3,723,809	35.13	14.95	12.76	18.99
$src2_1$	209,728	4.77	55.99	58.20	11.86
stg_1	688,922	91.11	11.29	41.90	8.30
usr_1	112,405	18.05	38.08	45.12	6.10
$wdev_2$	140,988	99.93	9.32	4.19	9.33
web_1	105,324	59.20	21.18	36.79	10.43

CHAPTER V

SCFTL: AN EFFICIENT CACHING STRATEGY FOR FLASH TRANSLATION LAYER

Naturally, the efficiency of any cache system depends on workloads. A cache system takes advantages of its faster access time and workload localities, e.g., temporal locality and spatial locality, by buffering data that are likely to be accessed in the near future. However, the smaller capacity of the cache system cannot hold every data; thus, managing cached data can significantly affect the performance.

As already mentioned in Chapter 3, the core of a demand-based FTL is the cache of the page-level mapping table and its performance mostly relies on its cache efficiency. Since the size of a file can be larger than a flash page and many file systems prefer allocating contiguous logical pages for storing a large file, accessing it usually has spatial locality. Thus, many researchers proposed several demand-based FTLs that exploit the spatial locality ????.

The demand of larger capacity storage is inevitable. Accordingly, a representative from Micron Technology, Inc. presented that the size of a flash page tends to grow larger ?. As the page-level mapping table of demand-based FTL is stored in flash pages, a larger page can contain more mapping entries. Although the cache efficiency might be increased by retrieving many mapping entries in single page read, caching all of them also needs larger cache, i.e. SRAM, capacity.

For example, suppose that each page contains 2048 mapping entries, and there are 2048 sequential reads from logical address 0 to 2047 to the flash memory. In this example, the performance of two cache scheme will be compared. The first scheme caches 2048 mapping entries per translation page read while the second scheme caches only 64 entries per read. Since 2048 mapping entries can be read at once, caching all 2048 entries will result in only one cache miss or cache miss ratio of 1/2048. Consequently, only 2049 flash pages was read: one page for 2048 mapping entries and 2048 pages for data.

On the other hand, the second scheme will yield the miss ratio of 1/64 or 32 cache misses in total. Therefore, 2080 pages are read. As one can see, the number of reads is only 31 pages or about 1.51% more than the first scheme while the difference of spatial requirements is as high as 32 times.

Although caching many mapping entries in each translation page read might lower cache misses from sequential accesses, a very long sequential accesses is in fact infrequent. As published by **?**, the median file size is remain 4KB in 2009 even though the average file size is increasing. Accordingly, S-FTL and CDFTL, which cache entire translation pages, might gain very little benefit from larger page size. On the contrary, they might not be able to maintain the same level of performance because the lower number of cache entries means the higher possibility of evicting a modified cache entry. Additionally, the larger page size means that fewer pages are needed to store a large file; hence, the number of sequential accesses may decrease. As a result, the consecutive field technique, which is implemented in CFTL and CAST, will be less effective.

Given these points, a high performance demand-based FTL requires an appropriate technique for exploiting spatial locality of mapping table in large-page flash memory. Consequently, SCFTL, a demand-based FTL with novel cache management techniques, will be proposed in this chapter.

5.1 Design of SCFTL

SCFTL is a page-level address translation FTL that employs an efficient caching strategy. It consists of three main components: page-mapping table (PMT), translation page directory (TPD), and cache mapping table (CMT). In order to achieve the page-level address translation, SCFTL stores a page-mapping table in several translation pages (TPs). Each translation page keeps a group of physical page numbers (PPNs) mapped to consecutive logical addresses or logical page numbers (LPNs). Due to the gigantic flash page size, each translation page holds thousands of physical page numbers; hence, only few pages are needed for the complete page-mapping table. TPD keeps the addresses of every translation page in RAM and indexes them by the most significant bits of logical addresses. The performance degradation from offloading the mapping table is reduced by caching several mapping entries in CMT. Furthermore, CMT integrates two spatial locality exploitation techniques and a customized cache replacement policy in order to

enhance its efficiency.

5.1.1 Two-Level Address Translation

As the page-mapping table of SCFTL is kept inside the flash memory, the address translation has to be done by a two-level process. Generally, the physical address of a request could be found in CMT; hence, the two-level address translation is not triggered. However, the two-level address translation will be executed in case of a cache miss.

The two-level address translation splits a logical address into two parts: an index and an offset. In the first level, TPD converts the index into the location of the related translation page, and then the translation page is retrieved from the flash memory. After that, the second level uses the offset, which is a position of the physical address in the translation page, to extract the physical address from the translation page. Therefore, the logical address is finally translated to the corresponding physical address. An example of the two-level address translation is provided in step (2)-(4) of Figure 5.1.

5.1.2 Efficient Caching Strategy

Owing to the localities of PMT accesses, several translation page read operations can be omitted by caching mapped physical page numbers. However, the efficiency of CMT does not only depend on the temporal locality; it is also highly influenced by the spatial locality since the spatial locality can be easily found in sequential accesses, e.g., accessing large file.

As the page size grows larger, more mapping entries can be kept inside each page. This means a demand-based FTL can fetch contiguous mapping entries faster, but storing all of them will also take a large amount of SRAM capacity. Without larger SRAM, demand-based FTLs that exploits the spatial locality by caching entire translation pages, for instance, S-FTL and CDFTL, will run out of SRAM capacity very quickly. Furthermore, a decrease in the number of cache entries means a high probability of evicting a modified cache entry. Due to the fact that a page programing takes much longer time to finish than a page reading, frequently evicting modified cache entries is undesirable.

In this section, a using of fundamental technique for exploiting spatial locality is described. The consecutive field technique is also added for better spatial efficiency.

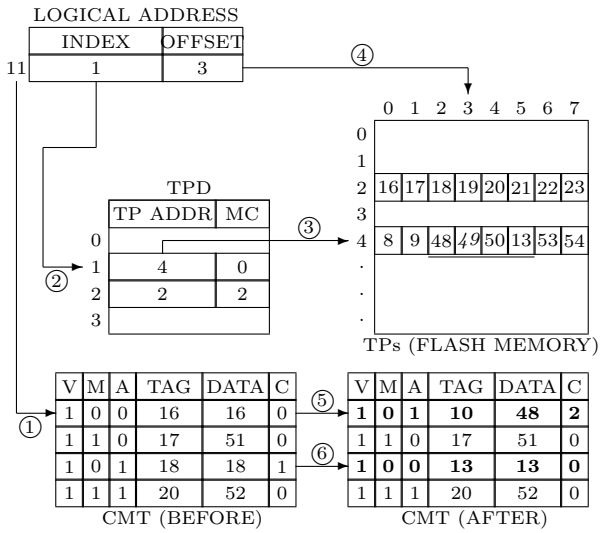


Figure 5.1: An example of the SCFTL address translation. Suppose a logical address of the request is 11, and each translation page contains eight physical addresses; the index and offset of the logical address is 1 and 3, respectively. (1) The access of the logical address 11 incurs a cache miss in CMT, and the first cache entry is selected as a victim. Writing back does not occur, as the victim is not modified. Then, (2) the two-level address translation is begun, and the translation page 1 is located at the page number 4. (3) The translation page is read from the flash memory, and (4) the physical address of the request is found at the offset 3. (5) Instead of storing only one mapping entry in CMT, the consecutive physical addresses of the logical addresses 10 and 12 are fetched and stored together with the logical address 11. (6) Assume that the spatial size is four; another mapping entry 13 will be fetched to CMT. Since this is a spatial fetching, the third cache entry is selected as a victim instead of the second cache entry, which has MC value lower than the threshold.

While both techniques is maintaining numerous choices of cache entry for replacement, a quintessential but simple replacement policy that prioritized replacing unmodified cache entries over modified cache entries diligently prevents ineffective translation page programing.

5.1.2.1 Spatial Locality Exploitation

As a flash page, which is the smallest reading or writing unit, can pack thousands of mapping entries, caching multiple mapping entries each translation page retrieving is convenient. However, caching an mapping entry that will not be accessed is wasting the cache space. In order to avoid caching unused mapping entries, a fine-grained spatial fetching technique is introduced.

Despite increasing the cache entry size to accommodate more mapping entries, SCFTL spends several small cache entries to exploit the spatial locality. In addition, as

the physical addresses of sequential write operations are likely to be sequentially assigned, facilitating the consecutive field could save the CMT capacity by combining several sequentially mapped entries into one single cache entry. The consecutive field technique was proposed by **?** and its details were described in Chapter 3.

As a result, the chance of cache trashing can be controlled by limiting the amount of mapping entries cached in each translation page read. Since SCFTL treats each mapping entry as an individual cache entry, a low demanded mapping entry can be independently replaced without disturbing others. However, not caching an entire translation page forces SCFTL to reacquire the translation page before writing back.

Another drawback is come from the consecutive field technique. Cache eviction can occur when modifying a cache entry even though it was a cache hit because the PPN of their consecutive mapping entries are no longer consecutive. Henceforth, the cache entry has to be split. The four cases of cache entry update are shown in Figure 5.2. The first case is a normal update. The value of consecutive field of the cache entry is zero; only one mapping entry is contained in this cache entry. Updating the mapping entry is straightforward. The second case and the third case are happened when the value of the consecutive field is one or more. In these case, each cache entry contains more than one mapping entries. The second case is when the head mapping entry is being modified, while the third case is the modifying of the tail mapping entry. The PPN of the modifying mapping entry will no longer be consecutive with the others; therefore, another cache entry is needed. Lastly, the fourth case is modifying a middle mapping entry of the consecutive mapping entries in a cache entry. Obviously, this situation will break a cache entry into three cache entries for the head mapping entries, the modifying mapping entry, and the tail mapping entries. However, the split cache entries can be merged back if their mapping entries are subjected to sequential write operations as shown in Figure 5.2b. As SCFTL sequentially program flash pages, only the LPN of the previous programed PPN will be evaluated in order to merge a freshly modified mapping entry.

Finally, as CMT allows a modified state of a mapping entry, the update of a translation page can be postponed. Consequently, several modifications of mapping entries from the same translation page can be combined and written together to minimize the number of page programing operations, and the effective way to update translation pages

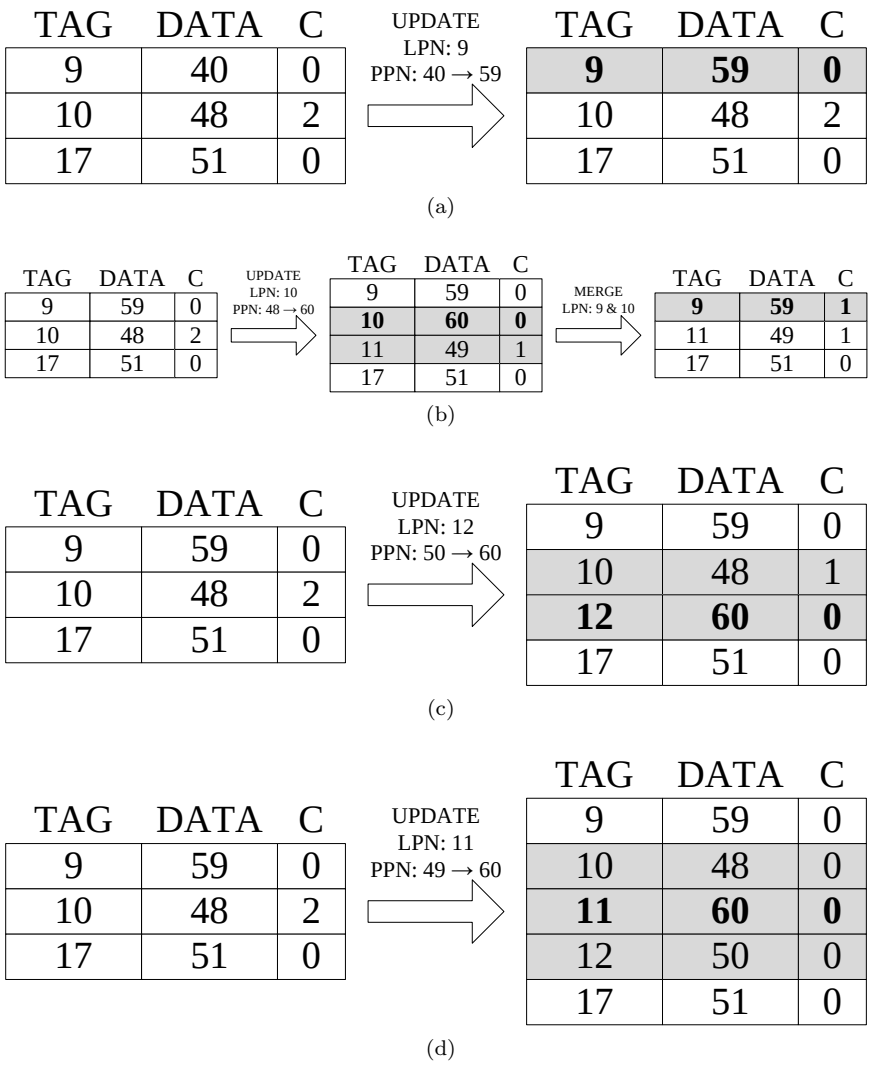


Figure 5.2: Four cases of CMT update in SCFTL: (a) normal update, (b) lower bound update (c), upper bound update, and (d) middle update.

is introduced in the next subsection.

5.1.2.2 Cache Replacement Policy

In order to decrease the number of translation page write operations, the victim selection process has to discriminate modified cache entries from others. However, preventing modified cache entries from being a victim in the fully associative cache may cause inefficient cache capacity utilization. Hence, SCFTL implements a customized cache replacement policy named D-NRU.

D-NRU is very similar to NRU **?**. Each cache entry contains a 1-bit flag for in-

dicating that it was recently accessed. Besides, D-NRU also takes a modified flag into account when it selects a victim. As the modified mapping entries from the same translation page can be written back simultaneously, writing the translation page that contains many modified mapping entries is more economical. Consequently, D-NRU considers the number of modified mapping entries in each translation page when selects a victim. The counters (MCs) are attached to TPD as shown in Figure 5.1. Each MC is very tiny, as it only needs to count until its value reaches the worthwhile threshold.

D-NRU is a combination of two variants of NRU algorithms. The algorithm selection is based on the type of a mapping entry fetching: normal fetching or spatial fetching. The normal fetching has high priority, as it is caused by an I/O request. Its victim selection prefers a cache entry that is not recently accessed ($\neg A$), unmodified ($\neg M$), and modified (M) with high MC value (MC_{TP}), respectively. On the other hand, the spatial fetching is initiated by spatial locality exploitation. As its mapping entry may not be reference, the cost of bringing it into the cache should be low. A modified cache entry that has the MC value lower than the threshold ($MC_{TP} < c$) will not become the victim of spatial fetching. Furthermore, the recently accessed flag is not set for the cache entry that is brought in by spatial fetching. The orders of D-NRU victim selection are provided in Table 5.1, and examples are illustrated in step (1) and (6) of Figure 5.1.

Table 5.1: Victim Selection Orders of D-NRU

Attributes	Normal Fetching	Spatial Fetching
$\neg A \wedge \neg M$	1	1
$\neg A \wedge M \wedge (MC_{TP} \geq c)$	2	2
$\neg A \wedge M \wedge (MC_{TP} < c)$	3	-
$A \wedge \neg M$	4	3
$A \wedge M \wedge (MC_{TP} \geq c)$	5	4
$A \wedge M \wedge (MC_{TP} < c)$	6	-

5.2 Performance Evaluation

In this section, the performance of SCFTL is evaluated and compared against DFTL and CDFTL. The experiments is done on the simulated 8GB MLC NAND flash memory as described in Chapter 4.

In these experiments, every cache is fully associative, and each FTL allows about

20KB of SRAM to be occupied by the cache. This amount of allowance is on par with the memory footprint required by a block-mapping table. As CDFTL prefers the second level cache to be large, the two-level cache of CDFTL is configured to 2KB (CMT) and 16KB (CTP), respectively.

The cache of SCFTL is manage by D-NRU replacement policy with 3-bit MC for each TP. The maximum value of MC is therefore 7. The consecutive field in each cache entry is 5 bits, and the number of fetches in each access is limited to 64.

The memory requirements are detailed in Table 5.2. According to Table 4.1, the total number of pages is 4096×256. As each page can store 8,192 bytes, 2,048 of 4-byte PPN can be contained. Therefore, only 512 translation pages, which are about 0.05% of the total pages, are required for SCFTL, DFTL, and CDFTL.

Since SCFTL keeps TPD and CMT in SRAM, the amount of SRAM needed is the summation of the requirements of these two components. Each TPD entry contains a 4-byte translation page number, a 3-bit modified counter, and a modified flag; hence, only 2.25KB of SRAM is needed for storing 512 TPD entries. On the other hand, each CMT entry consists of a 4-byte tag, a 4-byte PPN, a 5-bit consecutive field, and 3-bit for valid, modified, and recently accessed flags; therefore, each CMT entry is 9 bytes. With 2,048 CMT entries, the total size of CMT is 18KB. As a result, SCFTL requires only 20.25KB of SRAM.

Alternatively, DFTL and CDFTL require 18.50KB and 20.20KB of SRAM, respectively. Furthermore, the memory requirements of the original block-level FTL (BFTL), the original page-level FTL (PFTL), and the log-buffer-based FTL (FAST) with 3% log blocks are also provided in Table 5.3 for reference purpose.

5.2.1 Cache Performance

In Figure 5.6, SCFTL achieves the average cache miss of 20.68%. Due to the very small cache size configuration, the average cache miss of DFTL that does not exploit spatial locality is very high. Its average cache miss is 64.93%, which is significantly worse than SCFTL by 44.25%. In addition, the average cache miss of CDFTL is 20.46%.

Even though the average miss of CDFTL is very close to the average cache miss

Table 5.2: The configurations of FTLs for Chapter 5 experiments.

	DFTL	CDFTL	SCFTL
PMT	512 pages	512 pages	512 pages
GTD	2.00KB	2.19KB	2.25KB
CMT	16.50KB (2048 entries)	2.06KB (256 entries)	18.00KB (2048 entries)
CTP	-	16.00KB (2 * 2,048 entries)	-
GC	128KB	128KB	128KB
Total (without GC)	18.50KB	20.25KB	20.25KB
Total (with GC)	146.50KB	148.20KB	148.25KB

Table 5.3: The configurations of other FTLs.

FTL Type	BFTL Block-Level	FAST Log-Buffered-Based (Hybrid)	PFTL Page-Level
BMT	16KB	16KB	-
PMT	-	256KB	4096KB
GC	0.50KB	4.50KB	128KB
Total (without GC)	16.00KB	272.00KB	4096.00KB
Total (with GC)	16.50KB	276.50KB	4224.00KB

of SCFTL, the cache miss penalties of SCFTL are significantly lower because SCFTL is aware of the access time difference between reading and programing. Since the penalty of evicting a modified cache entry is significantly higher than evicting an unmodified one, the victim selection of SCFTL is biased by D-NRU. Subsequently, the cache miss with writing-back penalty, which is one or more flash programing operations, or the number of modified cache entries evicted is only 0.40% for SCFTL, while it is 1.05% and 5.36% for DFTL and CDFTL, respectively. Furthermore, the cache miss with fetching penalty, only one flash page read penalty, is 3.62% for SCFTL and 4.95% for CDFTL.

Additionally, the large portion of SCFTL cache misses does not have penalty, as they are cache misses caused by the garbage collection. While the garbage collection is relocating valid pages, the valid pages has to be read. Since the old PPN of the valid page is already known, the address translation may not required; however, a cache entry is still needed to be allocated for the updated PPN. Thus, the cache is accessed and may

need writing-back, but fetching the miss entry is unnecessary.

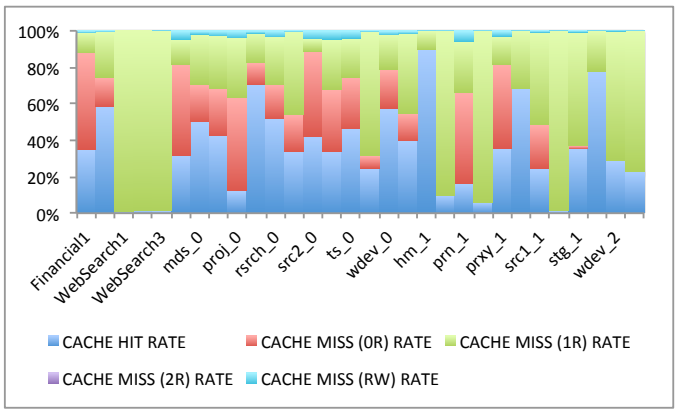


Figure 5.3: Cache performance of 20KB + 128KB DFTL

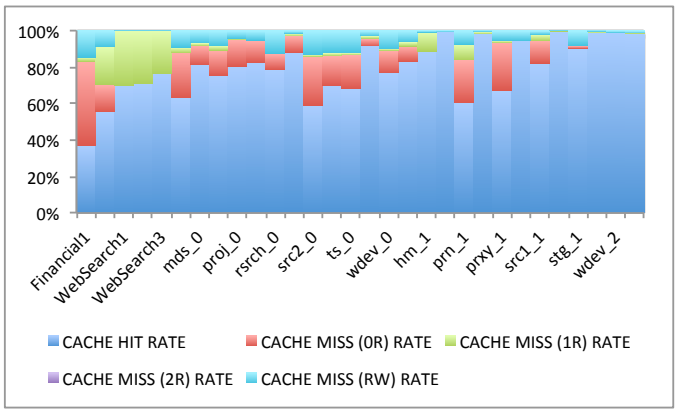


Figure 5.4: Cache performance of 20KB + 128KB CDFTL

5.2.2 Translation Performance

In this section, the translation performance, which is the most important metric, is measured. The translation performance (TP), which is calculated from the average system response time by (4.3), on each benchmarks are shown in Figure 5.7, 5.8, and 5.9, respectively. Then, the average values are compared in Figure 5.10. Since the shown values were derived from an average system response time, the lower value is better. Furthermore, the value of 1 means the average system response time is equal to the baseline, PFTL.

Due to the exceptional cache performance of SCFTL, its geometric mean of normalized average system response time is only 1.08. On the contrary, the value of DFTL and CDFTL is worse. Their value are 1.61 and 1.28, respectively. Hence, the speedups

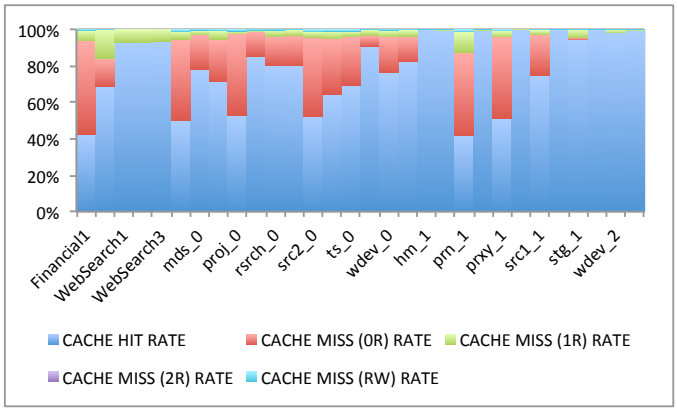


Figure 5.5: Cache performance of 20KB + 128KB SCFTL

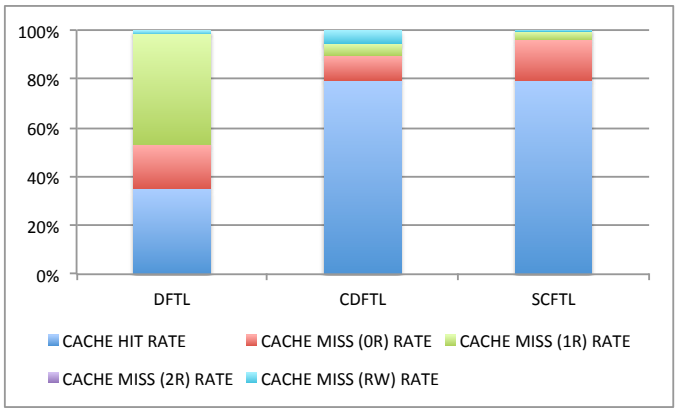


Figure 5.6: Cache performance of 20KB + 128KB FTLs

of SCFTL from DFTL and CDFTL are 1.50 and 1.19, respectively.

As shown in Figure 5.9, the average system response times of DFTL in `mds_1` and `proj_1` benchmarks is very high; the normalized value are 14.44 and 80.88, respectively. These two benchmarks numerous high frequency sequential accesses, according to Section 4.3.2. Since DFTL does not exploit spatial locality, an exceeding long servicing time due to consecutive cache misses is unavoidable. Consequently, the queuing time is accumulated. On the other hand, the performance of CDFTL is worse in low spatial locality benchmark because of the high ratio of cache miss with writing-back penalty.

SCFTL on the other hand can still excel on the benchmarks that do not have much spatial locality although it exploits the spatial locality. The reason is the small cache entry technique and D-NRU replacement policy that prevent SCFTL from frequently evicting modified cache entry.

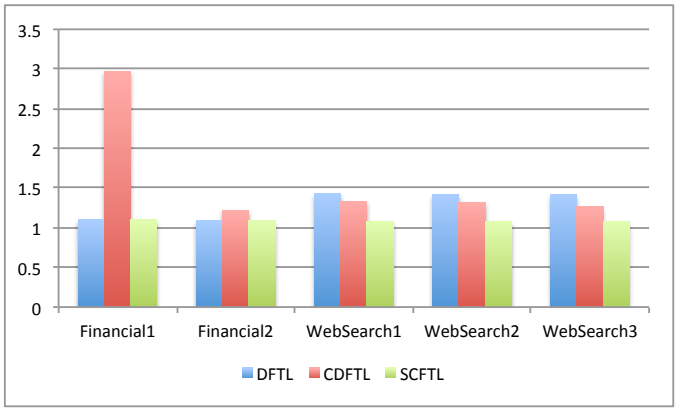


Figure 5.7: Normalized average system response time of 20KB + 128KB FTLs on SPC benchmarks

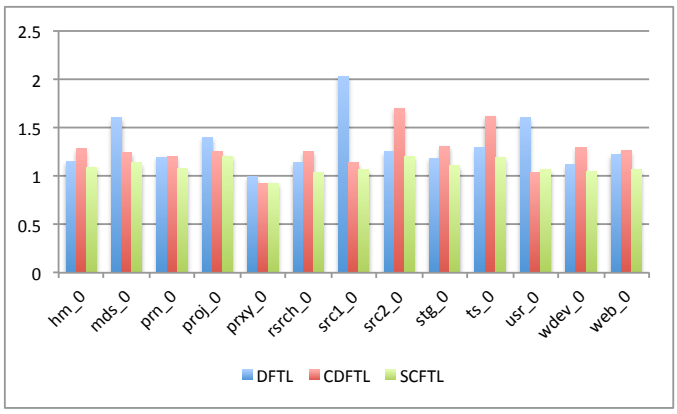


Figure 5.8: Normalized average system response time of 20KB + 128KB FTLs on MSRC benchmarks (OS volumes)

5.2.3 Block Utilization

Since the demand-based FTL periodically writes the translation page back to the flash memory, the number of translation page written-back affect the block utilization. For SCFTL, the average block utilization is 0.71, which is almost identical with the baseline, as shown in Figure 5.11. It is significantly higher than 0.61 of CDFTL that has high ratio of cache miss with flash programing ($Miss(RW)$).

5.2.4 Garbage Collection Performance

Since all of the FTLs in the experiments are page-level FTLs and they employ greedy selection for garbage collection, their garbage collection performances are very similar. The comparison of valid page move rate is shown in Figure 5.12.

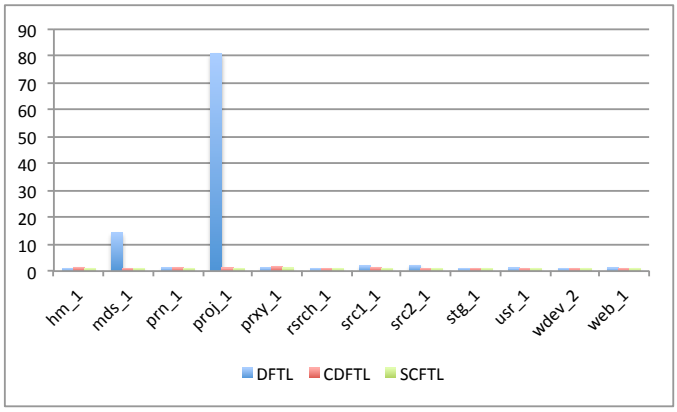


Figure 5.9: Normalized average system response time of 20KB + 128KB FTLs on MSRC benchmarks (data volumes)

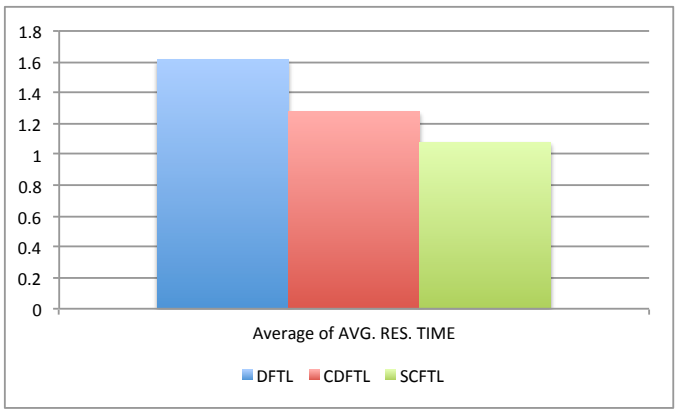


Figure 5.10: Normalized average system response time of 20KB + 128KB FTLs

5.3 Analysis of Cache Performance

In this section, the analysis of cache performance is separated into two parts: D-NRU replacement policy and spatial locality exploitation. The first part will compare D-NRU against traditional replacement policy LRU and NRU. Then, the second part will shown the efficiency of spatial locality technique that employed by SCFTL.

5.3.1 D-NRU Replacement Policy

The performances of D-NRU replacement policy and other replacement policies are shown in Figure 5.13. According to the design, D-NRU avoids cache trashing by not setting a recently accessed flag for spatial fetching, which results in subtle lower cache miss ratio. In addition, it prevents spatial fetching from replacing low beneficial modified cache entries. D-NRU also prevents premature cache writing back, which in turn provides

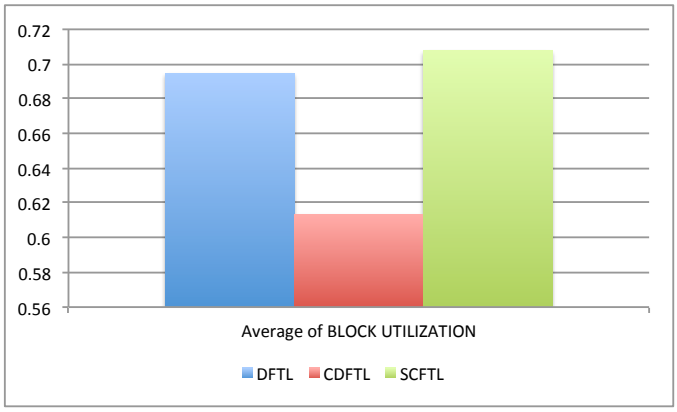


Figure 5.11: Block utilization of 20KB + 128KB FTLs

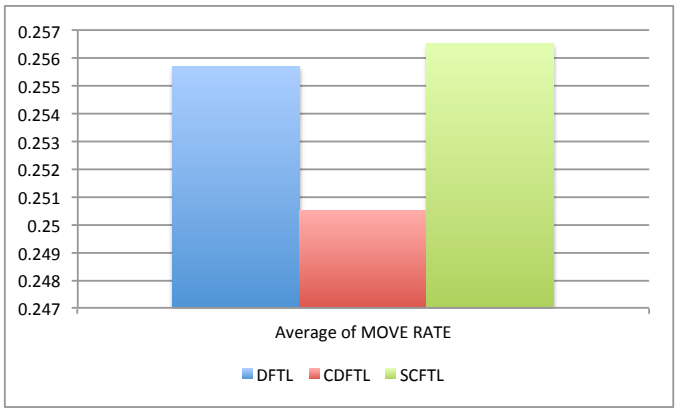


Figure 5.12: Valid page move rate of 20KB + 128KB FTLs

additional time to gather more modified mapping entries that can be written-back together to the same translation page.

Since the spatial locality exploitation of SCFTL is done by several small cache entries, it is possible to evict multiple cache entries in the same cache access. LRU and NRU do not aware of this fact and treat each small cache entry fetch equally; therefore, evicting cache entries originated from more than one modified translation pages in each cache access can easily happened as shown in Figure 5.14. On the other hand, D-NRU makes use of the modified counter (MC) of each translation page. It prefers evicting more modified cache entries from the same translation page to evicting cache entries from several translation pages. Consequently, the number of translation pages written per cache access significantly decreases. However, over protecting, which means too few victim candidates, will also heighten the risk of cache trashing and eventually will result in higher miss ratio.

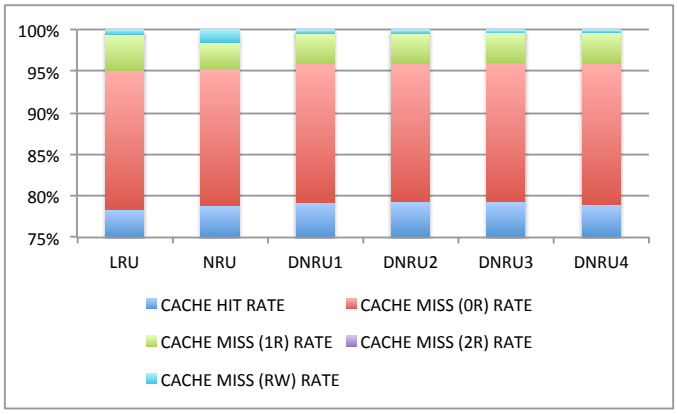


Figure 5.13: Cache performance of SCFTL with LRU, NRU, and D-NRU replacement policies

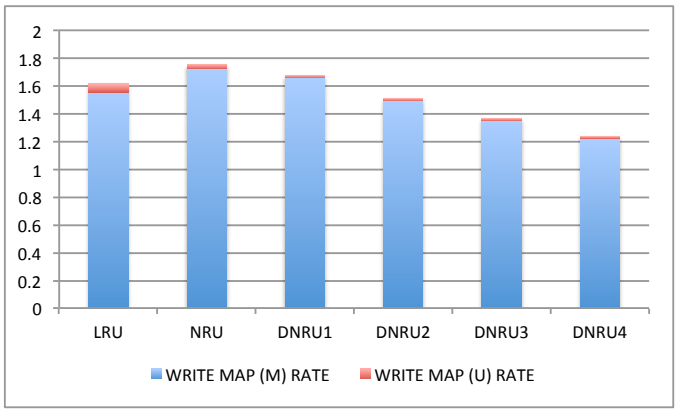


Figure 5.14: Number of modified translation page written per cache access of SCFTL with LRU, NRU, and D-NRU replacement policies

5.3.2 Spatial Locality Exploitation Techniques

As already mentioned in the beginning of this chapter, exploiting spatial locality by employing large cache entry gains little benefit. In contrast, the large cache entry means fewer number of cache entries per capacity; hence, the cache miss may increase. More importantly, it also increases the number of translation page written because of fewer victim candidates. The effect is shown in Figure 5.15.

Consequently, SCFTL employs the small cache entry with consecutive field and fetching multiple entries instead. The cache performance of each technique is shown in Figure 5.16 and Figure 5.17. Although both techniques do not reach the same level of cache miss ratio as large cache entry technique, they have significantly lower cache miss penalty. The advantage of small cache entry and consecutive field techniques is that they

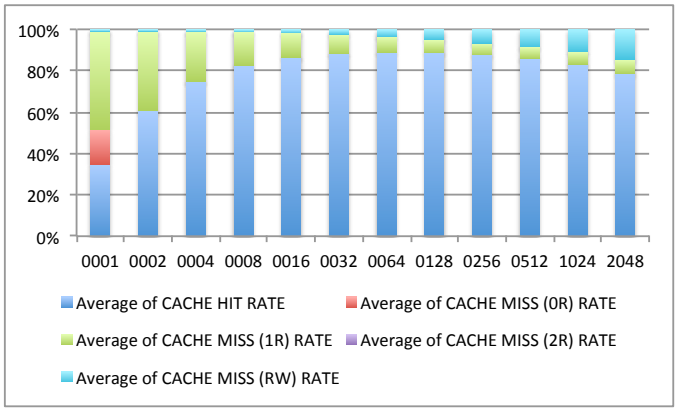


Figure 5.15: Cache performance of SCFTL with varied size of large cache entry technique

allow the garbage collection to update the PPN of a moved page to the cache without reading a translation page in case of cache miss. The impact of this advantage is shown as the cache miss without penalty (MISS (0R)) in the figures. This cannot be happened for the large cache entry because an cache entry for the PPN of a moved page has to be completely filled; hence, the PPNs of adjacent LPNs has to be fetched from the translation page.

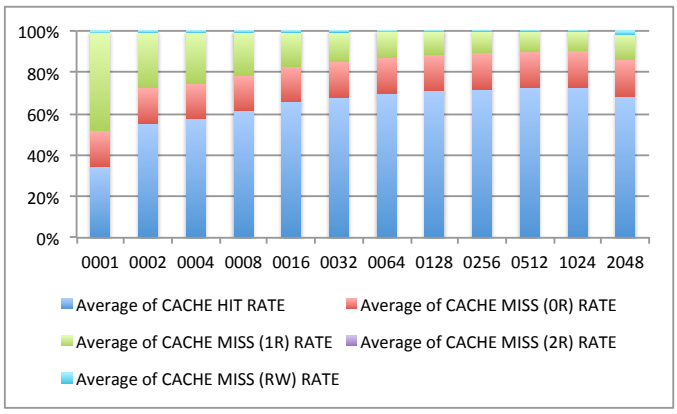


Figure 5.16: Cache performance of SCFTL with varied number of small cache entry technique

The consecutive field technique is indeed very spatially efficient for exploiting spatial locality, but it cannot be scaled because of the low possibility of large consecutive field. In contrast, the small cache entry technique is not spatially efficient as the tag to data ratio of each cache entry is very high. Nevertheless, combining these two together yield a very efficient technique as they complement each other. The consecutive field technique lowers the capacity demand while the small cache entry technique keeps fetching more entries. The cache performance of the combined small cache entry and 5-bit consecutive

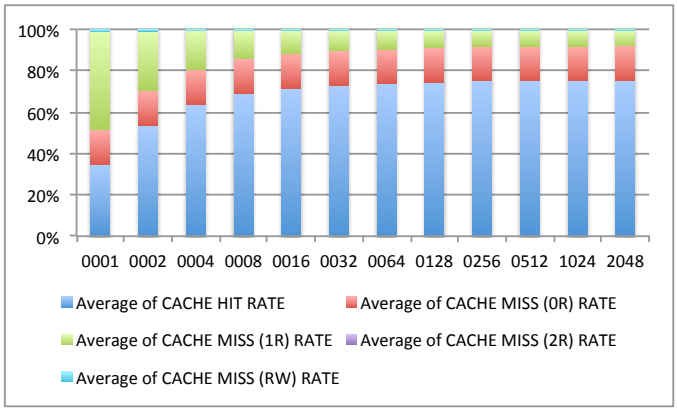


Figure 5.17: Cache performance of SCFTL with varied maximum value of consecutive field technique

field is shown in Figure 5.18.

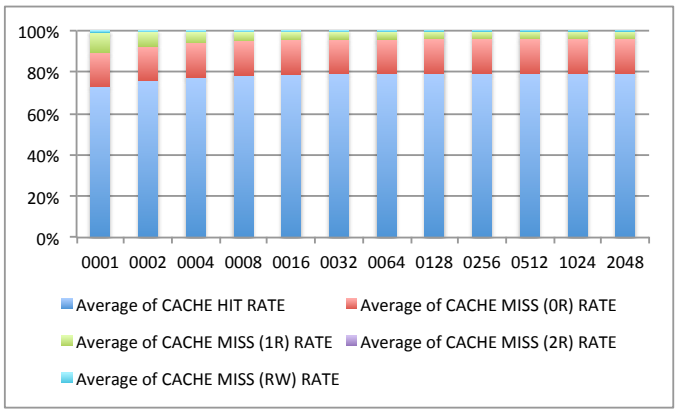


Figure 5.18: Cache performance of SCFTL with varied number of small entry and 5-bit consecutive field techniques

Finally, the comparison of ratio of cache miss with penalty (MISS (1R) and MISS (RW)) is shown in Figure 5.19, and the number of translation pages written between spatial exploitation techniques is illustrated and magnified in Figure 5.20. The best is the small cache entry with consecutive field technique as expected of the combined technique, and the worst is the large cache entry technique. Besides, the consecutive field technique is slightly worse than the small cache entry technique because of the effect of splitting a consecutive cache entry during update that may cause cache eviction.

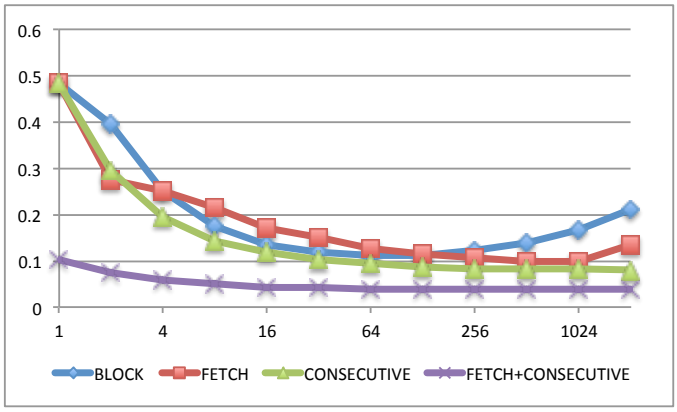


Figure 5.19: Ratio of cache miss with penalty of various spatial locality exploitation techniques

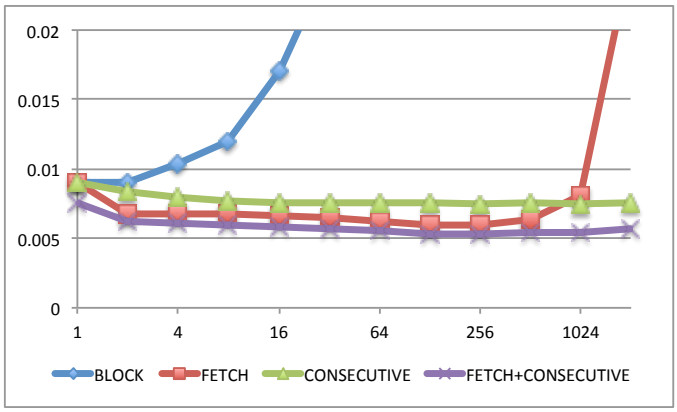


Figure 5.20: Ratio of translation page written per page write request of various spatial locality exploitation techniques

5.4 Summary

In this chapter, the high performance demand based FTL named SCFTL was described. The cache of SCFTL was optimized for exploiting spatial locality without increasing cache miss penalty, and the result is the small cache entry with consecutive field technique and D-NRU replacement policy. The spatial locality is exploited by the small cache entry with consecutive field technique. On the other hand, D-NRU keeps the impact of the the wide gap of NAND flash memory access time low by doing penalty estimation. It prevents SCFTL from exploiting spatial locality when the penalty is high and also selects a low penalty cache entry for eviction. As a result, SCFTL outperforms DFTL and CDFTL by 39.81% and 17.07%, respectively, in terms of average system response time.

CHAPTER VI

3DFTL: ZERO WRITES DEMAND-BASED FTL

In previous chapter, SCFTL, the demand-based FTL with the efficient caching strategy, is proposed. Even though SCFTL is optimized for lowering the number of flash memory programing operations, which each has much higher cost that a reading operation, some flash memory programing operations are still needed. Furthermore, the consequence of a flash programing operation is that it may need block erasure; thus, a lengthy garbage collection process is triggered and results in many fluctuations in access time. In addition, the block utilization also decrease, albeit a little.

Another problem that happened with demand-based FTLs is that an I/O write request is becoming non-atomic because the modified mapping entry is not immediately committed to a translation page or the flash memory after the data was programed. Therefore, the page-level mapping table, which resides in the flash memory, is always inconsistent with the data until all cached mapping entries are flushing back to the flash memory. As a consequence, a lengthy process of recovery has to be performed after a unexpected power-loss, which also means lower fault tolerance.

In this chapter we propose a novel demand-based FTL named 3DFTL. Without translation pages, updating data does not require additional page programing; hence, it is inconsistency-free. The cache miss ratio of 3DFTL is kept low by spatial locality exploitation. In addition, omitting translation page programing also decreases the maximum cache miss penalty, which in turn improves the average system response time.

6.1 Demand-based three-level address translation

3DFTL is a page-level FTL with a cache for the mapping table. Typically, demand-based FTLs reduce the spatial requirement of SRAM by moving PMT to data areas of flash memory pages, which in turn causes the inconsistency problem. To overcome this obstacle, 3DFTL places the PMT entries in the spare area of the pages that store their corresponding data instead. As both data and mapping information are stored in the

same page, updating is considered as an atomic operation.

Since the spare area is much smaller than the data area, PMT demands more pages for storing. Adopting the two-level address translation, as same as typical demand-based FTL, will result in large global translation directory (GTD), which resides in SRAM. In order to save precious SRAM capacity, 3DFTL decreases the number of GTD entries by inserting an local translation directory (LTD) between GTD and PMT. In other words, 3DFTL employs three-level address translation via GTD, LTD, and PMT. LTD entries are also placed in the spare area of pages along with PMT entries.

6.1.1 Three-Level Address Translation

In the first level of address translation, GTD maps an LPN to the PPN of the corresponding LTD entry, and then the LTD entry points to the PMT entry in the second level. Consequently, the PPN of the data, which is kept in PMT, can be retrieved in the third level. Since two read operations are involved, one for an LTD entry and another for a PMT entry, the address translation could take longer time.

However, the second read operation can be omitted if the required PMT entry is in the same page as the retrieved LTD entry. In other words, packing more PMT entries in one page can decrease the number of read operations. Due to the spatial locality of data write requests, the most significant bits (MSBs) of PPNs of nearby LPNs are having high likelihood of repetition. 3DFTL takes advantage of this property by employing a compression technique in order to make room for more PMT entries. A PPN, which is the content of LTD and PMT entry, is split into two parts: index (MSBs) and offset. A duplicated index is omitted from the spare area; hence, extra PPNs can be stored.

6.1.2 Compression

As illustrated in Figure 6.1, every spare area has a compression flag for indicating the format of its metadata. In this example, PPN9 is uncompressed while PPN6 and PPN11 are compressed. Both formats contain an LPN, LTD entries, and PMT entries; however, a compressed format can have more PMT entries. An uncompressed format keeps its contents unaltered since it is used in case of very low compressibility to ensure mapping integrity.

On the other hand, a compressed format stores them as a collection of distinct indices and several pairs of index position and offset. The creation of a compressed metadata begins with splitting every PMT entry that associated with the same GTD entry into index and offset. Then, a PMT entry is replaced with the related LTD entry until every distinct index can be packed into the compressed format. Since the PPN of a page has to be known before accessing, keeping it in the spare area is unnecessary. Therefore, the PPN of the compressing metadata will be replaced by its LPN, which is indispensable for garbage collection and recovery, in the next step. The LPN is also split into an index and an offset, but the index is the least significant bits (LSBs). After substituting the PPN with the LPN, the last step is sorting the indices in chronological order so that LTD entries, which always equal to their latest PMT entries, can be identified. Furthermore, the index of the LPN always holds at the first position. Combining with the fact that offsets are ordered by the LSBs of LPN, the LPN of the compressed metadata can be recomposed.

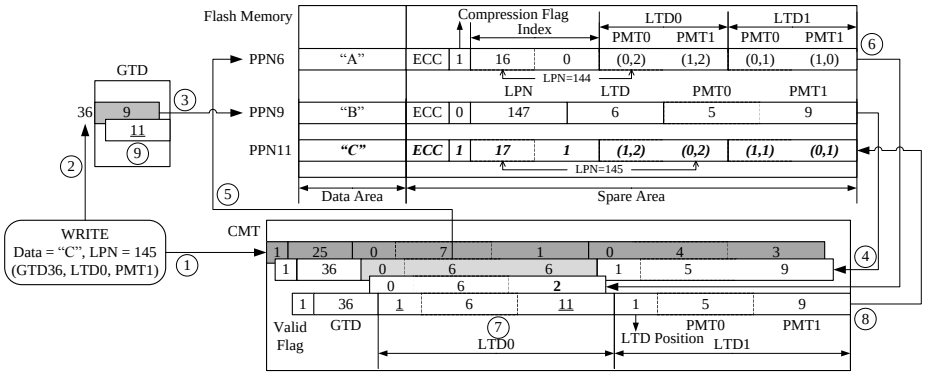


Figure 6.1: The example of 3DFTL address translation.

6.1.3 Cache

Another important component of 3DFTL is a cache of PMT called CMT. The objective of CMT is to accelerate the address translation by exploiting temporal and spatial locality. A cache lines resembles an extracted compressed metadata, except that LTD entries are pointers to their latest PMT entries. Hence, each cache line has sufficient information for generating either a compressed or uncompressed metadata without accessing the flash memory.

6.1.4 Example

An example of address translation is illustrated in Figure 6.1. We assume that, firstly, both LPN and PPN are 8-bit. Secondly, an LPN can be broken into addresses of 6-bit GTD, 1-bit LTD, and 1-bit of PMT. In other words, each GTD entry has two LTD entries, and each LTD entry has two PMT entries. Lastly, each compressed metadata is limited to only two distinct indices. In the beginning of a request for writing C at LPN145 (GTD36, LTD0, and PMT1), 3DFTL searches for GTD36 in CMT and the lookup results in a cache miss. GTD25 is selected as a victim by LRU policy and can be evicted immediately because of write-through policy. Since the PPN of LPN145 has not been cached, a three-level address translation is required. The LTD entries of GTD36 are located to be in the spare area of PPN9 according to GTD in step 2. PPN9 is read in step 3 and will be cached in step 4. However, the PPN9 metadata is uncompressed; it does not contain all PMT entries associated with GTD36. CMT will store the LTD value in place of the missing PMT entries. As the LPN of this page is 147 (GTD36, LTD1, PMT1), two PMT entries in PPN9 belong to LTD1. In other words, the PPN of LTD1 is 9, and the PPN6 in the LTD field belongs to LTD0. Since the PMT entries of LTD0 is in another page, only the second level of the address translation, accessing LTD, can be done. Next, PPN6 is read for the PMT entries of LTD0 in step 5. The PPN6 metadata is compressed. This page is PMT0 in LTD0 according to the first index, which is LSBs of the LPN. Hence, the LPN is 144 and the PMT entries are 6, 2, 5, and 0 after substituting the first index and the offset of PMT0 in LTD0 by the index and offset of the current PPN6. In this step, the PPN of LPN145 is known to be 2. The PMT entries of LTD0 will be merged to the cache line of GTD36 in step 6. In order to update LPN145, PPN2 will be invalidated and replaced by PPN11, an empty page, in step 7. In step 8, the updated cache line of GTD36 is compressed and written along with data C to PPN11. Finally, the record 36 of GTD is updated according to the modification.

6.2 Performance Evaluation

The configuration of FTLs that will be used for evaluating the performance of 3DFTL is provided in Table 6.1. According to Table 4.1, the data area of a page is 8192B while the spare area is 448B. However, only 112B are usable because of ECC. Therefore, 3DFTL is configured to have 4 LTD entries and 16 PMT entries in an uncompressed spare

area, while an compressed spare area has 4 LTD entries and 64 PMT entries by employing 8 25-bit indices. Hence, only 106KB of spare area are needed for compressed mapping entries, while the uncompressed mapping entries take only 81KB. However, the size of 3DFTL GTD is 64KB. It is much larger than GTD of other demand-based FTLs that take only about 2KB because the number of valid LTD pages is more than the number of valid translation pages.

Table 6.1: The configurations of FTLs for Chapter 6 experiments.

	DFTL	CDFTL	SCFTL	3DFTL
PMT	512 pages	512 pages	512 pages	spare area
GTD	2.00KB	2.31KB	2.25KB	64.00KB
CMT	99.00KB	33.00KB	99.00KB	32.73KB
	12,288 entries	4,096 entries	11,264 entries	128*64 entries
CTP	-	64.00KB	-	-
		8*16384 entries	-	-
GC	128KB	128KB	128KB	128KB
Total (without GC)	101KB	100.31	101.25KB	96.73KB
Total (with GC)	229KB	228.31	229.25KB	224.73KB

6.2.1 Cache Performance

The performance of the smaller cache (32KB) of 3DFTL is compared to the larger cache (96KB) of other demand-based FTLs. The cache miss ratio of 3DFTL is a little bit lower than the others because each cache entry of 3DFTL contains 16 to 64 mapping entries, which is the size that provides good cache miss ratio according to Figure 5.15. However, the downside of large cache entry technique is that every cache miss has a cache miss penalty of at least one page read in order to fetch and complete the requested cache entry. As shown in Figure 6.5, every cache miss that was caused by garbage collection always has a cache miss penalty of at least one page read despite being already known the mapping entry. Therefore, the ratio of cache miss with penalty of 3DFTL is higher than CDFTL and SCFTL as shown in Figure 6.6. Furthermore, 3DFTL also has cache misses with penalty of two pages read due to the three-level address translation, and they are as much as 2.53%. However, other FTLs have cache misses with higher penalty of one or more pages program. They are 0.64%, 0.93%, and 0.09% for DFTL, CDFTL, and SCFTL, respectively. Programing a page is over ten times slower than reading one, and it may trigger a garbage collection that requires even longer time. Having zero additional pages

programed significantly affects the translation performance as shown in Section 6.2.3.

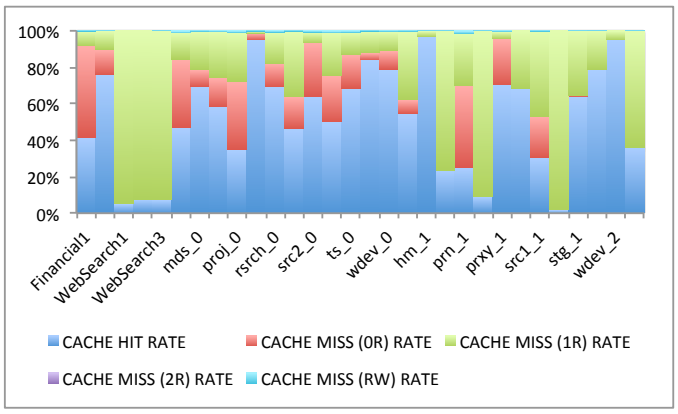


Figure 6.2: Cache performance of 100KB + 128KB DFTL

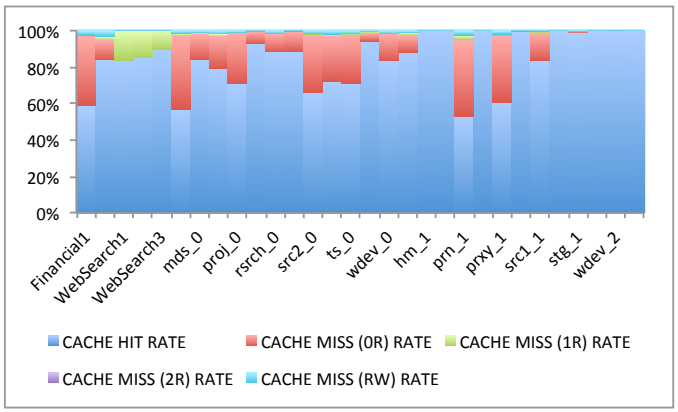


Figure 6.3: Cache performance of 100KB + 128KB CDFTL

6.2.2 Effect of Compression Technique

3DFTL employs a compression technique for reducing the cost of retrieving the last level mapping table or PMT. The compression ratio of pages written is shown in Figure 6.7. The `Compress` is the ratio of pages that was written with 64 PMT entries, while the `Uncompress` is the ratio of pages that was written with only 16 PMT entries. Lastly, the *Semicompress* is the ratio of pages that was written with only 32 or 48 PMT entries in compressed format. The benchmarks that have higher degree of spatial locality are easier to compress.

The effect of the compression technique is shown in Figure 6.8 and compared with Figure 6.9 that is the 3DFTL without compression technique. In 3DFTL without compression technique, CMT entry size is 64 PMT entries but only 16 PMT can be fetched

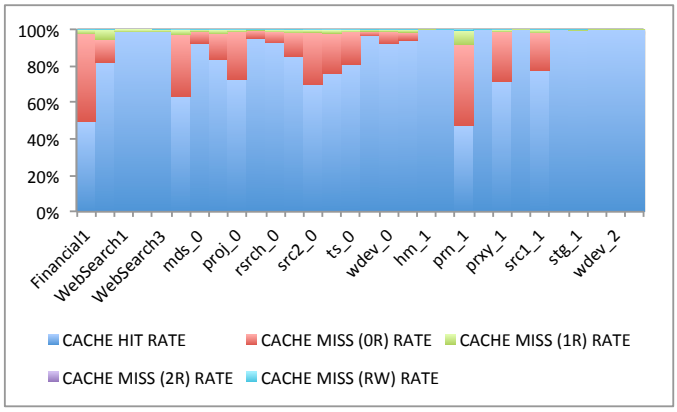


Figure 6.4: Cache performance of 100KB + 128KB SCFTL

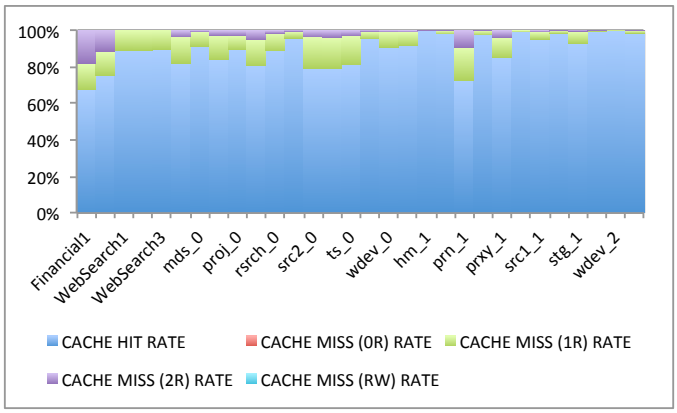


Figure 6.5: Cache performance of 100KB + 128KB 3DFTL

in each spare area read. The number of cache miss penalty with two pages read or a full three-level address translation is much higher than 3DFTL with compression technique. Moreover, the number of pages read although the LTD entry is cached is also significantly higher. Excluding the case of LTD entry cache hit, the number of address translation that was done in the first page read of 3DFTL with compression technique is higher than without compression technique.

6.2.3 Translation Performance

The translation performance of 3DFTL is illustrated in Figure 6.14. Despite the higher ratio of cache miss with penalty due to smaller cache size, 3DFTL performs better than other FTLs. It shows the speedups of 1.46 and 1.06 from DFTL and CDFTL, respectively. The wide gap of performance between DFTL and 3DFTL is an effect of spatial locality, which causes many cache misses in DFTL. Comparing 3DFTL with CDFTL,

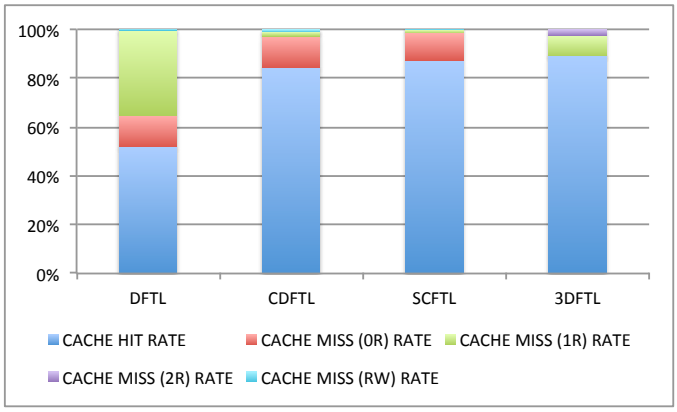


Figure 6.6: Cache performance of 100KB + 128KB FTLs

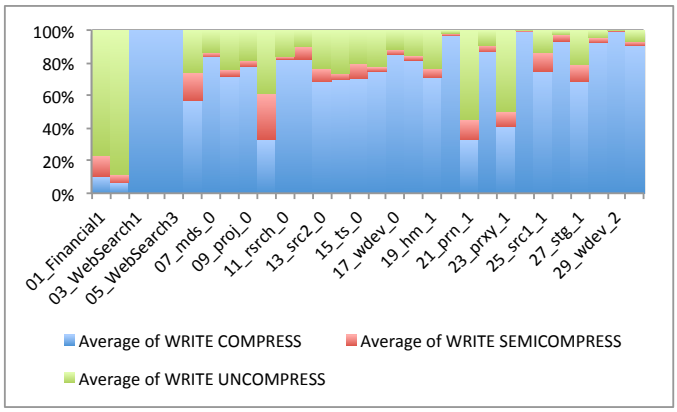


Figure 6.7: Ratio of compressed-spare-area pages written

there are large different in the several benchmarks as shown in Figure 6.12 and 6.13. These benchmarks have many hotspots of write request, i.e., many logical pages that had been written many times. Since CDFTL has fewer cache entries but more mapping entries, cache trashing can happened easier than an FTL that has more cache entries. However, the translation performance of 3DFTL is only 0.23% better than SCFTL since SCFTL catches up with 3DFTL by the lower number of cache miss with penalty.

6.2.4 Block Utilization

3DFTL does not introduce any flash memory programing operations; every page was used for storing data. On the contrary, other demand-based FTLs has to spend some pages for storing PMT. Hence, the block utilization of 3DFTL is the best among the demand-based FTLs as demonstrated in Figure 6.15. In fact, the block utilization of 3DFTL is identical to PFTL, the baseline.

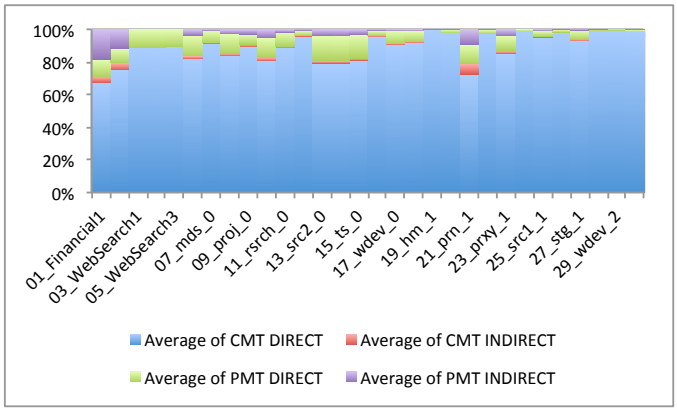


Figure 6.8: Ratio of address translation levels of 3DFTL with compression techniue

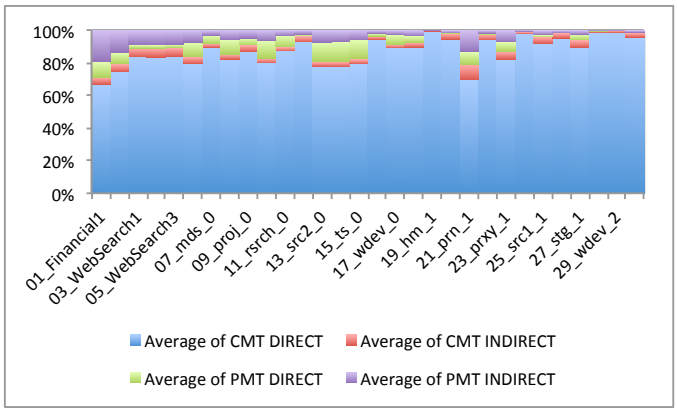


Figure 6.9: Ratio of address translation levels of 3DFTL without compression techniue

6.2.5 Garbage Collection Performance

As the selecting algorithm of the garbage collection is fixed, the performance of every FTL is equivalent in this aspect. The valid page move rate is shown in Figure 6.16.

6.2.6 Recovery

Even though the size of GTD that required to be recover of 3DFTL is significantly larger than those of other demand-based FTLs, the PMT of 3DFTL is always consistent the data. In contrast, the PMT of other demand-based FTLs may inconsistent with the data; the large PMT is also needed to be verified during recovery.

Consequently, 3DFTL has lower number of pages read during recovery than other FTLs as illustrated in Figure 6.20. Although 3DFTL exhibits more than two times faster

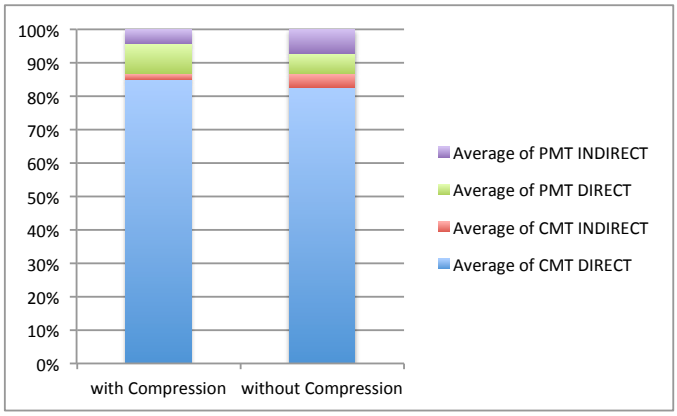


Figure 6.10: Ratio of address translation levels

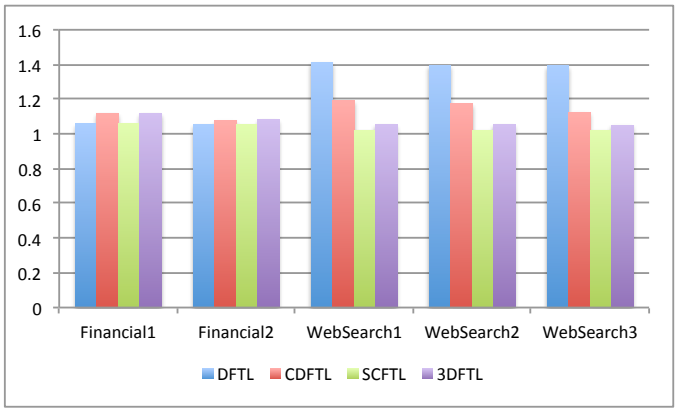


Figure 6.11: Average system response time of 100KB + 128KB FTLs on SPC benchmarks

than other demand-based FTLs, this results cannot represent general cases as the simulator was configured to make almost available pages being occupied and valid. However, more recovery details, experimental results and discussions will be in Chapter 8, which provides an additional recovery technique for 3DFTL.

6.3 Summary

In this chapter, a novel demand-based FTL named 3DFTL is proposed. It does address translation at the page-level and employs a cache of the mapping table like other demand-based FTLs. Differently, 3DFTL gets rid of translation pages by utilizing the spare areas of flash memory pages. Since the mapping information and data are simultaneously stored, the inconsistency problem is creased to exist; hence, fault tolerance is improved. However, keeping the locations of the page-level mapping table that stored in many little spare areas demands large SRAM. Thus, the three-level address translation

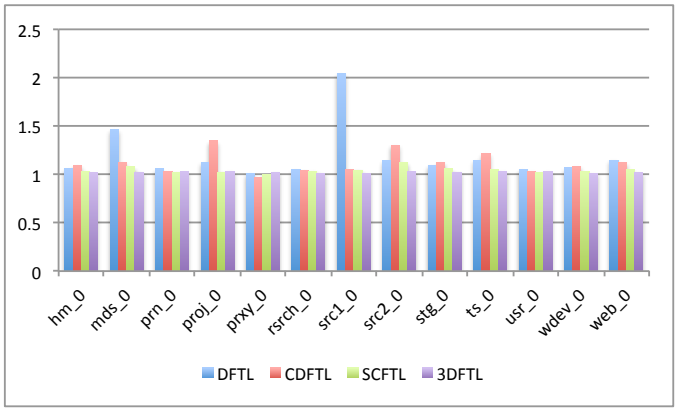


Figure 6.12: Average system response time of 100KB + 128KB FTLs on MSRC benchmarks (OS volumes)

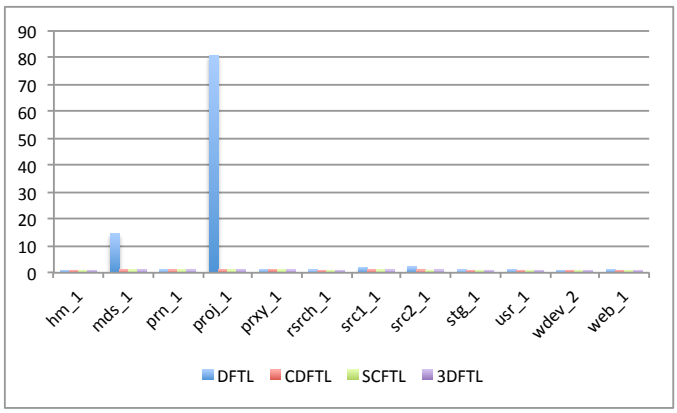


Figure 6.13: Average system response time of 100KB + 128KB FTLs on MSRC benchmarks (data volumes)

is required for controlling SRAM size. The compression and caching techniques have been applied in order to exploit the spatial locality. The average cache miss penalty is very low owing to zero explicit cache write-back operations. To sum up, 3DFTL is an economical inconsistency-free high-performance demand-based FTL. 3DFTL is more suitable for managing the flash memory in a high performance mobile device than other demand-based FTLs.

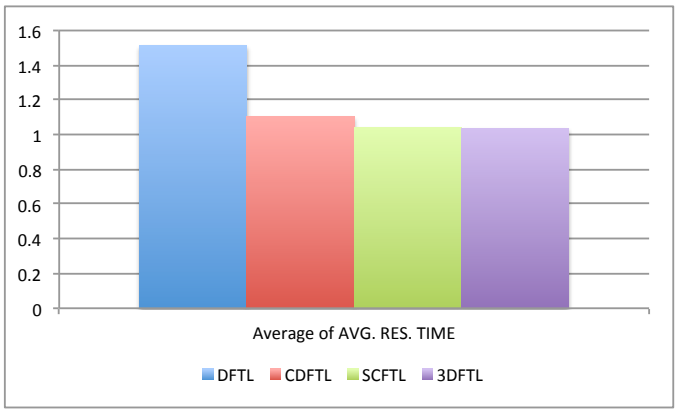


Figure 6.14: Average system response time of 100KB + 128KB FTLs

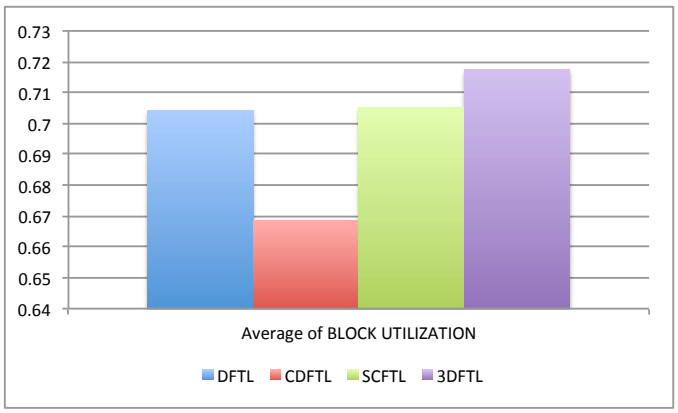


Figure 6.15: Block utilization of 100KB + 128KB FTLs

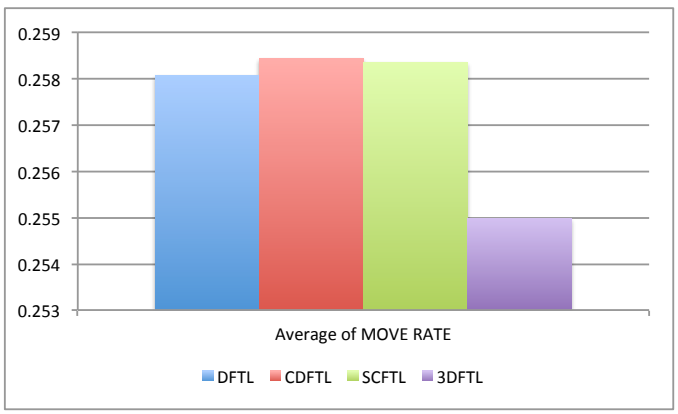


Figure 6.16: Valid page move rate of 100KB + 128KB FTLs

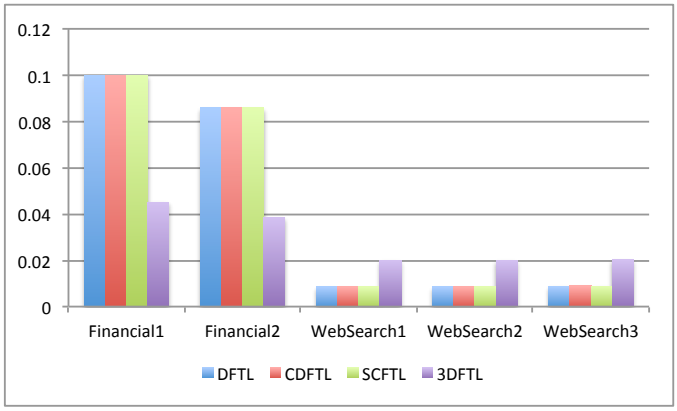


Figure 6.17: Ratio of pages read during recovery of 100KB + 128KB FTLs on SPC benchmarks

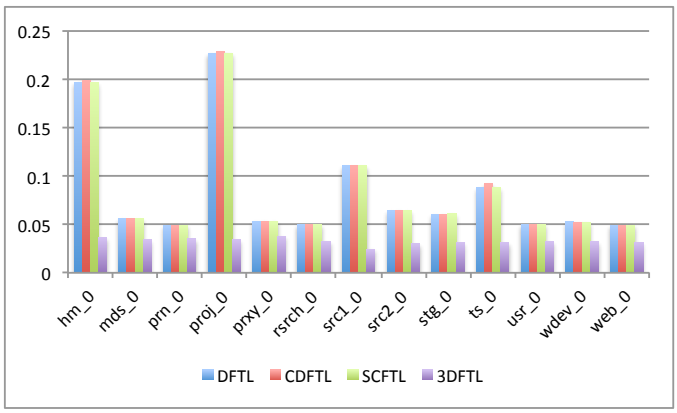


Figure 6.18: Ratio of pages read during recovery of 100KB + 128KB FTLs on MSRC benchmarks (OS volumes)

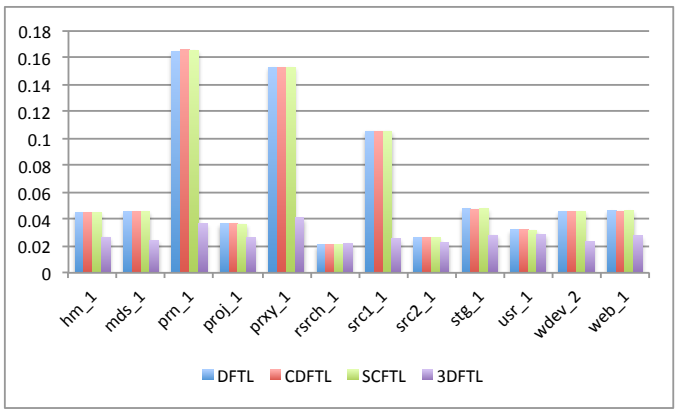


Figure 6.19: Ratio of pages read during recovery of 100KB + 128KB FTLs on MSRC benchmarks (data volumes)

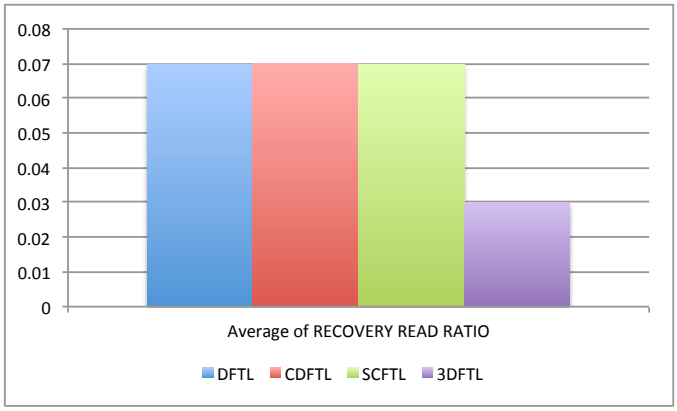


Figure 6.20: Ratio of pages read during recovery of 100KB + 128KB FTLs

Table 7.1: The cache size of FTLs for Chapter 7 experiments.

Name	Actual Cache Size (KB)				
	DFTL	CDFTL		SCFTL	3DFTL
	CMT	CMT	CTP	CMT	CMT
2KB	2.06	-	-	2.25	2.05
4KB	4.13	-	-	4.50	4.09
8KB	8.25	1.03	8.00	9.00	8.18
16KB	16.50	2.06	16.00	18.00	16.37
32KB	33.00	4.13	32.00	36.00	32.73
64KB	66.00	8.25	64.00	72.00	65.47
128KB	132.00	16.50	128.00	144.00	130.94
256KB	264.00	33.00	256.00	288.00	261.88
512KB	528.00	66.00	512.00	576.00	523.75
1024KB	1056.00	132.00	1024.00	1152.00	1047.50
2048KB	2112.00	264.00	2048.00	2304.00	2095.00
4096KB	4224.00	528.00	4096.00	4608.00	4190.00
8192KB	8448.00	1056.00	8192.00	9216.00	8380.00

CHAPTER VII

SCALABILITY

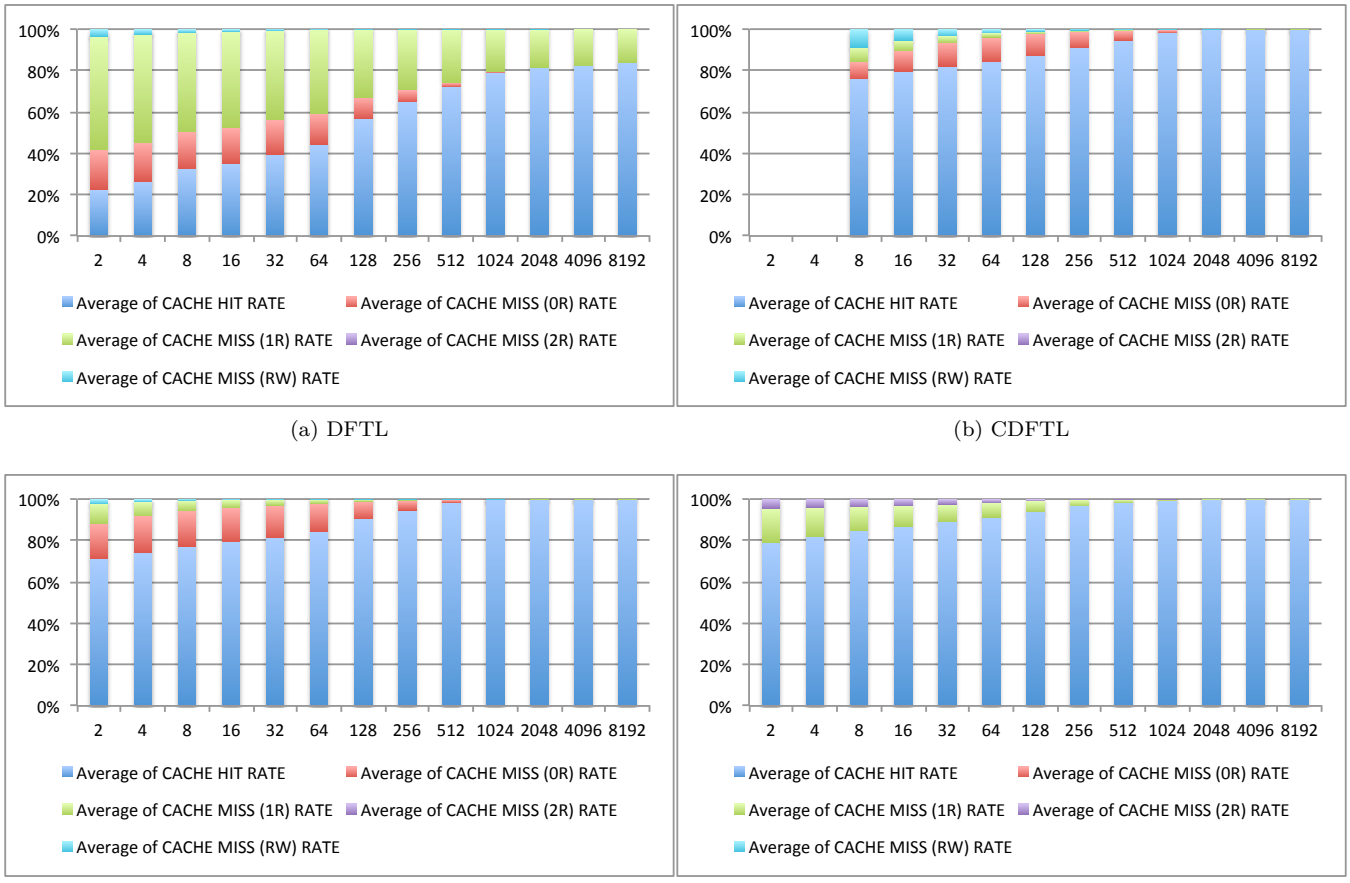
Before going to the next work, we would like to discuss about the cache scalability of SCFTL and 3DFTL. In this chapter, the cache size of FTLs will be varied without touching other configurations. The approximate cache size is varied from 2, 4, 8, ..., 8192 KBs. The maximum cache size is 8192KB although the size of PMT is only 4096KB. The reason is that the size of cache tag and status flags are included in the cache size. The actual cache sizes of FTLs are shown in Table 7.1, and the sizes of other components are shown in Table 7.2.

7.1 Cache performance

The average cache performance of FTLs is shown in Figure 7.1. The minimum cache miss ratio of DFTL is about 0.16 at the cache size of 8192KB that can cache the entire mapping table. In contrast, CDFTL and SCFTL can achieve this level of performance

Table 7.2: The controlled configurations of FTLs for Chapter 7 experiments.

FTL	DFTL	CDFTL	SCFTL	3DFTL
PMT	512 pages	512 pages	512 pages	spare area
GTD	2.00KB	2.19KB	2.25KB	64.00KB
GC	128KB	128KB	128KB	128KB
Total	130.00KB	130.19KB	130.25KB	172.00KB



(a) DFTL

(b) CDFTL

(c) SCFTL

(d) 3DFTL

Figure 7.1: Cache performance of FTLs on various cache size

with only 64KB cache. Moreover, 3DFTL already has the cache miss ratio lower than 0.16 since 8KB cache configuration. Therefore, we can conclude that only temporal locality exploitation is insufficient and spatial locality exploitation is very efficient for address translation.

The differences between cache miss with writing-back penalty ratio of SCFTL and CDFTL are more than 10 times in every configuration until it shrinking to none. In

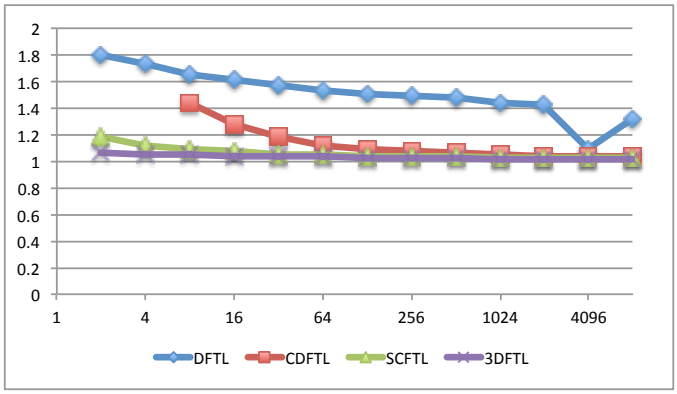


Figure 7.2: Average system response time of FTLs on various cache size

addition, the cache miss with writing-back penalty ratio of SCFTL is lower than half of DFTL in almost every configuration. In other words, the efficient caching strategy of SCFTL effectively decreases the cache miss with writing-back penalty ratio regardless of cache size.

The ratio of cache miss with fetching penalty of one page and cache miss with fetching penalty of two pages is about 3:1 to 4:1 in small cache size. The ratio is significantly increasing in higher cache size as the number of cached LTD entries increased. Hence, we could say that the compression in 3DFTL works better with larger cache size; however, it is efficient enough in small configuration when comparing to other demand-based FTLs.

7.2 Translation Performance

The geometric means of average systems response times of FTLs are shown in Figure 7.2. The smaller cache size does not have much impact on the translation performance of SCFTL and 3DFTL because the average miss penalty of both FTLs are very low. Besides, the difference of translation performance between 3DFTL with cache size of only 2KB and enormous 8192KB is only 4.71%. Therefore, SCFTL and, of course, 3DFTL are suitable for small cache.

The sudden drop of average system response times of DFTL in 4096KB configuration came from the registers in each plane of the NAND flash memory. Since our simulator does not simultaneously utilize multi planes, a translation page can be struck in the registers of one plane while the data is operating on another plane. In this case, the next cache miss may directly fetch the translation page from the registers without reread-

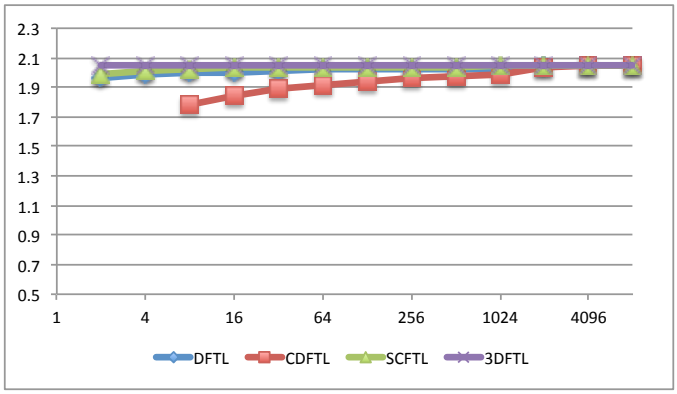


Figure 7.3: Block utilization of FTLs on various cache size

ing the actual physical page of NAND flash memory. Hence, the registers are logically become a second-level cache, and the spatial locality is exploited. As a result, the miss penalty is much lower, which in turn drastically decreases response times of sequential requests and queuing times of their subsequent accesses.

7.3 Block Utilization

Since the block utilization is a direct variation of number of blocks erased that is triggered by page programming, a demand-based FTL that frequently updates translation pages is expected to have worse block utilization, as shown in Figure 7.3.

7.4 Garbage Collection Performance

Since every FTL has the same garbage collection algorithm, the garbage collection performance is expected to have minimal difference even though the cache size are varied. As shown in Figure 7.4, the valid page moved ratios are ranged between 0.247 and 0.261.

In case of 8192KB configuration that does not have any additional pages written, the ratio of valid page moved depends on page occupied ratio. Thus, 3DFTL has the lowest valid page moved ratio. As 3DFTL does not have translation pages the ratio is remain the same for every configuration.

On the contrary, the ratios of other FTLs are decreasing with smaller cache size. The very small cache size causes the obsoleting of recently programmed translation page and can result in a translation block with many invalid translation pages. In this case,

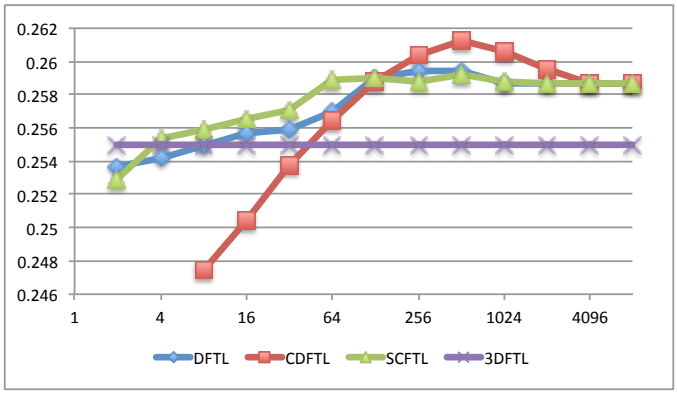


Figure 7.4: Valid page moved rate of FTLs on various cache size

the number of blocks erased is increased but the garbage collection can select a block with many invalid pages for erasure; therefore, the valid page moved ratio is decrease. However, the ratio in FTLs with medium to large cache size may not decrease and can even increase from 8192 configuration because the number invalid translation pages in a block is not that many.

CHAPTER VIII

ILOG: FAST GARBAGE COLLECTION AND RECOVERY FOR 3DFTL

In Chapter 6, we shown that the performance of 3DFTL is comparable to PFTL although the SRAM overhead is very little. Since the cache of demand-based FTLs can be resized and consume less capacity, 3DFTL can have as small cache size as 2KB but still gets very high translation performance according to Chapter 7.

However, the reduction of SRAM overhead is come from just the size of mapping table; the size of garbage collection metadata is remaining the same. As already shown in Table 5.2 and Table 6.1, the metadata required by the garbage collection of a demand-based FTL always take 128KB of SRAM capacity.

Since the greedy selection of garbage collection requires only the number of invalid pages of every block for selecting the victim block. The invalid flags of pages can be discarded; hence, the metadata of the garbage collection will demand only 4KB. However, the garbage collection has to perform a page validation by comparing the LPN and the PPN with the mapping table, and this method costs one page read operation per each page validation. Noneless, this overhead statement is not true for demand-based FTLs. Accessing the mapping table might causes a cache miss, and requires reading the translation page. Furthermore, a eviction of a modified cache entry, which includes program operations, might be needed, and it drastically worsen the overall performance.

Another problem that obstructs a page-level FTL from adoption is its lengthy recovery time. The page-level mapping table of the page-level FTL (PFTL) is stored in SRAM; thus, the mapping table will is lost with the power-outage. Without a backup of page-level mapping table, the recovery of a page-level FTL requires to find and read all valid pages for their LPNs in order to reconstruct the table. This recovery process takes unacceptable amount of time. For example, reading every page of the 8GB flash memory using in the experiments takes longer than one minute to finish.

```
1: procedure RECOVERY(FlashMemory)
2:     comment: sort PBN
3:     for all PBN ∈ FlashMemory do
4:         read the first Page of PBN for BSN
5:         if the first page of Block[PBN] is an translation page then
6:             add PBN into TranslationBlockList
7:         end if
8:     end for
9:     sort PBN by BSN

10:    comment: recover GTD from backup translation pages
11:    for all PBN ∈ TranslationBlockList in descending order of BSN do
12:        for all PTPN ∈ Block[PBN] in descending order do
13:            read LTPN and PSN from Page[PTPN]
14:            if GTD[LTPN] is empty then
15:                GTD[LTPN] ← PTPN
16:                MinPSN[LTPN] ← PSN
17:            end if
18:        end for
19:    end for

20:    comment: recover CMT
21:    for all PBN ∉ IPT.PBN ∪ TranslationBlockList in descending order of BSN do
22:        for all PPN ∈ Block[PBN] in descending order do
23:            if PSN > min(MinPSN) then
24:                read LPN and PSN from Page[PPN]
25:                convert LPN to LTPN
26:                if PSN > MinPSN[LTPN] then
27:                    if LPN ∉ CMT then
28:                        cache PPN in CMT
29:                    end if
30:                end if
31:            end if
32:        end for
33:    end for
34: end procedure
```

Figure 8.1: Pseudo code of a demand-based FTL recovery process

On the other hand, the page-level mapping table of a demand-based FTL, is kept in a non-volatile flash memory. However, some update mapping information is in the cache and is lost because of the power-outage. Consequently, the mapping table is inconsistent with the data, which in turn requires reconstruction. The recovery of a demand-based FTL involves validating every page as same as PFTL. On the contrary, the translation pages can be considered as a backup of the mapping table; hence, decrease the number of pages read. The recovery process begins with identifying the crucial backup point and then updates the mapping table with every consequent modification. The process can be described by a pseudo code as shown in Figure 8.1.

In the worst case that the crucial backup point cannot be found, the recovering process has to read as much as $m + n$ pages; where m is the number of blocks and n is the number of pages, which is the same as the page-level FTL.

Since the recovery does not know which page is invalid, reading invalid pages is unavoidable. The PPN of an invalid page has to be recorded in the mapping table and consequently will be replaced by the PPN of the valid page. Even with the complete information of invalid pages that can skip reading invalid pages, reading every consequent valid page is still essential. Therefore, the worst number of pages read during recovery is unaffected.

As mentioned earlier, the additional constraints of MLC NAND flash memory are sequential page programing and no-partial page programing. Although many FTLs, especially page-level FTLs, can effortlessly obey the sequential page programing constraint by enforcing their address translation, overcoming the no-partial page programing constraint is a different story. In SLC, an FTL can utilized the partially programs a page and keeps the updated metadata in the same erasable unit as the data without relocating. For example, keeping the status of a block, e.g., invalid flags of its pages, can prevent the garbage collection from exhaustively finding the valid pages that are needed to be moved, or recording the new location (PPN) of an obsoleted page in the spare area of the obsoleted page itself can improve fault tolerance and accelerates recovery. In addition, a demand-based FTL can also append the modification of a translation page without moving it. However, the advent of MLC NAND flash made the partial page programing no longer supported; all of these examples are not possible.

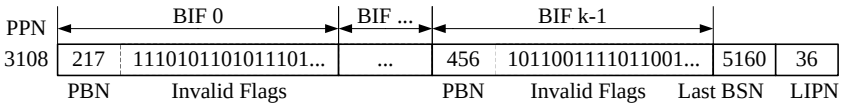
In this chapter, an additional technique for 3DFTL called Invalidation Log technique or ILog is proposed. In spite of slow page-by-page validation, ILog provides a fast validation technique when validating every page in the same block. Therefore, ILog is a swift garbage collection and a recovery accelerator for 3DFTL and is compatible with MLC NAND flash memory constraints. Moreover, ILog does not adhere to 3DFTL. Its can be applied to other page-level FTLs.

8.1 Design of ILog

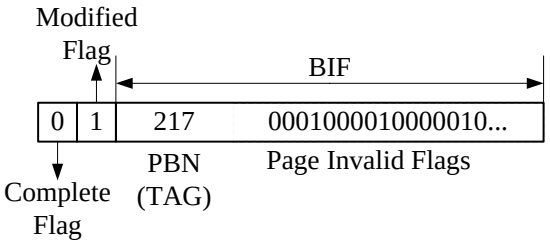
ILog or Invalidation Log technique has only one objective that is reducing validation time by validating the entire block simultaneously without keeping the complete sets of invalid flags in SRAM. In other words, the ILog objective of ILog also means shrinking SRAM overhead of invalid flags without losing the validation performance. Without invalid flags, the page validation has to be done by reading all pages in the block for their LPNs and then comparing the PPNs that are mapped from the LPNs to the PPNs of read pages. This process is done one-page-by-one-page, but the garbage collection or the recovery required validating the whole block. If a block contains only one valid page out of total 256 pages, 256 pages is need to be read and mapped in order to find the valid one; hence, it is not worthwhile.

The cache principle is applied to invalid flags. In order to reducing SRAM requirement, ILog maintain invalid flags in pages of the flash memory called invalidation page (IP) and cache some of the flags in SRAM. The cache of invalid flags is a queue named BIFQ. Since the validation is usually done on block-basis, keeping invalid flags of the same block pages together is more efficient. A group of these flags are called block of invalid flags (BIF). Therefore, IPs and BIFQ are containers of BIFs, and each BIF can validate the entire block.

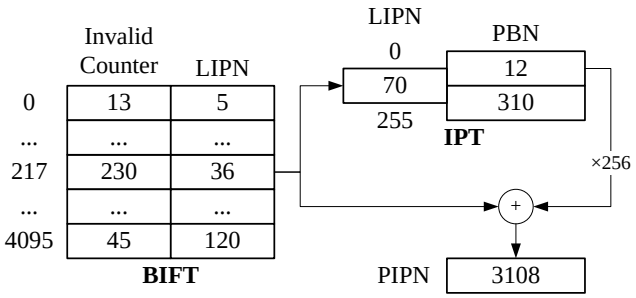
A BIF is originally stored in an IP and is reference by the physical block number (PBN) of its invalid flags owner. The PBN has to be translated into the PPN of the IP or the physical invalidation page number (PIPN). However, the number of BIF is very few and the size of BIF is very small. For example, the simulated 8GB NAND flash memory **?** has only 4,096 BIFs and each BIF is only 35B (3 bytes of PBN and 256 invalid flags); only 18 IPs are required. Due to limited SRAM capacity, a PBN will be translate into a 4-byte PIPN by two mapping tables: BIF table (BIFT) and IP table (IPT). BIFT converts the PBN into the 1-byte LIPN, and then the LIPN is mapped to the PIPN by IPT This two-table scheme needs only $\frac{\log_2(n)}{8}m$ bytes for BIFT despite $4n$ bytes of one-table scheme; where m is the number of blocks and n is the number of possible logical invalidate page number (LIPN) locations. The LIPN is consumed in round-robin; the space needed by IPT is therefore negligible because it is a very small block-level mapping table. Moreover the number of invalidated pages is also attached in each record of BIFT



(a) Invalidation Page (IP)



(b) Block invalid flag queue (BIFQ)



(c) Block invalid flag table (BIFT) and invalidation page table (IPT)

Figure 8.2: Examples of ILog components

for greedy garbage collection.

Examples of IP, BIFQ, BIFT, and IPT are illustrated in Figure 8.2.

ILog has two basic operations: validation and invalidation, which is described in the following subsections. Moreover, the reconstruction of BIFT and IPT during recovery will be detailed in Section 8.1.3

8.1.1 Invalidation

The invalidation process is rather complex since an BIFQ may has to be enqueued. It is described in Figure 8.3.

The process begins with looking for the corresponding BIF in BIFQ. If the BIF is already existed in BIFQ, the unmodified BIF will be move to the tail before modification; otherwise, the head of BIFQ will be evicted. The head BIF has to be written back to the flash memory if it was modified; hence, the target LIPN is round-robin selected, and its

IP is retrieved. very BIF in the retrieved IP has to be validated with BIFT. Then, the valid BIFs will be merged with their matched BIFs from BIFQ and temporarily stored before flushing back to the new IP. Since some BIFs is already invalid, the IP will be filled by modified BIFs from BIFQ. The modified BIFs are selected according by first-modified-first-out, the head BIF in BIFQ is the first. Moreover, if the selected modified BIF is not complete, which is identified by its complete flag in BIFT. The paired BIF has to be read from the corresponding IP before putting it into the new IP. During reading the adjacent BIFs will be merge into BIFQ. When the new IP is full, it will be programed into the flash memory and hence obsoleted the old IP. Finally the head BIF can be evicted, and the new invalid flag can be enqueued.

Indeed, the reading of IP is delayed until the eviction for the purpose of optimization. Since a BIF will be used only for validation that generally occurred only once during garbage collection, there is no need to bring it into the cache. Also, the BIF that have the full number of invalid pages will be simply erased since it block can be immediately erased without validation. However, opting out this modification from backing up into the flash memory may arise the inconsistency, but this problem will be resolved after the block erasure.

8.1.2 Validation

ILog is optimized for the efficiency of the validation process. Instead of reading all the pages in a block, ILog can validate every page in the block by just one IP reading since their invalid flags are gathered in one place BIF. The BIF in IP will be combined with the BIF in BIFQ to form the up-to-date BIF. However, the some invalid flag of invalidated pages might not be set as it may be lost and unable to recover. An invalid flag can only tell that the page status is either invalid or may be valid. Therefore, a remapping has to be performed after reading the LPN from the suspected page. If it is found invalid, the invalidation process is begun.

8.1.3 Recovery

As ILog can validate a whole block by only one IP reading, it can help decreasing the number of pages reading during the recovery. Invalid pages will be skipped and only valid pages will be read in order to get their metadata. However, the BIFs may not contains

the latest information of invalid flags. As previously stated, one reason is that the invalid flags of freshly erased block were not updated until one of its pages is invalidated. ILog deals with this problem by keeping the latest BSN to the spare area of each IP. This number guarantees that every invalid flag in this IP is usable if the BSN of the validating page, which was marked as invalid by the flag, is not over this number. In other words, its guarantees that every valid page will not be skipped.

Another reason is that the lost of invalid flags of freshly invalidated pages in BIFQ due to unexpected power-loss. This results in unset invalid flags and causes unnecessary pages reading and validating. Nevertheless, the impact is marginal and the recovery will know that a page is invalid by verifying the LPN and PPN with the recovering mapping table.

A recovery process of a demand-based FTL with ILog is shown in Figure 8.4. The process is almost as same as the previous one (Figure 8.1) except that BIFT and IPT have to be recovered before scanning and the validation is done as block-basis by the corresponding BIF in IP. A lost invalid flag will be recovered if its page is read as every read page is subjected to verification by mapping table. However, recovering every lost invalid flags during recovery is not mandatory since its will be eventually done during garbage collection.

Nevertheless, integrating ILog with PFTL, DFTL, CDFTL, or even SCFTL will not drastically change the worst situation. In the worst case, the recovery of these FTLs with ILog might require reading $2m + n$ pages, which is slightly worse than $m + n$ pages of the original. Anyhow, ILog still help these FTLs trimming invalid page from being read.

8.1.4 Integration with 3DFTL

ILog was designed for 3DFTL. The cache, BIFQ, of ILog was optimized for the small size and few flash memory accesses, especially the programing accesses that is the main strength of 3DFTL. Moreover, ILog assists 3DFTL to recover faster by avoiding reading invalid pages. The recovery processes of 3DFTL and 3DFTL with ILog are shown in Figure 8.5 and Figure 8.6, respectively.

Theoretically, if every page in the flash memory is valid, the maximum number of pages read during recovery of 3DFTL can be approximated only $m + n/c$ pages; where m is the number of blocks, n is the number of pages, and c is the number of minimum mapping entries contained a spare area.

Nevertheless, a page can be either valid or invalid, and the worst case is that only the PPN of the page that was obsoleted by the current one is gained. Hence, the estimated maximum number of pages read become $m + n/2$

On the contrary, ILog needs only one page read for one block validation; therefore, the estimated maximum number of pages read of 3DFTL with ILog is $2m + (m_{IP} * 256) + n/c$ pages; where m_{IP} is the number of blocks that contain IP.

8.2 Performance Evaluation

The comparison of the original 3DFTL against 3DFTL with ILog will be shown in this section in order to illustrate the performance of ILog. Also, the performance of DFTL and CDFTL in the similar configuration will be provided as references. The configuration of experimented FTLs are as Table 8.1.

The configuration of DFTL, CDFTL, SCFTL, and 3DFTL are almost identical with their configurations in the previous chapter, except that the invalid flags are not included in the metadata of their garbage collection and are replaced by 4,096 1-byte invalid counters. Therefore, their garbage collection have to read and map every page of the victim block.

On the other hand, 3DFTL with ILog needs about 16KB of SRAM for storing the garbage collection metadata: 8KB for BIFT and another 8KB for BIFQ. The a 4KB of invalid counters is already included in the 8KB BIFT. Since the the 16KB metadata of 3DFTL with ILog is larger than those 4KB metadata of the others, the cache (CMT) size of 3DFTL with Ilog is decreased by 12KB. As a result, the cache size of 3DFTL with ILog is only 20KB or about $\frac{1}{5}$ of DFTL, CDFTL, and SCFTL.

Table 8.1: The configurations of FTLs for Chapter 8 experiments.

	DFTL	CDFTL	SCFTL	3DFTL	3DFTL with ILog
PMT	512 pages	512 pages	512 pages	spare area	spare area
GTD	2.00KB	2.31KB	2.25KB	64.00KB	64.00KB
CMT	99.00KB	33.00KB	99.00KB	32.73KB	20.46KB
	12,288 entries	4,096 entries	11,264 entries	128*64 entries	80*64 entries
CTP	-	64.00KB	-	-	-
		8*16384 entries	-	-	-
GC	4KB	4KB	4KB	4KB	16KB
					BIFT: 8KB
					BIFQ: 8KB
					IP: max. 256 pages
Total	105KB	104.31	105.25KB	100.73KB	100.46KB

8.2.1 Validation and Invalidation Performance

The first performance aspect needed to be evaluated is the additional overhead of validation and invalidation.

Normally, the invalidation does not have much overhead if the invalid flag is store in SRAM. However, invalid flags of ILog are not stored in SRAM but the NAND flash memory in order to reduce SRAM overhead. The performance degradation is avoided by caching invalid flags in BIFQ. In order to avoid unnecessary read operation, ILog can caches only an invalid flag or an incomplete BIF in BIFQ. Anyway, BIFQ size is limited, and the eviction is eventually unavoidable. The overhead of the ILog invalidation process is shown in Table 8.2. The average ratio of IPs read per page invalidation is only 0.00048, while the average ratio of IPs written is only 0.00005 pages per page invalidation.

Next, the efficiency of ILog validation is measured by the number of pages read during garbage collection. The number of pages read includes the data pages read (for retrieving LPNs and moving), translation pages read (for retrieving PPNs), and invalidation pages read (for invalid flags). The comparison of validation efficiency is shown in Figure 8.7, and also the the numbers of pages written that are the consequence of cache eviction are shown in Figure 8.8. Nevertheless, several benchmarks was omitted in this experiments because they have very few block erasures, which means few number of invalidation. The validation overhead of ILog is very low. The average number of pages

Table 8.2: Invalidation Overhead of ILog

Benchmark	Read per Invalidation (IPs)	Write per Invalidation (IPs)
Financial1	0.001861	0.000292
Financial2	0.002539	0.000318
WebSearch1	0.000000	0.000000
WebSearch2	0.000000	0.000000
WebSearch3	0.000000	0.000000
hm_0	0.001318	0.000110
mds_0	0.000185	0.000018
prn_0	0.000157	0.000012
$proj_0$	0.000092	0.000011
$prxy_0$	0.000014	0.000002
$rsrch_0$	0.000183	0.000017
$src1_0$	0.000175	0.000040
$src2_0$	0.000538	0.000033
stg_0	0.000321	0.000020
ts_0	0.000659	0.000038
usr_0	0.000129	0.000015
$wdev_0$	0.000181	0.000019
web_0	0.000127	0.000014
hm_1	0.000000	0.000000
mds_1	0.000000	0.000006
prn_1	0.005491	0.000428
$proj_1$	0.000000	0.000014
$prxy_1$	0.000014	0.000008
$rsrch_1$	0.000000	0.000000
$src1_1$	0.000389	0.000075
$src2_1$	0.000000	0.000000
stg_1	0.000007	0.000004
usr_1	0.000000	0.000000
$wdev_2$	0.000000	0.000000
web_1	0.000000	0.000000
Average	0.000479	0.000050

read is only 1.41 per page moved. This number is less than half of other FTLs. Moreover, the number of write is 1, which is minimum, since 3DFTL does not allow any modified cache entries.

Specifically, the maximum number of invalidation page read per block erasure is only 1, which could translate to only $\frac{1}{256}$ pages read per page moved; therefore, the majority

of the additional pages read is caused by cache misses and the reason is the small CMT of 3DFTL.

8.2.2 Translation Performance

Unsurprisingly, the translation performance of 3DFTL with ILog is the better than other FTLs that do not have invalid flags. Their normalized average systems response times are shown in Figure 8.9d. Enhancing ILog give the average speedup of 1.08 to 3DFTL. When looking closely, the performance of 3DFTL with ILog is slightly worse than of 3DFTL without ILog in some benchmarks that do not have many block erasure; thus the ILog did not have many chances to perform.

For comparison purpose, the average speedups of 3DFTL with ILog over DFTL, CDFTL, and SCFTL are 1.61, 1.35, and 1.10, respectively. Furthermore, the performance of 3DFTL with ILog, which takes only 100.46KB of SRAM, is only 3.90% difference from the enormous PFTL, albeit 42 times difference in terms of SRAM overhead.

Additionally, the 100KB 3DFTL with ILog is compared to the 212KB 3DFTL that has every invalid flag in SRAM. The 212KB 3DFTL has the same configuration as the 100KB 3DFTL with ILog that is used throughout this chapter, but every invalid flag of 212KB 3DFTL can be stored in SRAM; thus the validation and invalidation can be done instantaneously. Nevertheless, the recovery accelerator ability is lost. The comparison is shown in Figure 8.10. Surprisingly, The 100KB 3DFTL with ILog is only 0.18% faster; however, the the difference is caused by only $\texttt{proj}_0$, which is a write dominant I/O trace. In fact, the access time of 3DFTL with ILog is slower than 3DFTL with complete invalid flags, but the queuing time of 3DFTL with ILog is faster. The faster queuing time is caused by periodically fewer number of pages moved when the garbage collection select a block of invalidation pages for erasure. Therefore, it can be concluded that ILog is as efficient as storing every invalid flag in SRAM. Furthermore, ILog helps 3DFTL recover faster.

8.2.3 Block Utilization

Adding Ilog to 3DFTL does not have much effect on the block utilization. The results are almost identical to the 3DFTL without ILog that has larger cache as shown in

Figure 8.11. However, there are a large drop in `WebSearch1`, `WebSearch2`, and `WebSearch3` benchmarks, but these benchmarks have very few number of block erasure; thus, the block utilization that is a ratio of page-write requests per block erasure is fluctuated.

8.2.4 Garbage Collection Performance

As same as previous chapters, the garbage collection performance is undifferentiated, and the value is about 25% to 26% as shown in Figure 8.12

8.2.5 Recovery Performance

As shown in Figure 8.13, the 3DFTL with ILog can recover faster than others page-level FTLs. Averagely, it needed to read only 2.61% of total pages to recover from unexpected shutdown. The speedups of 3DFTL with ILog from 3DFTL, SCFTL, CDFTL, DFTL, and PFTL are only 1.15, 2.67, 2.67, 2.67, and 38.42, respectively.

However, the normal setup of our simulation cannot shown their worst case situations since the worst case recovery of PFTL, DFTL, CDFTL, and SCFTL require reading $m + n$ pages; where m is the number of blocks and n is the number of pages, while the worst case of 3DFTL is $m + n/2$ pages. Since every user accessible capacity is fully occupied for the purpose of stressing the address translation and controlling garbage collection performance, almost pages are valid and very few invalid pages are existed; thus, every FTL can perform well.

Generally, the storage are not 100% filled, and other FTL functional units can manipulate the storage. Moreover, the special I/O command, TRIM, enable the file system to communicate with the FTL and lets the FTL invalidate pages without written it. This leaves certain amount of invalid pages; hence, the garbage collection can work efficiently. On the contrary many invalid pages may bring the worst case situation to the recovery.

Therefore, another experiments had been done for evaluating the recovery in the more general case. The simulator are configured to fill the storage thoroughly several times before beginning the simulation, but the number of valid pages can be as minimum as 25% of total capacity. However, many benchmarks need more space; hence, the number of valid pages are ranged from 25% to 97%. The recovery results of this setting is in Figure 8.14.

3DFTL with ILog can still show its performance with the average number of pages read during recovery of 1.82% of total capacity. In contrast, other FTLs performances are much worse than the previous setting with the average number of pages read of 20.78% for 3DFTL and about $35 - 37\%$ for DFTL, CDFTL and SCFTL.

8.3 Summary

ILog is an technique for caching invalid flags, which provides a shortcut for block validation during the garbage collection. Instead of keeping every invalid flags in SRAM, ILog keeps only a small portion in SRAM while the complete set is maintained in the flash memory. Moreover, keeping them in the flash memory, which is a non-volatile memory, also benefits the recovery as it can correctly identify valid pages without a false negative.

Furthermore, ILog is not a dedicated technique of 3DFTL. It does not need any information the address translation. The only information that ILog needed is the PPN of an invalidating page and the PBN of a validating block. Thus, ILog can be applied to other FTLs.

ILog avoid excessive flash memory reading and programing by delaying them until necessary. Therefore, by keeping the operation cost low, ILog is very efficient as already shown in the experimental results. The 3DFTL with ILog is 8.04% faster than the one without invalid flags and almost indifference from the one that stores every invalid flag in SRAM in terms of average system response time. To summarize, the speedup of 3DFTL with ILog is 1.61, 1.35, and 1.10 from DFTL, CDFTL, and SCFTL, respectively.

```
1: procedure INVALIDATE(PPN)
2:     convert PPN to PBN
3:     if PBN ∉ BIFQ then
4:         if the head of BIFQ is modified then
5:             VictimLIPN ← OldestLIPN
6:             VictimPIPN ← IPT[VictimLIPN]

7:             comment: create new invalidation page
8:             read Page[VictimPIPN]
9:             for all BIF ∈ Page[VictimPIPN] do
10:                if BIFT[BIF.PBN] = VictimLIPN then
11:                    if BIF.PBN ∈ BIFQ then
12:                        merge BIF with BIFQ[PBN].BIF
13:                        set BIFQ[PBN].CompleteFlag
14:                    end if
15:                    add BIFQ[PBN].BIF to NewIP
16:                    clear BIFQ[PBN].ModifiedFlag
17:                end if
18:            end for

19:            comment: fill new invalidation page
20:            while NewIP is not full do
21:                get NextPBN from head to tail of BIFQ
22:                if BIFQ[NextPBN] is modified then
23:                    if BIFQ[NextPBN] is not complete then
24:                        NextLIPN ← BIFT[NextPBN].LIPN
25:                        NextPIPN ← IPT[NextLIPN]
26:                        read Page[NextPIPN]
27:                        for all BIF ∈ Page[NextPIPN] do
28:                            if BIFT[BIF.PBN] = NextLIPN then
29:                                if BIF.PBN ∈ BIFQ then
30:                                    merge BIF with BIFQ[PBN].BIF
31:                                    set BIFQ[PBN].CompleteFlag
32:                                end if
33:                            end if
34:                        end for
35:                    end if
36:                    add BIFQ[NextPBN].BIF to NewIP
37:                    clear BIFQ[NextPBN].ModifiedFlag
38:                    BIFT[NextPBN].LIPN ← VictimLIPN
39:                end if
40:            end while

41:            comment: write-back
42:            VictimPIPN ← GETEMPTYPAGE
43:            program NewIP to Page[VictimPIPN]
44:            update IPT
45:        end if

46:        comment: replacement
47:        remove the head of BIFQ
48:        add BIF[PBN] to the tail of BIFQ
49:    end if

50:    comment: invalidation
51:    Set BIF[PBN].invalidFlags[PPN] in BIFQ
52: end procedure
```

Figure 8.3: A flow chart of ILog invalidation process

```
1: procedure RECOVERY(FlashMemory)
2:     comment: sort PBN and Recover IPT
3:     for all PBN ∈ FlashMemory do
4:         read the first Page of PBN for BSN
5:         if the first page of Block[PBN] is an invalidation page then
6:             add PBN into IPT
7:         else if the first page of Block[PBN] is an translation page then
8:             add PBN into TranslationBlockList
9:         end if
10:    end for
11:    sort PBN by BSN

12:    comment: recover BIFT
13:    for all PBN ∈ IPT.PBN in ascending order of BSN do
14:        for all PIPN ∈ Block[PBN] in ascending order do
15:            read Page[PIPN]
16:            set IPT.LIPN according to Page[PIPN].LIPN
17:            for all BIF ∈ Page[PIPN] do
18:                BIFT[BIF.PBN].LIPN ← Page[PIPN].LIPN
19:                BIFT[BIF.PBN].InvalidCount ← Count(BIF.InvalidFlags)
20:            end for
21:        end for
22:    end for

23:    comment: recover GTD from backup translation pages
24:    for all PBN ∈ TranslationBlockList in descending order of BSN do
25:        for all PTPN ∈ Block[PBN] in descending order do
26:            read LTPN and PSN from Page[PTPN]
27:            if GTD[LTPN] is empty then
28:                GTD[LTPN] ← PTPN
29:                MinPSN[LTPN] ← PSN
30:            end if
31:        end for
32:    end for

33:    comment: recover CMT
34:    for all PBN ∉ IPT.PBN ∪ TranslationBlockList in descending order of BSN do
35:        read BIF from Page[IPT.PIPN[BIFT[PBN].LIPN]]
36:        for all PPN ∈ Block[PBN] in descending order do
37:            if PSN > min(MinPSN) then
38:                if PPN ∉ BIF.InvalidFlags then
39:                    read LPN and PSN from Page[PPN]
40:                    convert LPN to LTPN
41:                    if PSN > MinPSN[LTPN] then
42:                        if LPN ∉ CMT then
43:                            cache PPN in CMT
44:                        end if
45:                    end if
46:                end if
47:            end if
48:        end for
49:    end for
50: end procedure
```

Figure 8.4: Pseudo code of a demand-based FTL with ILog recovery process

```
1: procedure RECOVERY(FlashMemory)
2:     comment: Sort PBN
3:     for all PBN ∈ FlashMemory do
4:         read the first Page of PBN for BSN
5:     end for
6:     sort PBN by BSN

7:     comment: recover GTD
8:     for all PBN ∈ FlashMemory in descending order of BSN do
9:         for all PPN ∈ Block[PBN] in descending order do
10:            if PPN ∉ SkipList then
11:                read LPN, LTD, and PMT from Page[PPN]
12:                convert LPN to LTPN
13:                if GTD[LTPN] is empty then
14:                    GTD[LTPN] ← PPN
15:                end if
16:                for all PPN ∈ LTD ∪ PMT do
17:                    add PPN into SkipList
18:                end for
19:            end if
20:        end for
21:    end for
22: end procedure
```

Figure 8.5: Pseudo code of 3DFTL without ILog recovery process

```
1: procedure RECOVERY(FlashMemory)
2:     comment: sort PBN and recover IPT
3:     for all PBN ∈ FlashMemory do
4:         read the first Page of PBN for BSN
5:         if the first page of Block[PBN] is an invalidation page then
6:             add PBN into IPT
7:         end if
8:     end for
9:     sort PBN by BSN

10:    comment: recover BIFT
11:    for all PBN ∈ IPT.PBN in ascending order of BSN do
12:        for all PIPN ∈ Block[PBN] in ascending order do
13:            read Page[PIPN]
14:            set IPT.LIPN according to Page[PIPN].LIPN
15:            for all BIF ∈ Page[PIPN] do
16:                BIFT[BIF.PBN].LIPN ← Page[PIPN].LIPN
17:                BIFT[BIF.PBN].InvalidCount ← Count(BIF.InvalidFlags)
18:            end for
19:        end for
20:    end for

21:    comment: recover GTD
22:    for all PBN ∉ IPT.PBN in descending order of BSN do
23:        read BIF from Page[IPT.PIPN[BIFT[PBN].LIPN]]
24:        add BIF.InvalidFlags into SkipList
25:        for all PPN ∈ Block[PBN] in descending order do
26:            if PPN ∉ SkipList then
27:                read LPN, LTD, and PMT from Page[PPN]
28:                convert LPN to LTPN
29:                if GTD[LTPN] is empty then
30:                    GTD[LTPN] ← PPN
31:                else
32:                    add PPN into BIFQ
33:                    BIFT[PBN].InvalidCount++
34:                end if
35:                for all SkipPPN ∈ LTD ∪ PMT do
36:                    add SkipPPN into SkipList
37:                end for
38:            end if
39:        end for
40:    end for
41: end procedure
```

Figure 8.6: Pseudo code of 3DFTL with ILog recovery process

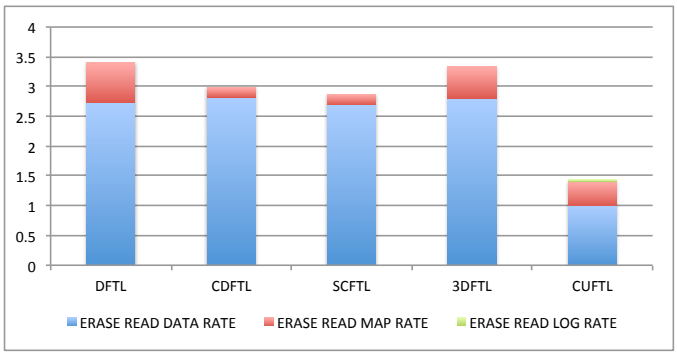


Figure 8.7: Number of pages read per valid page moved of 100KB FTLs

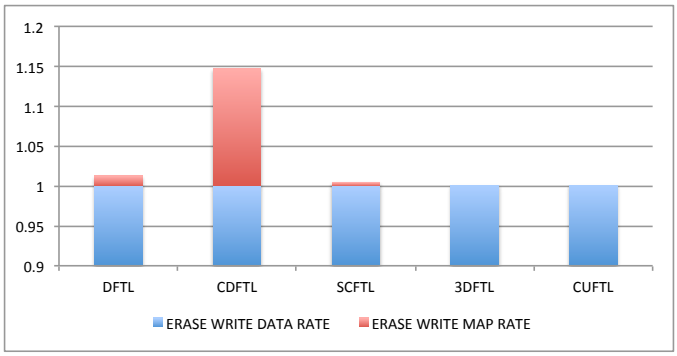
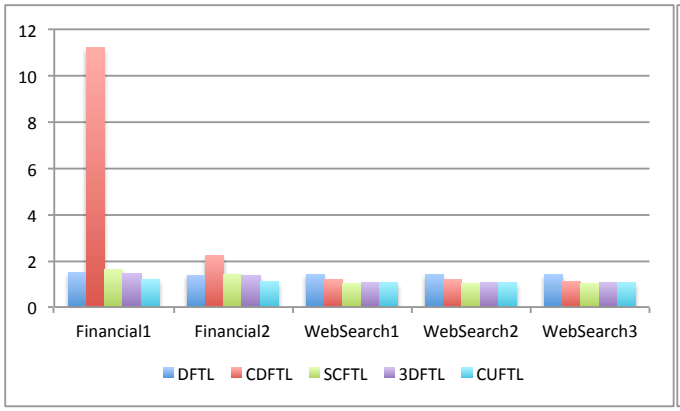
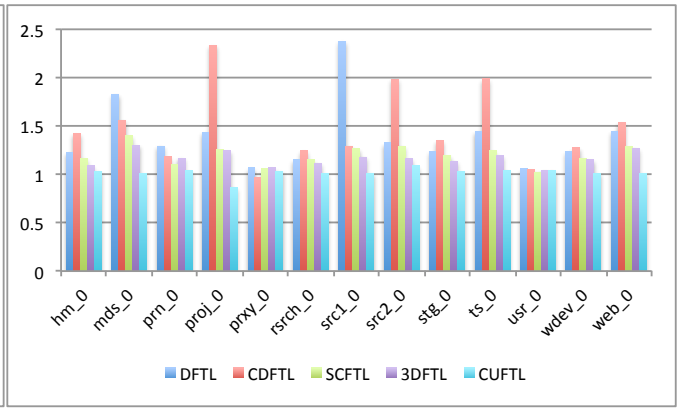


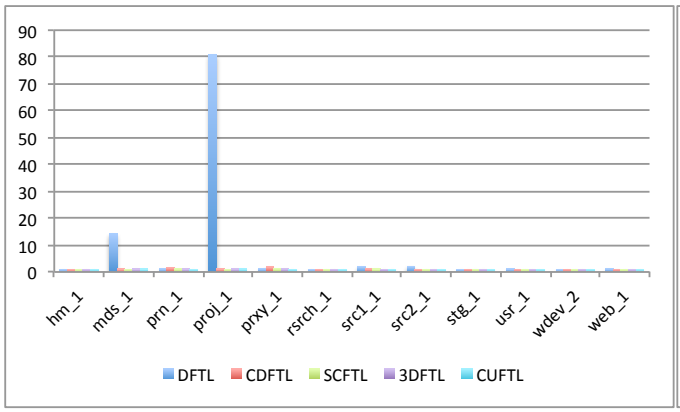
Figure 8.8: Number of pages programed per valid page moved of 100KB FTLs



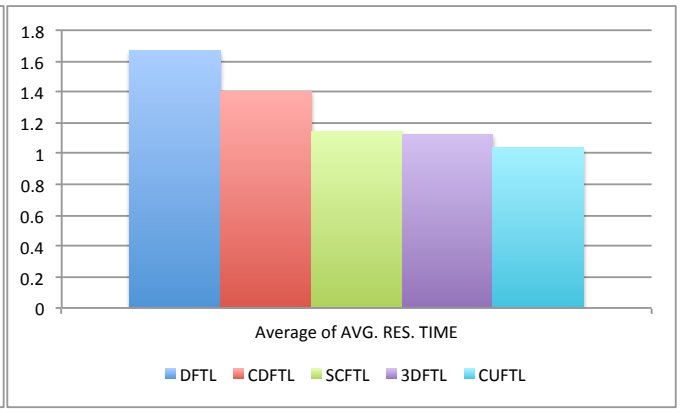
(a) SPC benchmarks



(b) MSRC benchmarks (OS Volumes)

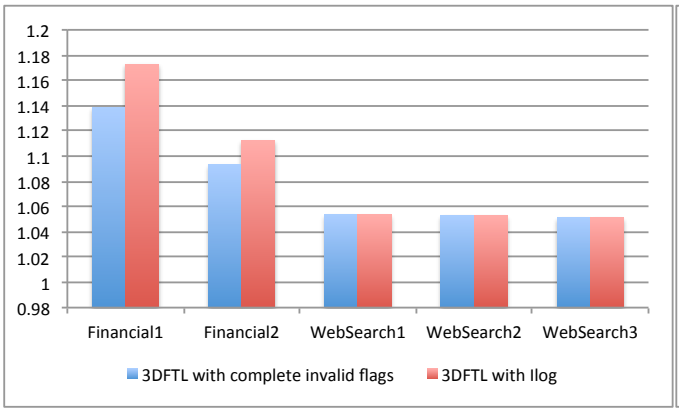


(c) MSRC benchmarks (data volumes)

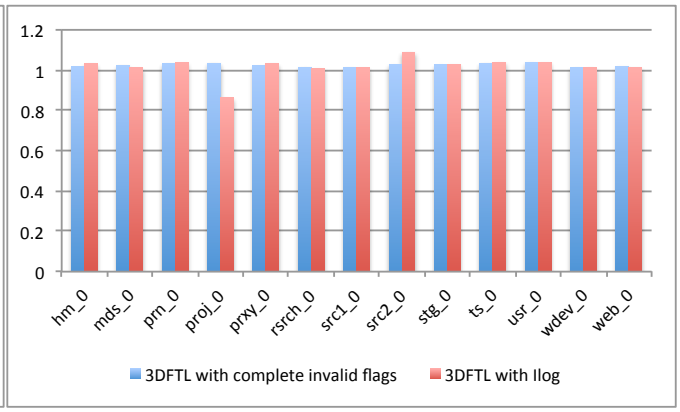


(d) Average

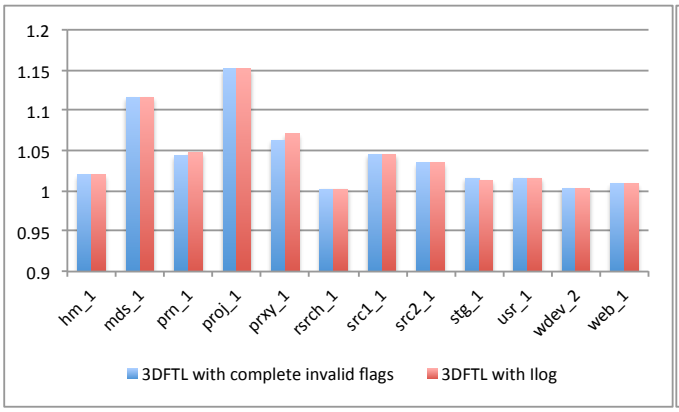
Figure 8.9: Average system response time of 100KB FTLs



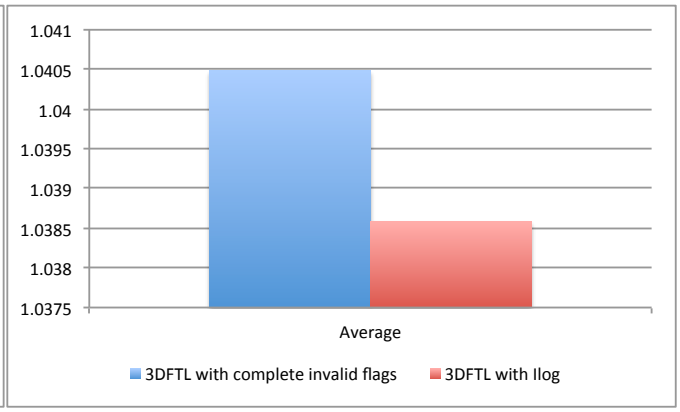
(a) SPC benchmarks



(b) MSRC benchmarks (OS Volumes)



(c) MSRC benchmarks (data volumes)



(d) Average

Figure 8.10: Average system response time of 3DFTL without invalid flags, 3DFTL with ILog, and 3DFTL with complete invalid flags

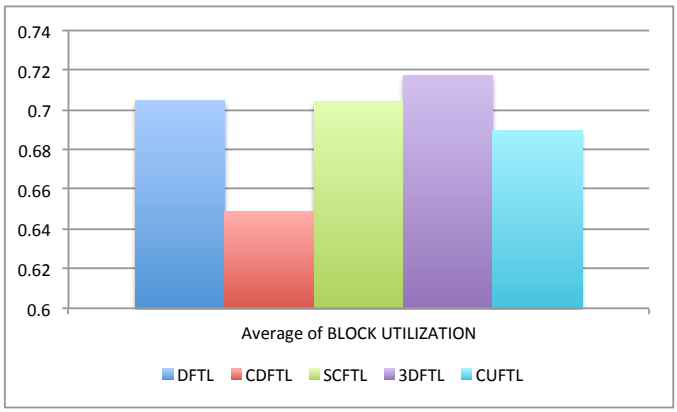


Figure 8.11: Block utilization of 100KB FTLs

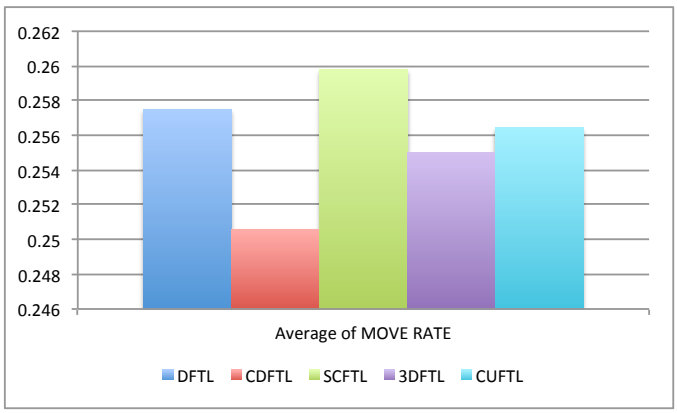
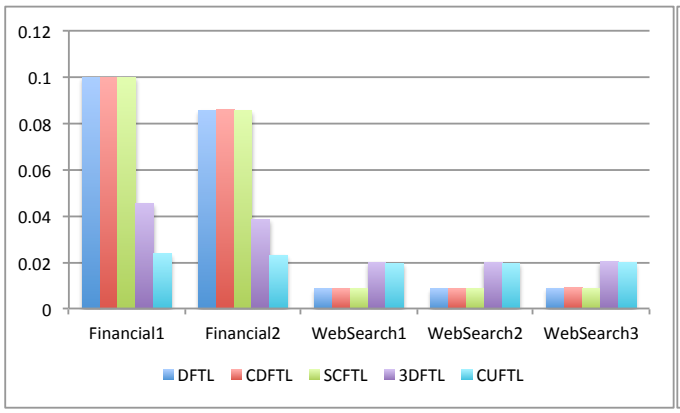
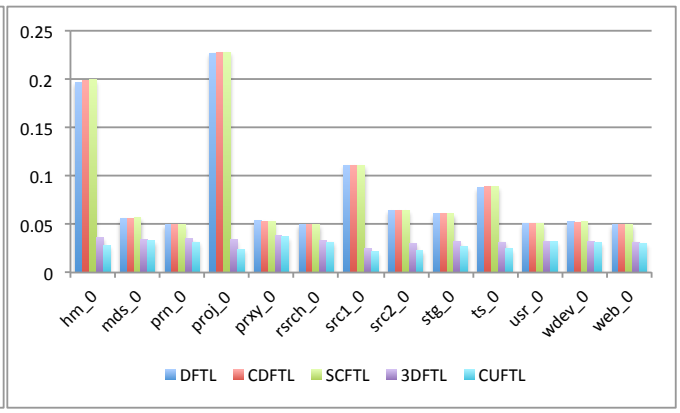


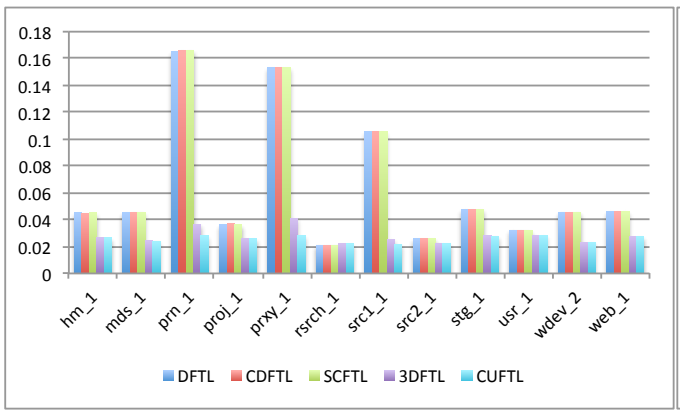
Figure 8.12: Valid page move rate of 100KB FTLs



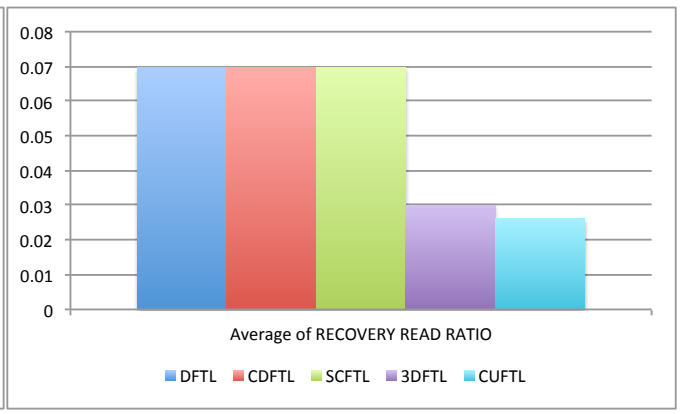
(a) SPC benchmarks



(b) MSRC benchmarks (OS Volumes)

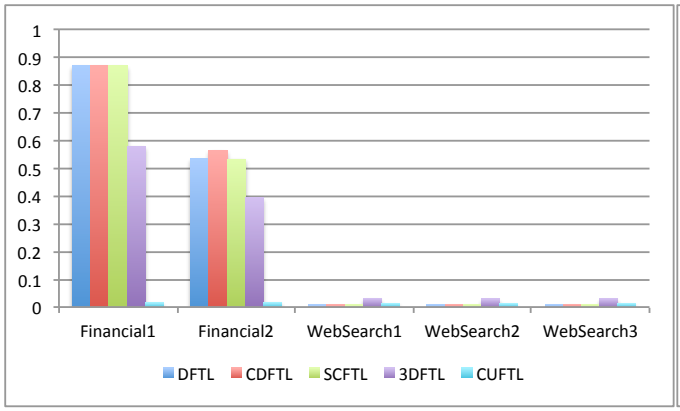


(c) MSRC benchmarks (data volumes)

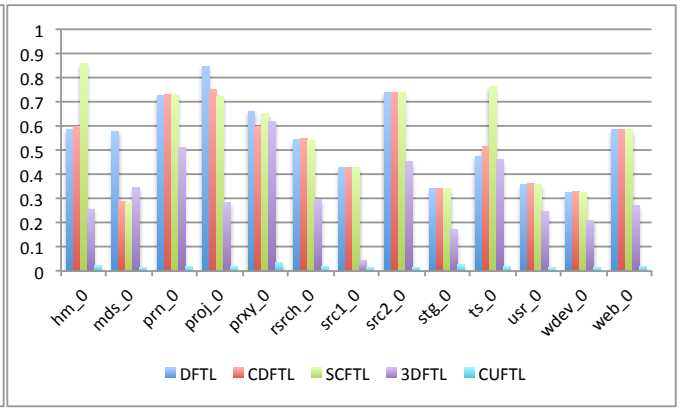


(d) Average

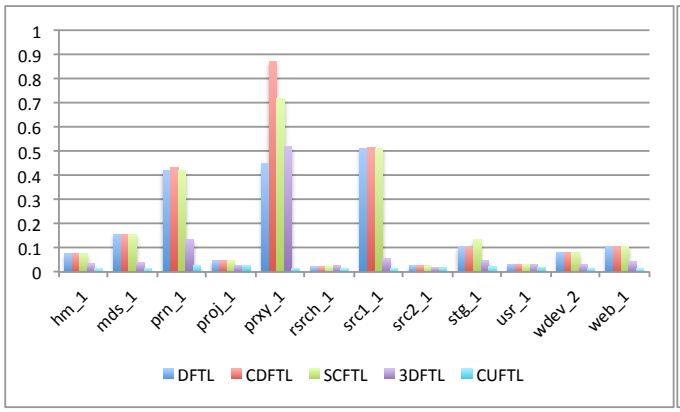
Figure 8.13: Ratio of pages read during recovery of 100KB FTLs



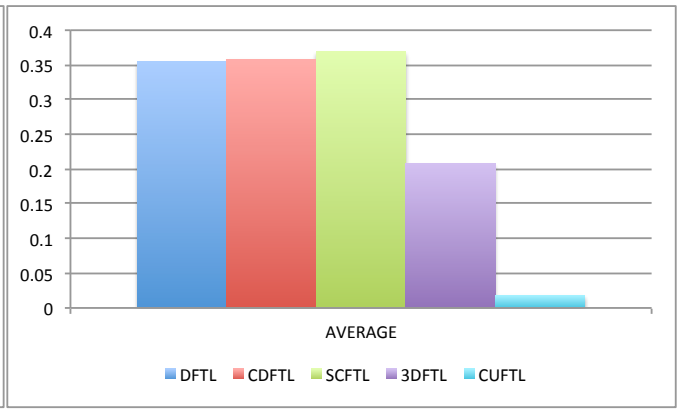
(a) SPC benchmarks



(b) MSRC benchmarks (OS Volumes)



(c) MSRC benchmarks (data volumes)



(d) Average

Figure 8.14: Ratio of pages read in general case recovery of 100KB FTLs

CHAPTER IX

CONCLUSION

9.1 Dissertation Contributions

Since NAND flash memory was invented, its adoption has been continuously growing and eventually became ubiquitous. One of the main factors that drives this growth is the user-friendly and efficient management of flash translation layer or FTL. In this dissertation, three novel approaches for demand-based FTL have been proposed and discussed. To summarize, the contributions of this dissertation are consisted of this three approaches.

First SCFTL is a novel demand-based FTL that is aware of asymmetrical access time. It has several techniques working together in order to maximize the benefit of each translation page programming operation and limit unessential fetches. The result is very fast response time even though the very small cache size.

Second 3DFTL is also a novel demand-based FTL that is aware of asymmetrical access time. By eliminating explicit translation page, 3DFTL utilizes the spare area of flash memory pages for storing mapping table. A three-level address translation is employed due to the limited size of each page spare area. However, 3DFTL decreases the additional page reading operation by caching and compression techniques. Therefore, 3DFTL not only is responsive, but also wears the flash memory cell slower.

Third ILog is an additional technique for 3DFTL; however, it should be able to adapt for other FTL as well. The main idea behind ILog is providing block-level validation for page-level FTL without much increasing overhead. It is a cache of invalid flags, but an invalid flag is kept for using only once during garbage collection; hence, ILog also includes this rule in to consideration. It will not fetch any invalid flag until necessary, and will write-back only unused invalid flags.

The proposed FTLs have been thoroughly verified by experiments. The performance improvement of each is significant as shown in the experimental results. Regardless of the ILog, 3DFTL is the best in terms of translation performance or average system response time; however, the large global translation directory (GTD) might make SCFTL more suitable for a flash memory-based storage with a very limited SRAM. ILog is indispensable for 3DFTL. With only tiny SRAM and flash memory capacities required, ILog accelerates not only the garbage collection, but also the recovery. The recovery time is also guaranteed; therefore, 3DFTL with ILog is very appropriate for a flash memory-based storage in enterprise servers and mobile devices.

9.2 Discussion on Future Works

Despite of several benefits, the proposed FTLs still have limitations that could be improved in the future.

- As already mentioned in the previous section, 3DFTL has a large global translation directory (GTD). According to the observation, the whole GTD is not accessed simultaneously. In fact, we found that only parts of it is accessing during a certain period. Offloading GTD to the flash memory and caching some entries in SRAM might save the precious SRAM capacity. In addition, the offloaded GTD, which is in the flash memory, may act as a backup point; hence, the recovery time is hasten. However, adding GTD cache will result in four-level address translation; hence, its effectiveness is still opened for validation.
- Integrating ILog with SCFTL – or other page-level FTLs – should be practical, but it still has not been done. Therefore, the enhancement has not been verified although we expects a significant improvement over the non-enhanced SCFTL. Moreover, maintaining both translation page and invalidation page may arise complications.
- The scalability of ILog has not been studied, but we are not expecting performance improvement from an increasing cache, BIFQ, size. On the contrary, the increased in the number of pages read during recovery is more concerned.
- Wear leveling was not focused in this dissertation, but it would be interesting to see the suitable one for these FTLs. In addition, our preliminary results found that classifying hot and cold pages might drastically improve translation performance.

- Also, load balancing and parallelization are not focused in this dissertation. The scheduling of operations during a cache miss in multi-plane and multi-die flash memory should be explored.
- Finally, applying these technique to a file system or having the OS assisting these FTL are still in question.