DESIGN AND SIMULATION OF A FLASH TRANSLATION LAYER ALGORITHM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YUSUF YAVUZ AYAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

MAY 2010

Approval of the thesis:

DESIGN AND SIMULATION OF A FLASH TRANSLATION LAYER ALGORITHM

submitted by **YUSUF YAVUZ AYAR** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Dr. Attila Özgit _____
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Payidar Genç
Computer Engineering Dept., METU _____

Dr. Attila Özgit
Computer Engineering Dept., METU _____

Dr. Onur Tolga Şehitoğlu
Computer Engineering Dept., METU _____

Dr. Cevat Şener
Computer Engineering Dept., METU _____

Assoc. Prof. Özgür Barış Akan
Electrical and Electronics Engineering Dept., METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: YUSUF YAVUZ AYAR

Signature :

ABSTRACT

DESIGN AND SIMULATION OF A FLASH TRANSLATION LAYER ALGORITHM

Ayar, Yusuf Yavuz
M.Sc., Department of Computer Engineering
Supervisor : Dr. Attila Özgit

May 2010, 59 pages

Flash Memories have been widely used as a storage media in electronic devices such as USB flash drives, mobile phones and cameras. Flash Memory offers a portable and non-volatile design, which can be carried to everywhere without data loss. It is durable against temperature and humidity. With all these advantages, Flash Memory gets popular day by day. However, Flash Memory has also some disadvantages, such as erase-before restriction and erase limitation of each individual block. Erase-before restriction pushes every single writable unit to be erased before an update operation. Another limitation is that every block can be erased up to a fixed number. Flash Translation Layer - FTL is the solution for these disadvantages. Flash Translation Layer is a software module inside the Flash Memory working between the operating system and the memory. FTL tries to reduce these disadvantages of Flash Memory via implementing garbage collector, address mapping scheme, error correcting and many others. There are various Flash Translation Layer software. Some of them have been reviewed in terms of their advantages and disadvantages. The study aims at designing, implementing and simulating a NAND type FTL algorithm.

Keywords: Flash Memory, NAND, NOR, Address Mapping, Flash Translation Layer

ÖZ

BELLEK DÖNÜŞTÜRME KATMANI TASARIM VE SİMULASYONU

Ayar, Yusuf Yavuz
Yüksek Lisans, Bilgisayar Mühendisliği
Tez Yöneticisi : Dr. Attila Özgit

Mayıs 2010, 59 sayfa

Anlık (Flash) bellekler günümüzde kullanımı gün geçtikçe artan, mobil, sağlam, dayanıklı, tamamen elektronik olan hafıza cihazlarıdır. Anlık belleklerin doğasından gelen bazı dezavantajları vardır, örnek olarak yazma işleminden önce silme işlemi zorunluluğu ve silinebilir her bir ünitenin azami silinme kısıtlaması gibi. Bellek Dönüştürme Katmanı, tam bu noktada dezavantajlara cevap olarak karşımıza çıkmaktadır. Bellek Dönüştürme Katmanı; anlık belleklerde işletim sistemi ile bellek arasında köprü işlevi sağlayan bir yazılımdır. Belleğin doğru bir şekilde uzun süre kullanımı bu dezavantajların bilinmesi ile mümkündür. Bellek Dönüştürme Katmanı, anlık belleklerin doğasından gelen bu dezavantajları azaltmak, işletim sistemi ve bellek arasındaki adresleme işlemlerini yapmak, belleğin kullanım ömrünü mümkün olduğu kadar uzatmak ve hata bulma/düzeltme işlemlerini gerçekleştirmek için tasarlanmıştır. Bu çalışmada mevcut Bellek Dönüştürme Katmanları avantaj ve dezavantajları ile incelenmiş ve NAND bellek üzerinde çalışacak bir Bellek Dönüştürme Katmanı tasarım, gerçekleştirme ve simülasyonu sunulmuştur.

Anahtar Kelimeler: Anlık Bellek, NAND, NOR, Adresleme, Anlık Bellek Dönüştürme Katmanı

To my lovely wife, my parents and my nephews Emre and Buğra

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Attila Özgit for his guidance in this thesis. Without his endless patience and support I would not finish this work. I am feeling lucky to share his vision and knowledge throughout this thesis.

I would like to thank my father for all he has done and he will be done for me. He is the person, who encourages me not to stop learning and educating myself anytime. He is the second head teacher for me after Mustafa Kemal ATATÜRK.

I deeply wish to express my lovely feelings to my wife, who supports me in every way. She always gives me the power to work on my thesis.

I am also grateful to my parents, my sister, my brothers and my lovely nephews for their support. My brothers have always been by my side whenever I needed.

I would like to thank my everlasting friend İsmet Yalabık for his encourage through the work.

I also wish to express my gratitude to my commanders, especially to Capt.Hasan Çifci for their support to my Graduate Education.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Flash Memory is non-volatile computer memory that can be electrically erased and reprogrammed [1]. Flash Memory technology has been widely used as a storage media in electronic devices such as Mp3 players, mobile phones, flash drives, cameras. Flash Memory owes its reputation to its low power consumption, high shock resistance, non-volatility and its mobility [2, 3]. Since Flash Memory has no mechanical moving parts like a classical hard disk, it consumes low power and is high shock resistance. Flash Memory is a complete electronically device. Flash memory is non-volatile because of its memory type; it can keep the stored data even if the power goes off. Also Flash Memory devices are durable against temperature and humidity.

Flash devices are widely used in daily life. People don't use any more roll of films, they just shot hundreds of pictures easily without using any rolls. The classical floppy drives are no more needed in a contemporary computer. CD/DVD drives are also not much used like before. USB Flash drives have already dominated the market in terms of data storage. Recently, many hard disk vendors try to replace classical storage hard disks with Solid State Drives(SSD), a type of Flash Memory. The Solid State Drives are not like a ram drive or like USB Flash Memory, but it is like a ram disk. Companies are announcing portable computers with 512 GB SSD disk capacity [4].

On the other hand, Flash Memories are not miracle devices. They have also some disadvantages. For example, a written area in Flash Memory cannot be updated before it is erased. If a place is to be written, it has to be empty. If it is not empty, there has to be an erase operation in advance.

Another disadvantage is that each erasable unit in a Flash Memory has a maximum erase limitation, meaning that any erasable unit can be erased at most up to a certain number of times. These disadvantages can be seen as the limitation of the Flash Memory. It is important to design and produce Flash Memory architectures in consideration of these limitations. If these limitations are not known well, any design about Flash Memory will eventually fail.

At this point, Flash Translation Layer appears. Flash Translation Layer tries to overcome these limitations of Flash Memory. Flash Translation Layer (FTL) is a software module located in the Flash Memory that acts like a bridge between operating system and the Flash Memory. The operating system sends command to the Flash Memory in order to read some place or to write something on the memory. FTL directs the operating system commands and manages the internal structure of the Flash Memory.

Flash Translation Layer has some other important jobs in order to manage the Flash Memory.

Firstly, Flash Translation Layer manages the address translation between the operating system and Flash Memory. The operating system sends its commands with logical addresses. These logical addresses have to be translated into physical addresses. Moreover, extra data structure are needed for these operations. This translation mechanism and the extra necessary data structure build together the address translation in Flash Translation Layer.

Secondly, in a Flash Memory, there may arise some bad blocks from production or from usage. These bad blocks cannot be read, written or erased. Flash Translation Layer has to manage bad blocks in the Flash Memory.

Another duty of Flash Translation Layer is the power-off recovery. Whenever a power-off in the Flash Memory occurs, Flash Translation Layer has to handle the situation and maintain its state.

Wear leveling is another job of Flash Translation Layer. Each erasable unit in a Flash Memory has maximum erase limitations, in other words, has its own life cycle. After fixed number of erase operations that unit has to be marked as bad because of the physical properties of Memory cells. Therefore, Flash Translation Layer tries to arrange the Flash Memory in such a way that, every erasable unit wears off approximately around the same time.

Finally, Flash Translation Layer does the error detection and correction in Flash Memory.

Flash Translation Layer uses algorithms and extra data structures for conducting this duties.

There are various Flash Translation Layer algorithms in the literature. Some of them is designed to have a more read throughput from memory. For some of them, the write throughput of the Flash Memory is important. For some, it is important to have a minimum space for FTL algorithm itself. As a result, FTL algorithms could change Flash Memory performance and characteristics drastically.

The FTL algorithms in market are all closed source and proprietary, none of them is available for an open-source development. Once the FTL algorithm is build, then it is put on the chip in the Flash Memory. After this point, the algorithm cannot be changed. If the algorithm offers a fast read-throughput, it cannot be changed in order to have a faster write-throughput. In case a more sophisticated error detection and correction algorithm is desired, FTL would not perform it, because it is unable to change the error detection and correction algorithm in it. You cannot increase or decrease the size of the meta data used in FTL according to your needs.

This thesis tries to design and simulate an own FTL algorithm. The algorithm of the design will be an open source algorithm and design. The algorithm could be changed according to expectations from the Flash Memory. This brings great flexibility. If Flash Memory has to provide a faster write throughput, it could be designed in that way. Also, the error detection and correction algorithm could be designed according to the needs. The size of the data structure used in Flash Translation Layer algorithm could be adjusted. In other words, there will be full control over the algorithm.

1.1 Thesis Outline

The thesis is organized as follows: In Chapter 2, some basic definitions and background of Flash Memory are provided. Then Flash Translation Layer and its main capabilities are given. At the end of the chapter detailed information about existing translation layer algorithms is given. In Chapter 3, the thesis approach to Flash Translation Layer is proposed, its design and implementation details are given. Chapter 4 comes up with experimental results of our implementation. Some comparisons with existing Flash Translation Layer algorithms are stated. The last chapter concludes the thesis and tries to give some idea about the future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

Flash Translation Layer has a vital role in a Flash Memory. Address translation between operating system and Flash Memory, wear leveling, bad block management, error detection and correction and power-off recovery are all done by Flash Translation Layer. All these algorithms individually effect the performance of Flash Memory. There are various Flash Translation Layers in terms of their different approaches. A Flash Translation Layer may have better wear leveling algorithm compared to the other, on the other hand it may have a simple healthy power-off recovery. Most of the Flash Translation Layers, discussed in this study, differ from each other in terms of their address translation designs. Replacement Scheme, BAST, FAST, SUPERBLOCK and LAST are several of them. Address translation plays a big role in the performance of the Flash Translation Layer, since it effects all the other algorithms in the Flash Memory as well.

To better understand our approach, first we will look at some basic definitions. Then we will focus on the Flash Memory. After that, Flash Translation Layer and its duties in a Flash Memory will be discussed. Finally, in the related work section, Flash Translation Algorithms Replacement Scheme, BAST, FAST, SUPERBLOCK and LAST will be examined. Their address translation schemes, advantages and disadvantages will be discussed.

2.1 Definitions

With further definitions, some basic information about Flash Memory will be given.

Definition 1.

A page is the smallest unit which a Flash Translation Layer can read or write.

Definition 2.

A block is composed of a number of pages and is the unit of the erase operation in Flash Translation Layer.

Definition 3.

There are two types of Flash Memory, NOR and NAND types. In a NOR type Flash Memory, the erase and write times are longer, on the other hand random access to any memory location is provided. In a NAND type Flash Memory, erase and write times are shorter when compared with NOR type. NAND type Flash Memories require smaller chip area per cell, in other words the cost per bit decreases. Besides a NAND Flash Memory is much more durable than a NOR type Flash Memory. Most of the Flash Memory vendors are using NAND type Flash Memories in their products because they need greater storage densities with lower cost [5, 6].

From now on, it will be considered only NAND Flash Memory in the thesis.

2.2 Flash Memory

Flash Memory is a device with low power consumption, high shock resistance, non-volatility and mobility. With all these advantages, Flash Memory is slowly taking the place of secondary storage media. Besides all its advantages, Flash Memory devices have some disadvantages. One of them is that previously written data cannot be overwritten on its original place. The updated data has to be written to another suitable free location, meanwhile a mapping between the original data and the updated data has to be kept in order not to lose the updated data.

Where up-to-date data exists, consequently there will be garbage. There should be a garbage collector mechanism to clean up the Flash Memory. Another disadvantage of Flash Memory is that every block in a Flash Memory has limited erase lifetime [7].

Flash Memory is a Electronically Erasable Programmable Read Only Memory(EEPROM). There is a couple of transistors at each intersection in the cells and an oxide layer between these transistor. To change the value of cells from 0 to 1 or from 1 to 0, Fowler-Nordheim Tunneling is used [8].

A NAND Flash Memory consists of a fixed number blocks. To give an example, each of the block consists of a number of pages, typically 32 pages, where each page is 528 bytes. Each page is composed of 512 bytes of data area and 16 bytes spare area. This spare area is used for internal management and for error correction. A sample table of the data types is given in table 2.1

1 Block	32 Pages
1 Page	528 Bytes (512 Data + 16 Spare)

Table 2.1: Flash Memory - Data Types

In Flash Memory devices, read and write operations are done in page basis whereas erase operations take place in block basis. Write operations take much longer time than a read or erase operation. An erase operation means doing all the bits 1. Considering a block consists of several number of pages, an erase operation happens to be very expensive. Once an overwrite operation initiates on the data, a long time erase operation must take place.

Because of the nature of NAND flash memories, those erase operations, indexes and garbage collector process decrease the overall performance. In order to solve these problems, software called Flash Translation Layer (FTL) is designed.

2.3 Flash Translation Layer

Flash Translation Layer is a software module between the operating system and the Flash Memory as in figure 2.1. On top of the operating system, applications want to reach to the Flash Memory, either read, write or erase commands. Operating system converts application

requests to the appropriate commands by using its file system.

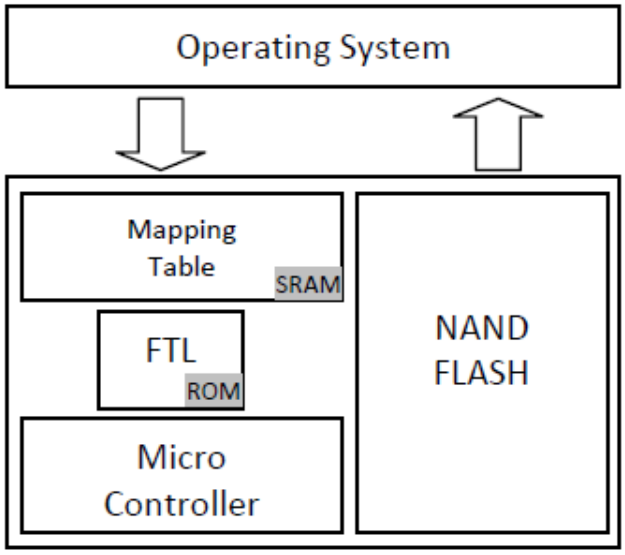

Figure 2.1: The Architecture of Flash Memory

Flash Translation Layer takes these commands converts into the corresponding commands in Flash Memory chips. Flash Memory keeps the volatile address mapping information in the Mapping Table. This mapping array is SRAM and volatile. When power goes off, the mapping table vanishes. However the FTL code is in the ROM area of Flash Memory and it is non-volatile. The micro controller of the Flash Memory executes the FTL code and calculates the corresponding addresses according to the operating system commands. These commands could be read, write or erase. FTL redirects each request to the corresponding place in NAND Flash Memory. If the command is a write command, then it has to find an empty location for this command. Also FTL manages an internal mapping table to record the mapping information from the logical sector number to the physical location [9, 10].

Bad block management, wear leveling, power off recovery in Flash Memory are also done by Flash Translation Layer.

Flash Translation Layer can detect the bad blocks, which have been generated from the beginning in the factory and during the lifetime of Flash Memory usage. Flash Translation Layer marks those bad blocks and will not use them anymore.

In Flash Memory, each block can be erased up to fixed number of times. This number can

differ from vendor to vendor. If a blocks erase count reaches this specific limit, then further erases of this block may corrupt this block with a great probability. Flash Translation Layer tries arrange write and erases, so they distribute evenly on Flash Memory.

When suddenly a power-off occurs, Flash Memory has to maintain internal data consistency and integrity.

2.3.1 Wear Leveling

In a Flash Memory, each block can be erased up to fixed number of times because of the nature of Flash Memory. This number may vary from vendor to vendor or from single-level Flash Memory to multi-level Flash Memory. To give an example from the market, this fixed number may differ from 10,000 to 100,000. The best way to have a long term usage with Flash Memory device is to keep the erase count of blocks close to each others. Without Flash Translation Layer, repeatedly updates to a logical location in Flash Memory yield to repeatedly updates to the same physical block in the Flash Memory. Eventually, the repeatedly updated blocks wear off in a short time period. A flash device with 10,000 maximum erase count can be used for 27 years with completely write and erase the entire content once per day [11] if Flash Translation Layer is implemented.

There is a sample case showing the difference between lifetime with or without wear leveling [6]. The case simulates frequent FAT table updates. Without wear leveling, the expected lifetime of Flash Memory is 0.55 days. However with wear leveling, the expected lifetime reaches up to 49.7 years depicted in table 2.2.

Wear Leveling	Lifetime
ON	49.7 years
OFF	0.55 days

Table 2.2: Lifetime with or without Wear Leveling

Wear leveling mechanism in Flash Translation Layer tries to arrange write and erases, so they distribute evenly on Flash Memory for a long-term usage. In other words, the wear leveling ensures no block is erased too much more from the others.

2.3.2 Bad Block Management

Bad blocks are the nature of Flash Memory. There are several situations to have bad block in Flash Memory. Even a new brand Flash Memory may have bad blocks except its first block.

Moreover after using the Flash Memory, there may occur some bad blocks. There is a status register that shows the completion of an operation. By inspecting this value, Flash Memory can decide, whether a block is bad or not.

Also every block has a maximum number of erase value. After this value is exceeded, the block is marked again as bad. A bad block cannot be written, read or erased, therefore bad blocks have to be listed somehow. There must be a bad block management that keeps track of bad blocks [12].

During the production of a Flash Memory, the factory level bad blocks are marked. The bad block's specific pages are marked with *FFh* value. With this information in mind, FTL can scan all blocks and examine whether they are bad or not. The sample algorithm is shown in figure 2.2. From the first to the last, all blocks are examined and the bad blocks are listed.

If a bad block arises, it has to be handled. Another healthy block has to be allocated for that block. There are two ways of handling bad blocks. One is the *Skip Block Method* and the other is *Reserve Block Method* [13].

In the Skip Block Method, when a block gets bad, the data will be written to the next good block. This change has to be stored somewhere as meta data.

In Reserve Block Method, when a block gets bad, it is forwarded to a preallocated healthy block. This preallocated blocks are invisible to the user and are not counted as the total storage of Flash Memory. This structure is shown in figure 2.3 [13].

2.3.3 Address Translation

A flash device has the erase-before-write constraint [14]. When new data has arrived, Flash Translation Layer(FTL) has to find an empty location for it. If there is no empty space left, FTL calls garbage collector and erases some blocks. Meanwhile some blocks are erased, some of them are written and some of them are moved to other places. FTL has to keep the

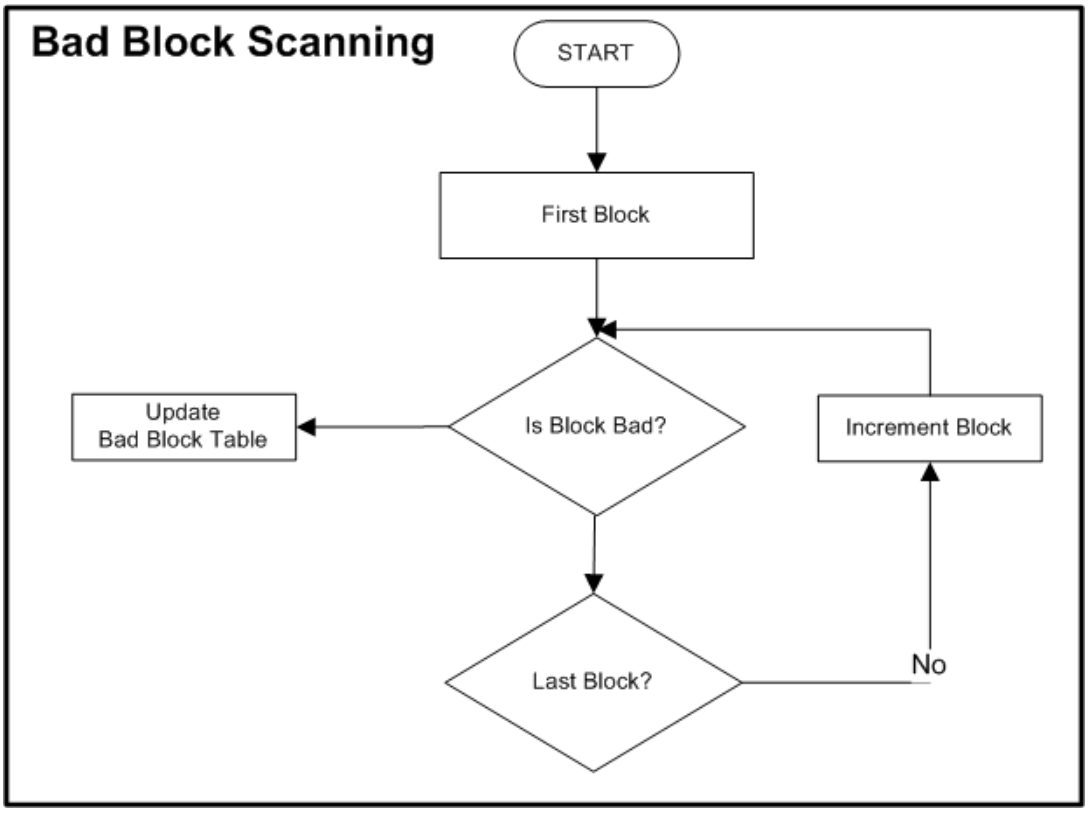

Figure 2.2: Bad Block Scanning

Figure 2.3: Preallocated Blocks

address information of all this actions in memory correctly. Therefore address translation and address mapping are very important for a FTL design.

There are three kinds of address mapping in Flash Translation Layer; **page-level, block-level and hybrid-level** address mappings.

2.3.3.1 Page Level Address Mapping

In page-level address mapping, any logical page from the operating system point of view can be mapped to any physical page in Flash Memory. This mapping scheme is simple and fast. In a page level address mapping, the mapping table will be quite big. On the other hand, the garbage collection mechanism will work efficiently [15].

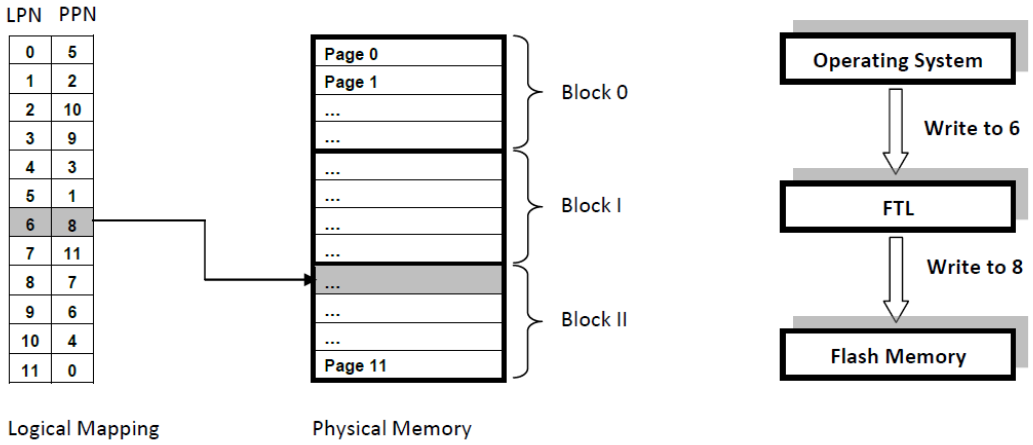

Figure 2.4: Page Level Mapping

In figure 2.4, operating system makes a write request to logical page 6. In this example the logical page 6 is mapped to physical page 8 as in the mapping table. There is no need to store the block of the logical page or physical page.

2.3.3.2 Block Level Address Mapping

In block-level mapping, a logical block can be mapped to a physical block. This approach comes with a small mapping table, but a more sophisticated garbage collection mechanism has to be designed.

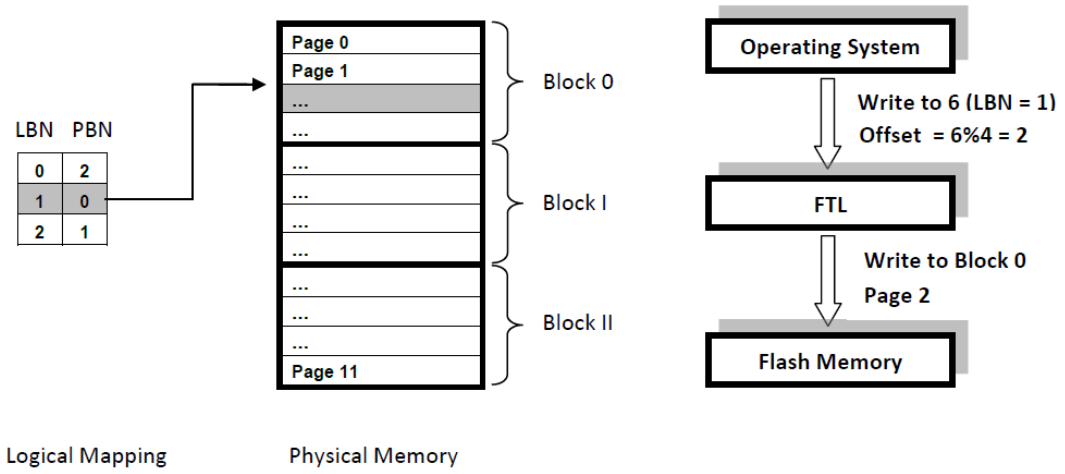

Figure 2.5: Block Level Mapping

In figure 2.5, operating system makes a write request to logical page 6. Logical block number of logical page 6 is 1. From the mapping table, the corresponding physical page 0 is found. 4 is the number of pages in a block. From 6%4, the page offset 2 is found. The data is written to the page 2 of block 0. If a write request to the same logical page occurs, the scheme allocates another block for that single page. In this scheme, the size of the mapping scheme is small compared to the others.

2.3.3.3 Hybrid Level Address Mapping

Efficient FTL schemes try to come up with some kind of hybrid-level address mapping in order to catch a suitable point in the trade-off between the mapping table size, garbage collection and the complexity. Hybrid-level address mapping tries to overcome the disadvantages of page-level and block-level address mapping schemes. In hybrid-level address mapping, data block's mapping is done via block-level scheme, whereas log block mapping is conducted via page-level scheme [16].

In figure 2.6, operating system makes a write request to logical page 6. Again the logical block number and the physical block number are found from the logical page number. After that, data is written to the page in the physical block.

Flash Translation Layer schemes that use data block and log block for normal and updated

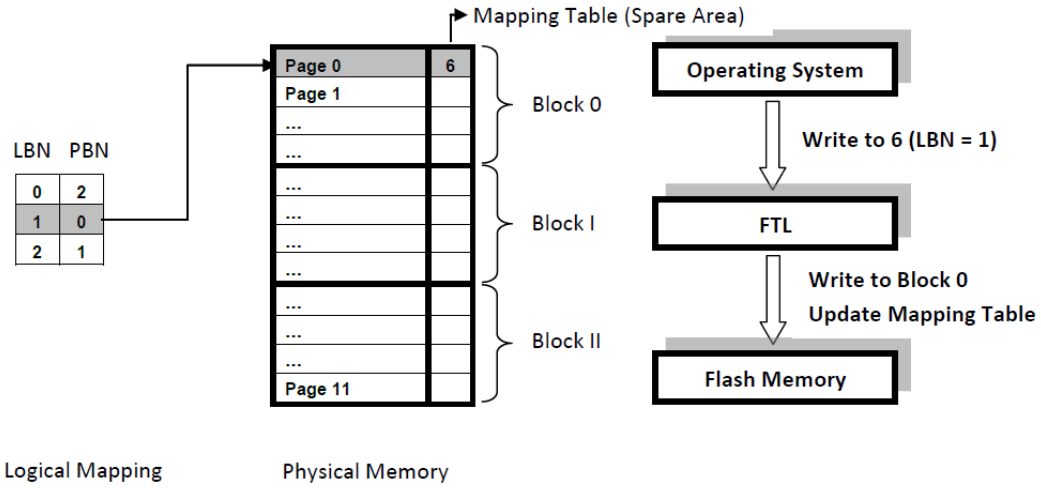

Figure 2.6: Hybrid Level Mapping

data, are called *Log Buffer-Based FTL* schemes. These Log Buffer-Based FTL schemes use Hybrid Level Address Mapping in address translation.

2.3.4 Merge Operations

In Log Buffer-Based FTL schemes, after some time there will be no suitable position in log blocks. Further page updates will result to call the garbage collector. With garbage collector, the log blocks are merged with corresponding data blocks. This merge operation plays a vital role in FTL algorithm's efficiency. There are three types of merge operations.

2.3.4.1 Switch Merge

The switch merge is the cheapest merge operation. The data block with old data is erased and the log block with the updated data gets the new data block. But there is a restriction for switch merge, the pages in the log block have to be sequential. The switch merge is shown in figure 2.7.

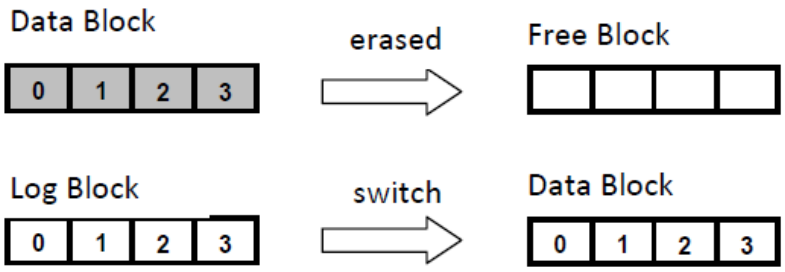

Figure 2.7: The Switch Merge

2.3.4.2 Partial Merge

In partial merge, only a part of the data block is updated and updates are sequential. These updates yield also sequential writes to the log block. By merge operation, the valid pages from data block will written to the corresponding empty pages of log block. Then, data block will turn into a free block. The log block will be data block with all its pages valid. The partial merge is shown in figure 2.8.

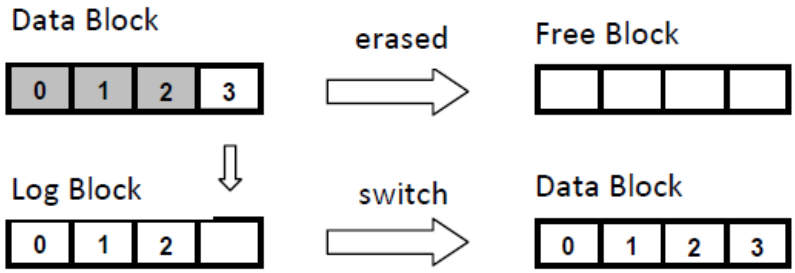

Figure 2.8: The Partial Merge

2.3.4.3 Full Merge

Full merge occurs when pages are often randomly updated. The clean-up the structure is a bit hard compared to other merge types. Firstly, a new empty block has to be allocated. Then, the valid pages from data block and the corresponding valid pages from log blocks are combined in the new block. The full merge is shown in figure 2.9.

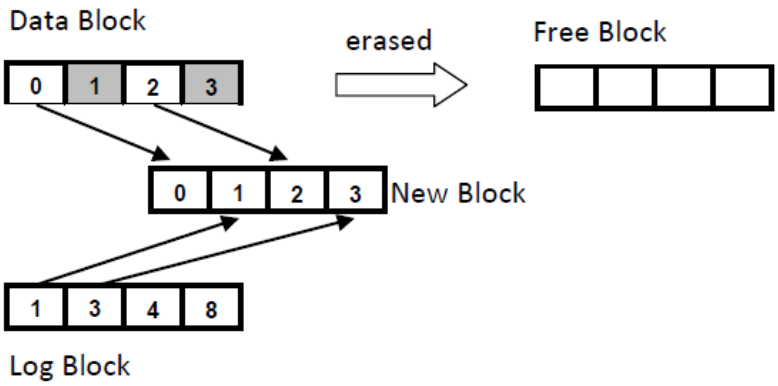

Figure 2.9: The Full Merge

2.3.5 In-Place vs. Out-Place

The smallest write unit in Flash Memory is the page. If a write request to the Flash Memory arrives, it is written either to a data block or to a log block. It is very important, where FTL puts the updated page. There is a block number and a offset number associated to every updated page. By looking these numbers, FTL finds the corresponding place and does the desired command.

There are two different schemes when considering an update operation of a page.

In *In-Place Scheme*, the offset number of the page never changes. Even if the page has to be written to another block, the offset number is preserved. A sample diagram is shown in figure 2.10.

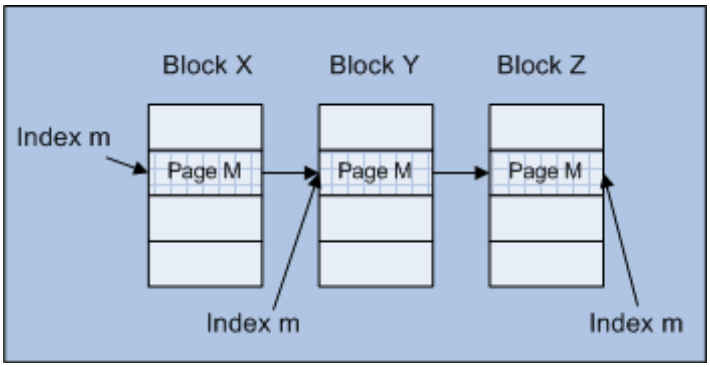

Figure 2.10: In-Place Scheme

In *Out-Place Scheme*, the offset number of the page may change. It has not to be preserved like the *In-Place Scheme*. In order not to lose the pages, a mapping table has to be kept. A sample diagram is shown in figure 2.11.

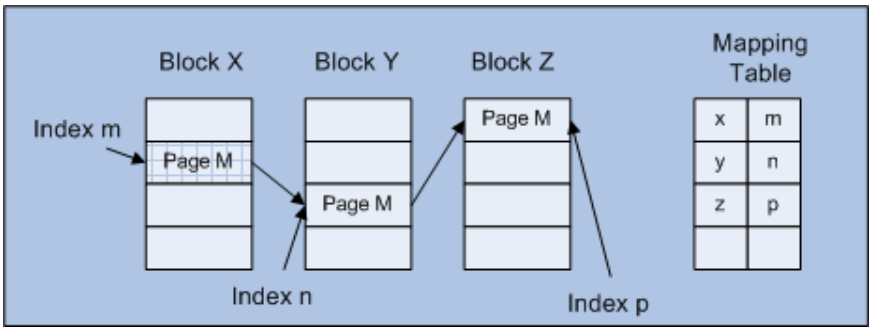

Figure 2.11: Out-Place Scheme

2.4 Related Work

There are various Flash Translation Layers according to their address translation, wear leveling, bad block management and error detection and correction algorithms. Most of the flash translation layers differ from each other in terms of the address translation algorithm, because address translation is the most effecting factor of a Flash Memory performance. The upcoming flash translation layer algorithms are all distinguished from each other in terms of their address translation.

2.4.1 Replacement Block FTL

A simple approach to FTL is Replacement Block FTL [17, 18, 19]. In this approach, there are two types of block; data block and replacement block. Data block simply stores the data and replacement block stores the updated data. P1, P2, P3 and P4 denote the pages. The pages with the 'X' sign are not valid any more. In this approach, a replacement block may have another replacement block for its updated pages.

In Replacement Block FTL, in-place scheme is used to store the pages, meaning that the offset number of pages are preserved, even they are written to a replacement block. Therefore, Replacement Block FTL suffers from updating the same page. While updating a page, it

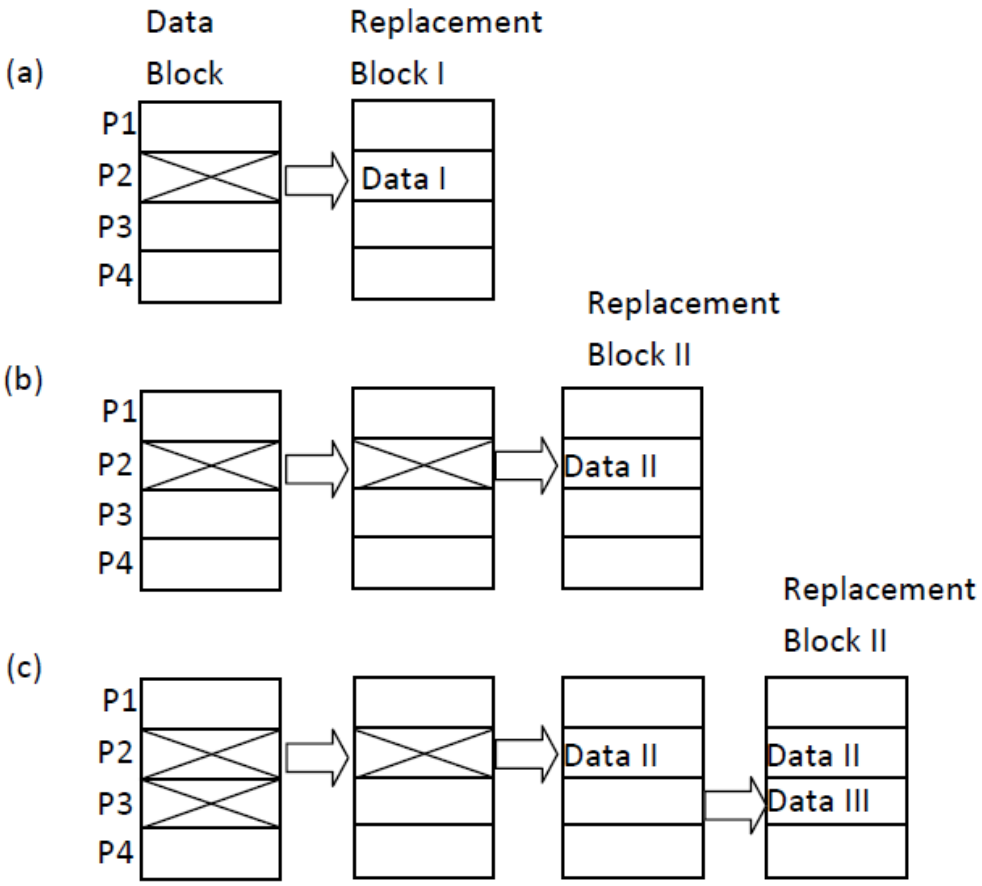

Figure 2.12: The Replacement Block FTL

allocates a replacement block for that page. When another update to the same page hits, it has to allocate another replacement block, while other pages of the previous replacement blocks are empty. The new replacement block will connect to the previous one. They behave like a link list structure. If there come new updates to the same page, the algorithm will allocate new replacement blocks until no free replacement blocks are left. If there are no replacement blocks in Flash Memory, the garbage collector runs and frees some of the blocks.

In figure 2.12, there is write request to Page 2. The Replacement Block FTL allocates a Replacement Block for that request. In figure 2.12.b, again there is write request to Page 2. Once more, FTL has to allocate another Replacement Block, because the update hits the same page. The second replacement block will become the secondary storage for the first one. Every write request makes FTL allocate a replacement block. Moreover, the other parts of the replacement blocks will be empty.

In figure 2.12.c, the write request hits another page. FTL does not have to allocate another new replacement block and it uses former replacement block.

2.4.2 Block Associative Sector Translation FTL - BAST

Block Associative Sector Translation FTL - BAST uses hybrid-level address mapping. The replacement block like structure is called as log block in BAST. A log block is dedicated to a data block in BAST. If a page in the data block is being updated, the updated page is written to the log block in sequential order independent from the offset number of the page. That is, BAST uses out-place scheme for page updates. In other words, the page offsets are not preserved. There has to be a structure to keep the page offsets.

Every update to the data block will consume space from the corresponding log block. If the corresponding log block gets full, the data block and corresponding the log block are merged together by the garbage collector. Sequential ordered write patterns make garbage collector run efficiently in BAST, whereas random ordered write patterns result bad performance.

If there are random ordered write requests, the probability to hit different blocks will gets higher. In BAST, when different blocks are updated often, log blocks have to be allocated more. After some point, garbage collector has to be called in order to free log blocks. However calling garbage collector before the log block is not fully filled, decreases the performance of

BAST. This is called the log block thrashing problem [1]. In log block thrashing, the space utilization gets worse and the probability of merge operations increases. This leads to a big number of erase, write and read operations for the Flash Memory and eventually decreases of the performance of Flash Memory.

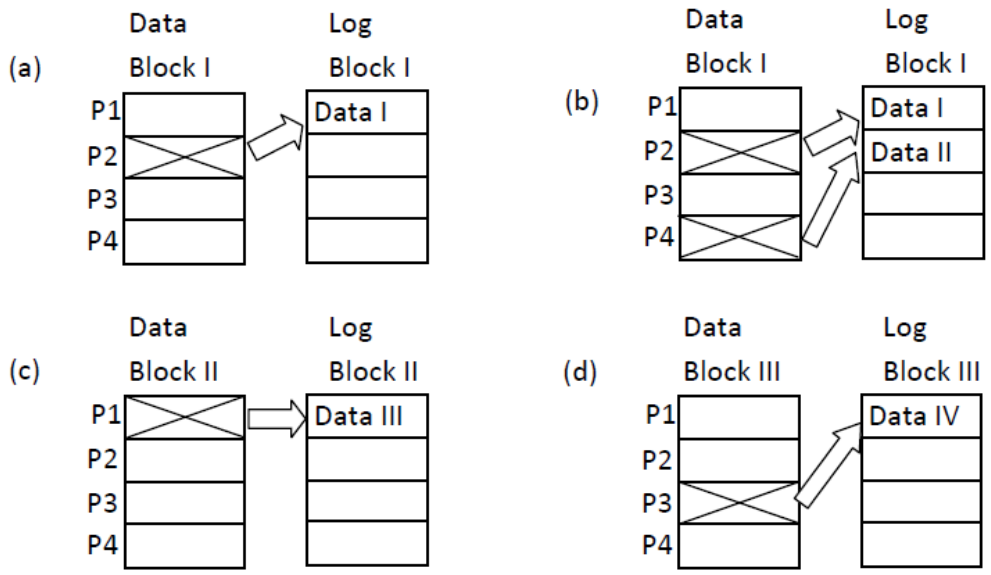

Figure 2.13: The Block Associative Sector Translation FTL

In figure 2.13, there is a log block dedicated to the data block. Whenever a page update to the data block hits, the update is written to the log block in sequential order. Again in figure 2.13.b, another update to the same data block may occur. BAST writes this update to the Log Block I. However, if updates hit other blocks, log block will be quite empty before garbage collector runs. As seen in figure 2.13.c and figure 2.13.d, update to the Data Block II and Data Block III result spending two more log blocks, although Log Block I is still half empty.

2.4.3 Fully Associative Sector Translation FTL - FAST

Fully Associative Sector Translation FTL - FAST [1, 12] uses also hybrid-level address mapping. Its main advantage is to overcome the disadvantage of BAST in log block thrashing problem. In FAST, a log block can be shared by many data blocks. Also FAST uses out-place scheme for pages updates like BAST. Any page from these data blocks can be mapped to any

page from the log block. This provides garbage collector run more efficiently compared to BAST.

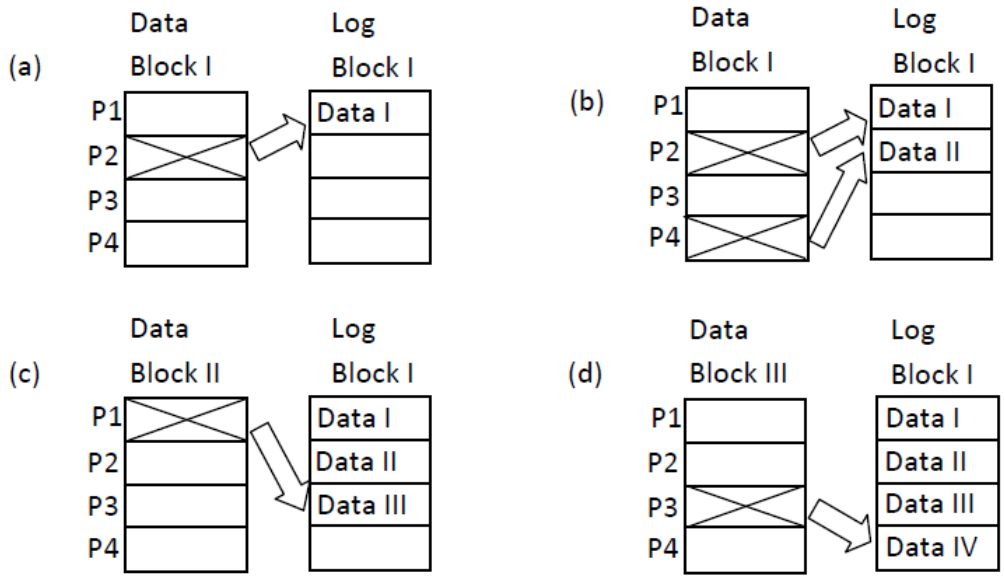

Figure 2.14: The Fully Associative Sector Translation FTL

In figure 2.14, updates to different data blocks are all written to the same log block. A bunch of data blocks use the same log block. Any update to these bunch of data blocks will eventually hit the same log block. This will increase the performance compared to BAST, when the write pattern is random ordered.

Also FAST has a separate log block for sequential writes. Sequential updates are all directed to the sequential log block in order to have a cheap switch merge at the end.

2.4.4 Locality-Aware Sector Translation FTL - LAST

In a Flash Translation Layer, the performance increases if the write pattern is sequentially ordered. If the sequential writes can be extracted from the input, then these sequential writes could be written to a place as a whole. The previous Flash Translation Layers cannot distinguish sequential writes and random writes. The workload in a daily usage of Flash Memory is a mixture of random and sequential writes. If there is a way to separate the sequential from the random ones, switch and partial merges can be applied [20] instead of full merge, which

is much slower than switch and partial merge.

LAST separates its log blocks into random log buffer and sequential log buffer. The locality detector decides which write goes to which buffer. In this scheme; sequential log blocks are related only to a data block that is block-level address mapping. On the other hand, random log blocks can be mapped to any data blocks because of the page-level address mapping. LAST also uses out-place scheme for page updates like BAST and FAST.

CHAPTER 3

THESIS WORK

In the thesis work, we aim to design and simulate our BAST like Flash Translation Layer. First, block management in the Flash Translation Layer will be discussed. Then with data structures, we will examine the internal design of our algorithm. After that, we will look at address translation mechanism. Related to address translation, some basic commands(READ and WRITE) will be explained. Then bad block management and wear leveling schemes will be discussed. Finally with garbage collector design, this part will come to end.

3.1 Block Management

The smallest erase unit in Flash Memory is the block. The wear leveling is about the erase count of individual blocks. Therefore, blocks play a vital role in Flash Memory. There are 3 types of blocks in this design, *data block*, *log block* and *reserve block* shown in table 3.1.

Data Blocks
Log Blocks
Reserve Blocks

Table 3.1: Block Types

The data block is used to store the data written by the operating system. The log blocks are used to store the data that tries to update the data blocks. The log blocks behave like a secondary memory for data blocks. The reserve block is used by merging the data block and the corresponding log block to each other. A sample diagram is shown in figure 3.1.

The number of data blocks, log blocks and reserve blocks are all determined at the beginning.

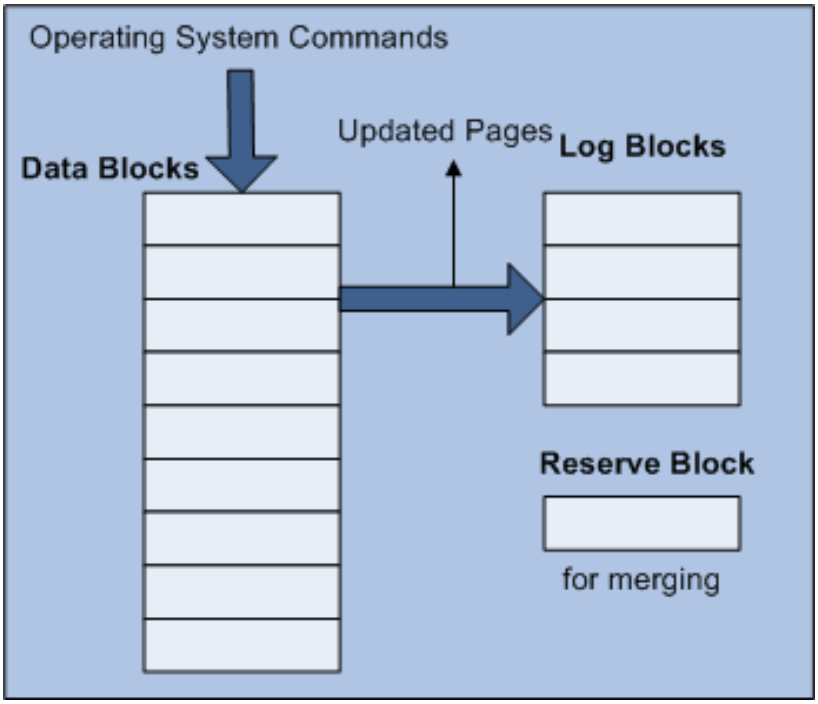

Figure 3.1: Block Overall Diagram

To give an example from real world, the log blocks could be %2 of data blocks. The number of log blocks cannot be too many, because every log block decrease the total storage capacity of the Flash Memory. Finally, there are some reserve blocks. There is only one reserve block in this design.

The operating system considers the Flash Memory as the total size of data blocks. That means, the log blocks and the reserve blocks are not counted as a storage in the Flash Memory. In the operating system side, the Flash Memory consists of logical blocks. The operating system does not know the inside mechanism of the Flash Memory, data blocks, log blocks or reserve blocks. In this design, every logical block by the operating system point of view is dedicated to only a single physical data block in the Flash memory. During the execution of FTL code, the matching of logical to physical blocks may change, but the one-to-one overlapping will be preserved.

Since the numbers of log blocks are small compared to the data blocks, a log block may serve to several data blocks. However, once a log block is used for a data block, the log block cannot be used for another data block until it is erased. That means, in a certain time, a log block is dedicated only to a single data block.

Flash Translation Layer software could change the type of a block in time, meaning a data block could be a log block, or the reserve block could be a log block during the execution of code.

3.2 Data Structures

Data structures in an algorithm play a vital role. If data structures are well designed, the implementation phase of the algorithm will be easier and quicker. Moreover, data structure heavily effect the performance of the algorithm. Well design data structures may decrease the total number of operations during the execution of Flash Memory.

Another important point is that, the size of the data structure matters in some cases. In a Flash Memory, where any single bit is important, the size of the data structure is vital. Therefore designing compact data structures may be important in some cases.

3.2.1 Data Structure : All_Blocks_Physical

In the design, there is the data structure named *All_Blocks_Physical*, which keeps all information about the blocks as shown in table 3.2. This data structure keeps *isBad*, *isWritten* and *type* char values, *eraseCount*, *logicalLocation*, *invalidPagesCount*, *logBlock*, *physicalBlock*, *updatedPagesCount* int values and *logPages* as an array of int values. *The All_Blocks_Physical* structure's size is the total number of blocks in the Flash Memory.

The *isBad* indicates whether the block is bad or not. This data structure area is a char value and is used like a boolean value. By default the char value is '0'. If the char value gets '1', then the block is bad. Once a block gets bad, it cannot be read or written any more, also the *isBad* never turns to value '0'.

The *isWritten* value also is a char value and is used like a boolean value. If a block is written, even one page, then this block is marked as written. By default the char value is '0'. If the block is written, then the char value gets '1'. During the execution of FTL code, a block's *isWritten* value may change many times.

The *type* of the block indicates, what type the block is. The block could have the type values

Name	Data Type
isBad	char
isWritten	char
type	char
eraseCount	int
logicalLocation	int
invalidPagesCount	int
logBlock	int
physicalBlock	int
updatedPagesCount	int
logPages[]	int

Table 3.2: Data Structure-All_Blocks_Physical

data, log or reserve. The type 'D' denotes the data block, the 'L' denotes the log block and the 'E' denotes the reserve block. During the execution these values may change according to algorithm.

The *eraseCount* number is an integer value and keeps the erase count of each block. Whenever the block is erased, its *eraseCount* value is incremented by one. This value cannot be changed in any way other than erase operation.

The *logicalLocation* indicates the logical block number of the physical block. The logical block number is the operating system point of view of the Flash Memory.

The *invalidPagesCount* structure is the type of int and keeps the invalid pages count in a block. Whenever a page inside a block is invalidated by a new write operation, the *invalidPagesCount* value is incremented by one.

The *logBlock* integer value keeps the address of the log block. The *logBlock* is applicable only for data blocks. For a log block or reserve block, the *logBlock* value is -1.

The *physicalBlock* integer value keeps the physical location of a log block. The *physicalBlock* is applicable for log blocks. For a data block or reserve block, the *physicalBlock* value is -1.

The *updatedPagesCount* integer value keeps the number of updated pages. The *updated-*

PagesCount is applicable for log blocks. For the reserve block, the *updatedPagesCount* value is -1. Whenever a page in the log block is used, the *updatedPagesCount* is incremented by one.

The *logPages[]* array structure keeps information about the pages in blocks. Its size is the *number of pages in block*. At the beginning all the *logPages[]* values are -1.

The pages in a data block are written to their corresponding places. That means, if there is a write request to block B with the Page number P, this page is written to Block B to Page P and the *logPages[P]* value gets 1. The value '1' indicates that, this page is a valid page and can be read later. If another write request to the same page hits, the new data will be written to corresponding log block. The *logPages[P]* value gets -2, meaning the page is no more valid, the valid data is in the corresponding log block's page.

The pages in a log block are written in sequential order instead. If a page-write request to a log block hits, the page is written in to next available page x and the written page offset is marked in the *logPages[x]*.

3.2.1.1 Size of Data Structure - All_Blocks_Physical

The data structure - All_Blocks_Physical is the most important data structure in the design. This structure is the dominant factor in the size of Flash Translation Layer.

A unit of All_Blocks_Physical's size is depicted in figure 3.2.

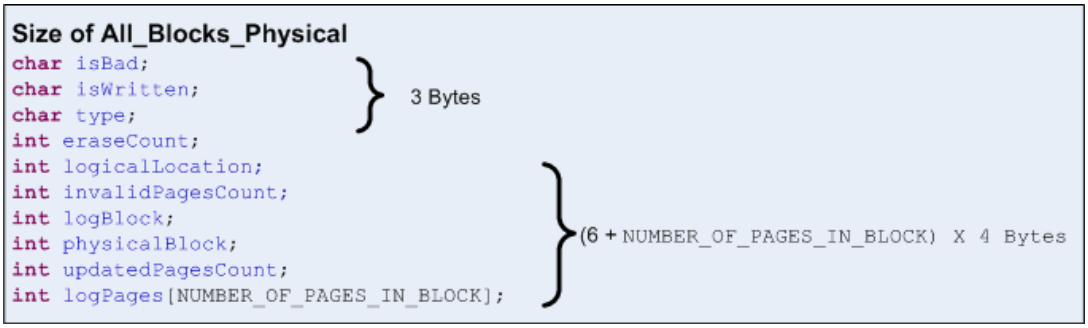

Figure 3.2: Size of All_Blocks_Physical

In equation below, the ratio of size of All_Blocks_Physical to the Flash Memory is given. When we calculate the equation according to the design, we found that the size of structure is around 1/128 of Flash Memory.

$$\frac{((6 + NumberOfPagesInBlock) * 4 + 3) * NumberOfBlocks}{NumberOfBlocks * NumberOfPagesInBlock * NumberOfBytesInPage}$$

3.2.2 Data Structure : All_Blocks_Logical

The *All_Blocks_Logical* structure shown in figure 3.3 simply shows which logical block maps to which physical block. Although this information can be obtained from *All_Blocks_Physical*, it is faster and simpler this way. The log blocks and reserve block are not visible by the operating system. In the operating system side, the Flash Memory size is the size of the data blocks. Therefore, the size of *All_Blocks_Logical* is the number of data blocks.

Figure 3.3: Data Structure - All_Blocks_Logical

3.2.3 Data Structure : FLASH_MEMORY

Every block consists of a fixed number of pages. From a wider point of view, the Flash Memory is composed of pages. Every page is composed of a fixed number of bytes. In this study, the byte structure is represented as an *unsigned char* structure in C, in other words the *FLASH_MEMORY* data structure is an array of unsigned char. Its size is *total number of blocks* times *number of pages in block* times *total bytes in a page* as shown in figure 3.4.

A page is composed of two sections; data and the spare area. To give an example, data section in this design is 512 bytes long and the spare area is 16 bytes long. A page in a typical Flash Memory is composed of 528 bytes as shown in figure 3.4. The spare area is used for storing some important data for meta data of Flash Translation Layer, such as error correction/detection. The size of data and spare area may change according to the FTL algorithm.

During usage of Flash Memory, Flash Translation Layer may invalidate pages. If a write-page

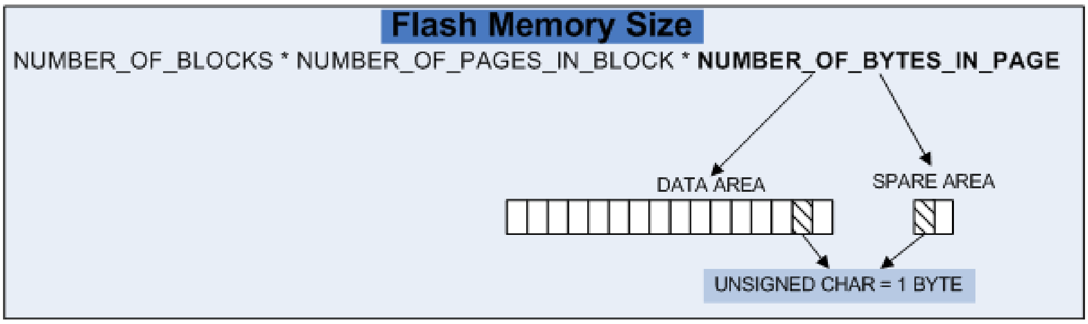

Figure 3.4: Flash Memory Size

request hits an already written page, the already written page must be invalidated. This is done by marking the *logPages[]* structure with -2. If a page has -2 in its corresponding *logPages[]*, then it can be derived that the updated page is in the log block.

3.3 Address Translation

The most important part of a Flash Translation Layer is Address Translation or Address Mapping. As mentioned before, there are three types of address mappings; Page Level Address Mapping, Block Level Address Mapping and Hybrid Level Address Mapping. In this study, we use and examine the *Block Level Address Mapping* in Address Translation.

The operating system sends commands like read or write. In this study, since we are designing and simulating a Flash Translation Layer, we simulate the operating system commands with an input file. A sample input string is shown in figure 3.5.

```
WRITE
89
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
```

Figure 3.5: An Example of The Input

The first line is the command itself. This may be *READ*, *WRITE* or *ERASE*. *READ* and *ERASE* commands are two-lined inputs, whereas *WRITE* command is three-line. The second line is the page number, when the command is *READ* or *WRITE*. When the command is *ERASE*,

then the second line indicates the desired block number. The last line is the simulated 512 bytes if the command is *WRITE*.

In address translation, the logical page number in the second line is first translated to logical block number and offset number. The address translation module uses this logical block number to look up in *All_Blocks_Logical* structure. This structure gives the physical block of the desired page. The offset number will remain the same. The flow is shown in figure 3.6.

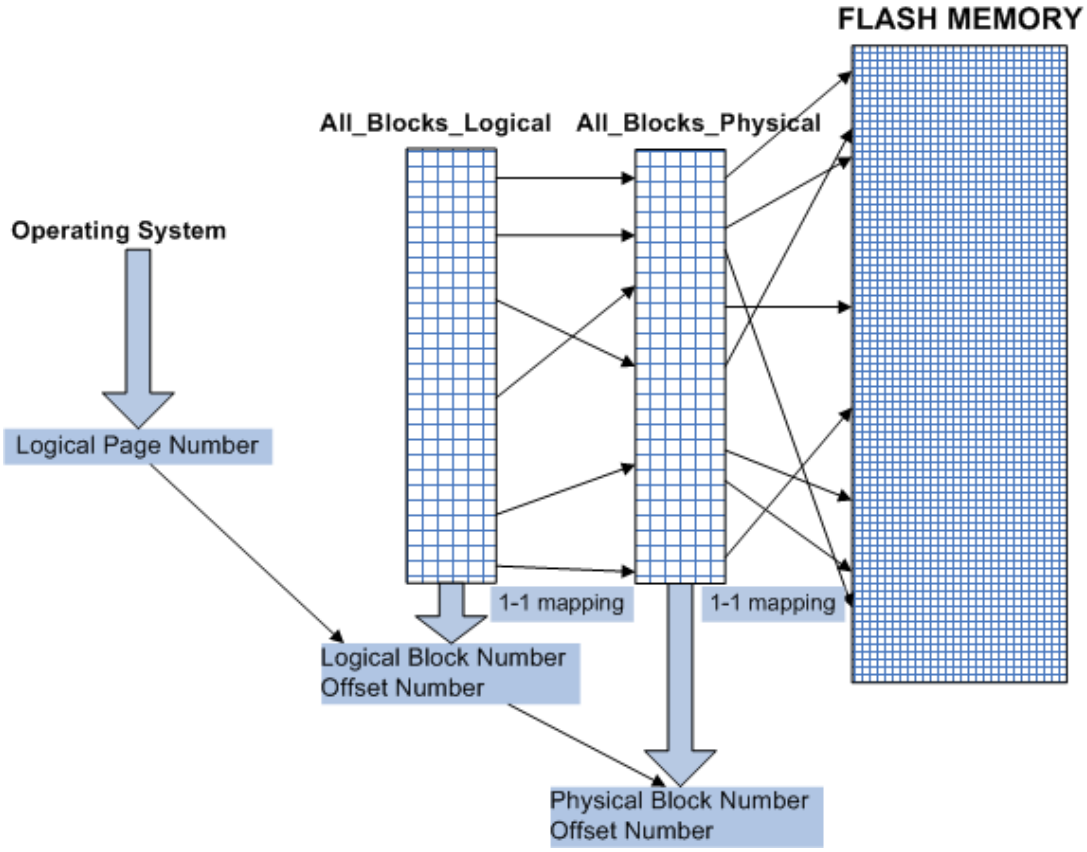

Figure 3.6: Address Translation

The logical block number can be found by dividing the page number by the page count per block as shown below. The offset number is the remainder of this operation.

$$LogicalBlockNumber = LogicalPageNumber/NumberOfPagesInBlock$$

$$OffsetNumber = LogicalPageNumber\%NumberOfPagesInBlock$$

3.4 Basic Commands

In the design, basic commands like read and write are simulated. There are two cases for read and three cases for write operation in the design. The cases and the corresponding algorithms are given below.

3.4.1 *READ* Command

Once the physical block number is obtained, FTL checks the *All_Blocks_Physical* with the given physical block number and the offset number.

If the offset number is equal to -1, then this page is empty. The page will not be read.

If the offset number is equal to 1, this page is valid and can be read. The location of the page is calculated with the given physical block number and the offset number, then the page will be read. The algorithm is shown in figure 3.7.

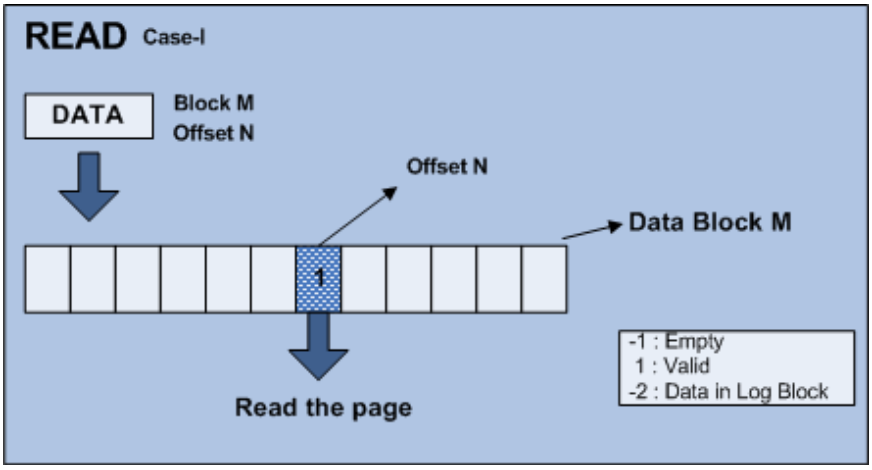

Figure 3.7: Read - Case I

If the offset number is equal to -2, then this page is not valid, it is updated before. To find the updated page, FTL looks to the *logBlock* value of the *All_Blocks_Physical*. This value shows the corresponding log block for the physical block. By design, every physical block is dedicated to a single block. All the pages inside this block will use the corresponding log block for updated pages. The corresponding log block is found, but the offset number in the log block cannot be found easily. The log block has to be searched for finding the most

updated data for the corresponding offset. Since the pages are written in sequential order, FTL starts to search the offset from the last position of pages in log block. The first page found with the corresponding offset number is the desired valid page. With this offset number, FTL calculates the actual address of the page and then reads the data 3.8.

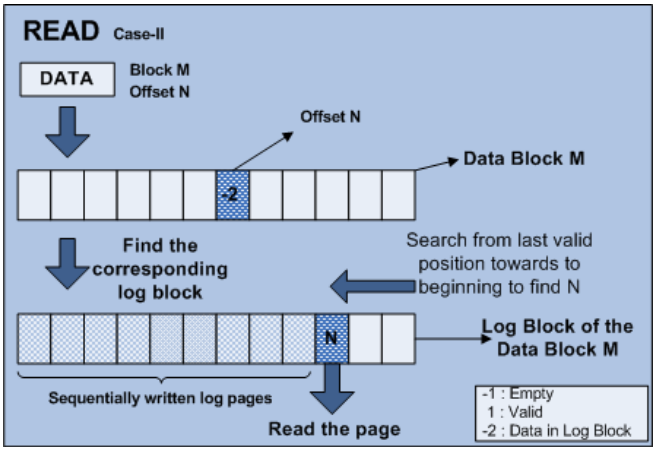

Figure 3.8: Read - Case II

The overall flow of the READ command is shown in figure 3.9.

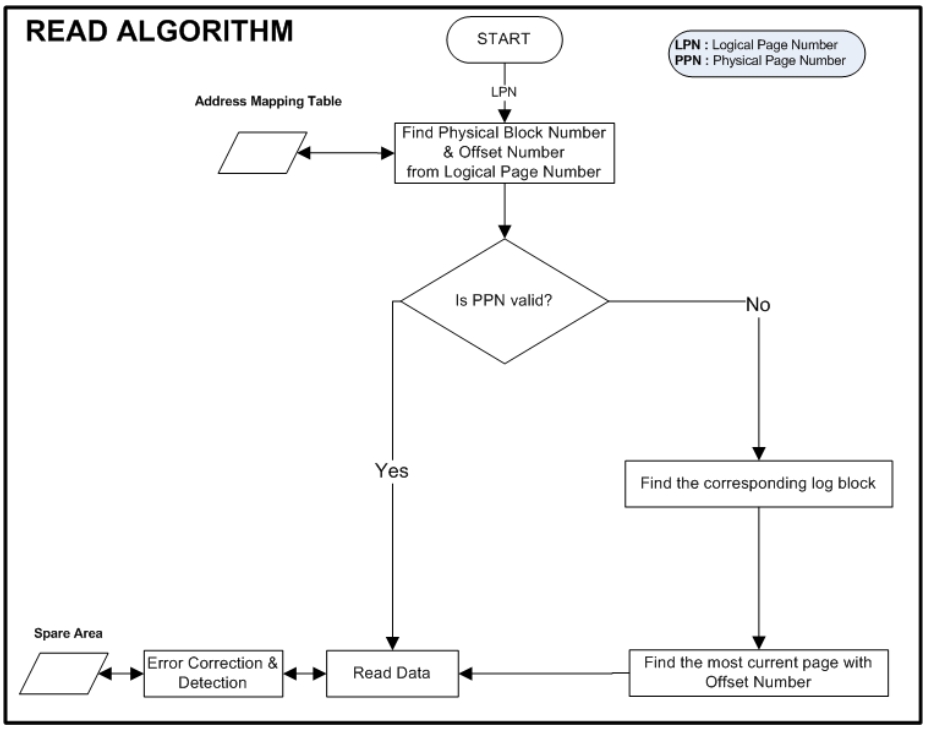

Figure 3.9: Read Algorithm Flow

3.4.2 *WRITE* Command

The first line is the *WRITE* command. The second line is the address of the page and the third line is the data to be written.

From the input, the physical block number and the offset number are obtained.

If the offset number in the corresponding *logPages[]* is equal to -1, then this page is empty. The write command can be easily issued to this page. After that, the place in *logPages[]* is marked with 1, meaning this page has valid data in it. The algorithm is shown in figure 3.10.

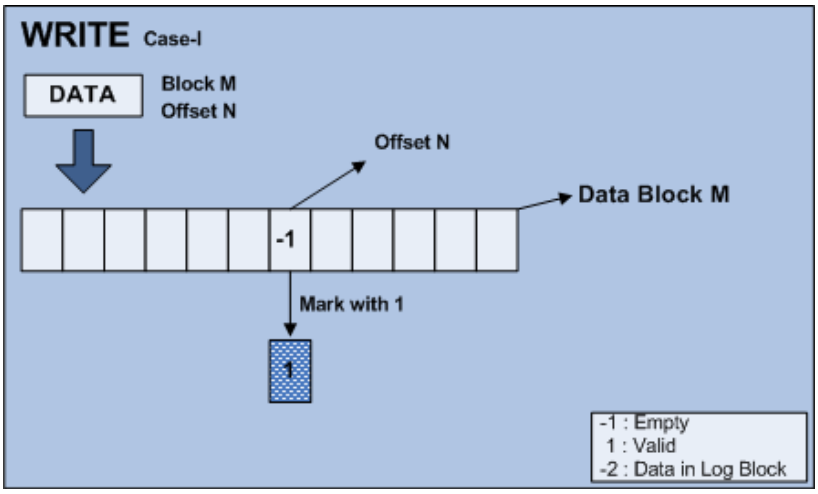

Figure 3.10: Write - Case I

If the offset number is equal to 1, then this page is a valid page. If there is an update for this page, this update will be done in a log block. FTL first gets the log block number of the physical block from the *All_Blocks_Physical*. Then it writes the data to the next empty location in this log block. After that, it writes the offset of the written page to the log block's corresponding *logPages[]*. Finally, FTL marks the physical block's corresponding *logPages[]* with -2, meaning this page is not valid any more, the updated data can be found in log block of this physical block. The algorithm is shown in figure 3.11.

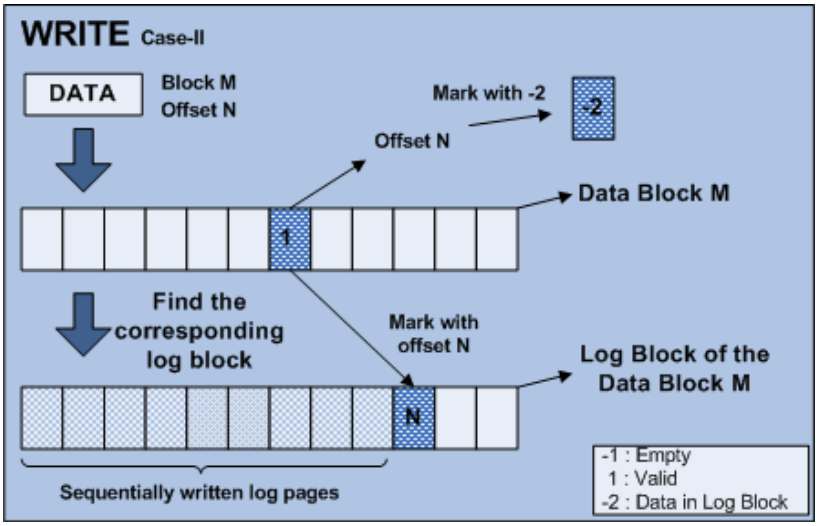

Figure 3.11: Write - Case II

If the offset number is equal to -2, the second algorithm above with offset -1 works fine, except the final marking step, because it is already marked with -2. The algorithm is shown in figure 3.12.

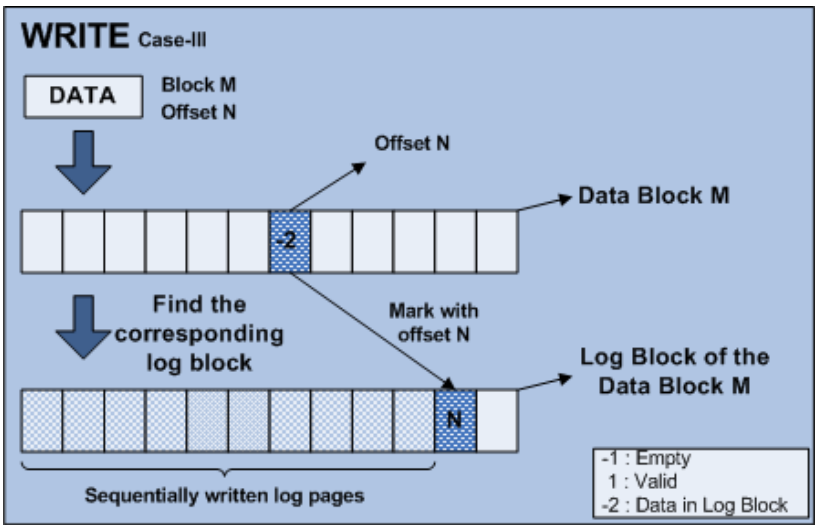

Figure 3.12: Write - Case III

The overall flow of the WRITE command is shown in figure 3.13.

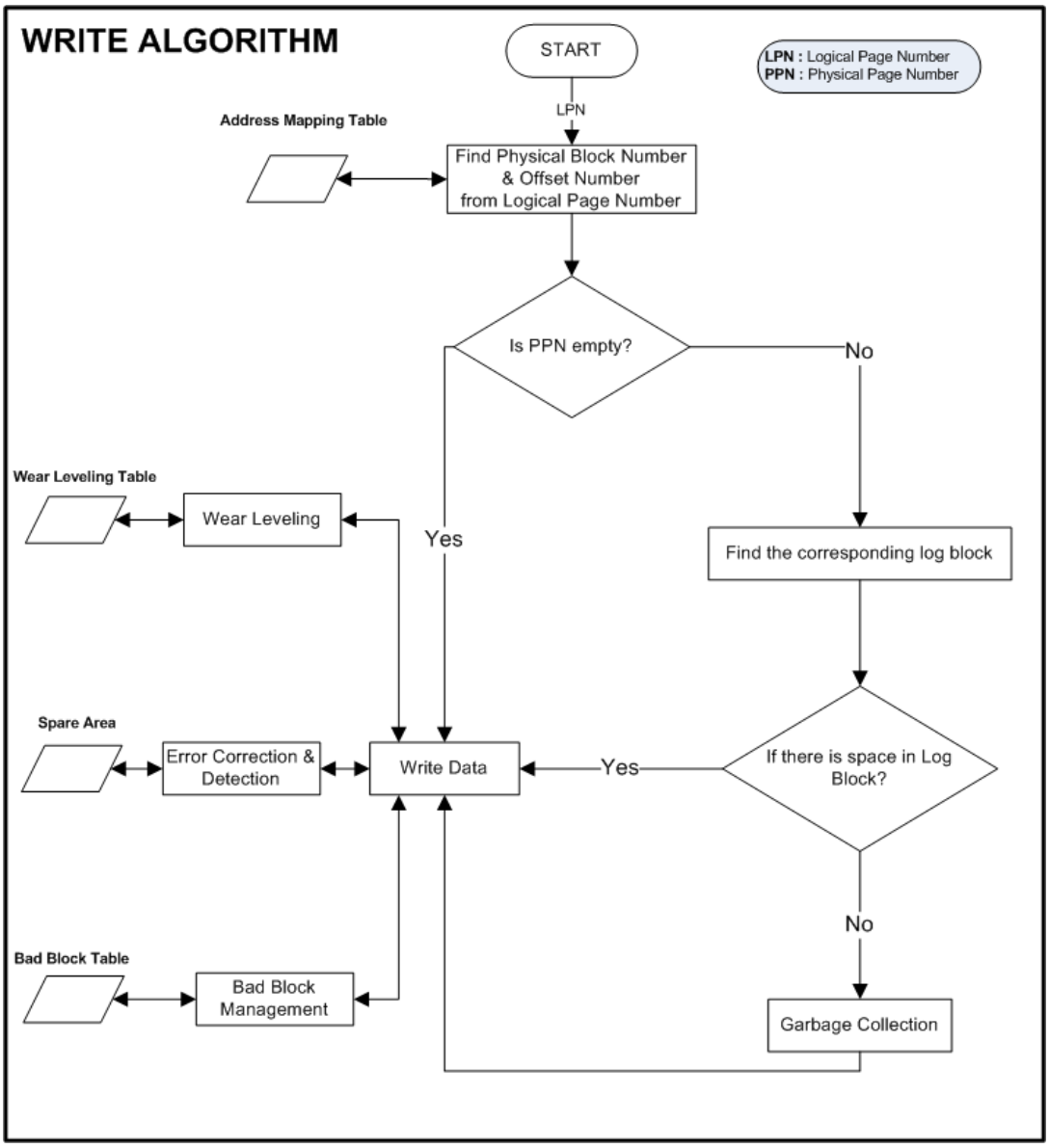

Figure 3.13: Write Algorithm Flow

3.4.3 *ERASE* Command

If the command is *ERASE*, then the second input line will be the logical block number. FTL finds the physical block number from the *All_Blocks_Logical* structure. Then it erases the physical block and its log block if exists.

3.5 Bad Block Management

The bad block management deals with the bad blocks in the Flash Memory. Simply a block may be either bad or healthy. Once a block is marked as bad, it cannot be read, written or erased again. The *isBad* area in *All_Blocks_Physical* keeps track of bad blocks.

As stated earlier, a bad block arises either from factory production or from the usage in time.

During factory production, there may be some bad blocks in a Flash Memory. It is acceptable as long as the bad block's percentage is below a fixed level. Another point is that, the factory has to ensure that the first block is healthy. Also during usage, some blocks may turn to bad. The job of bad block management in Flash Translation Layer is that to manage these bad blocks and direct all the read and write requests to correct locations.

There is a physical limitation that a block in Flash Memory could be erased up to a fixed number of times. To give an example from market, the blocks could be erased up to 10,000. If any block erased more than 10,000, then Bad Block Management Module marks it as bad. Being erased 10,000 times does not mean actually that the block is bad. But after this point, the probability of being bad increases. Therefore any block erased more than 10,000 marked as bad and will not be used any more for safety issues.

The *eraseCount* area in *All_Blocks_Physical* keeps the erase count of each block. By comparing this values with the global value *MAX_COUNT_OF_ERASE_VALUE*, the decision can be made.

When a block gets bad during the usage, the total storage decreases also, meaning the number of blocks addressable by FTL decreases.

In bad block management, implementing a bad block table may be useful. Once the Flash Memory is started, the algorithm scans all the blocks and check whether the block is bad or not. After that, it may write the bad block's information to a bad block table for an easier approach. Using a bad block table may increase the performance of the FTL.

3.6 Wear Leveling

As mentioned before, every block has a limited lifetime in terms of erase count. In order to have a wear leveling mechanism, the erase counts of the blocks are kept in *All_Blocks_Physical*. There is a global variable named *NEARITY_OF_ERASE_VALUE*. This constant indicates, how much more at most a block can be erased from other blocks.

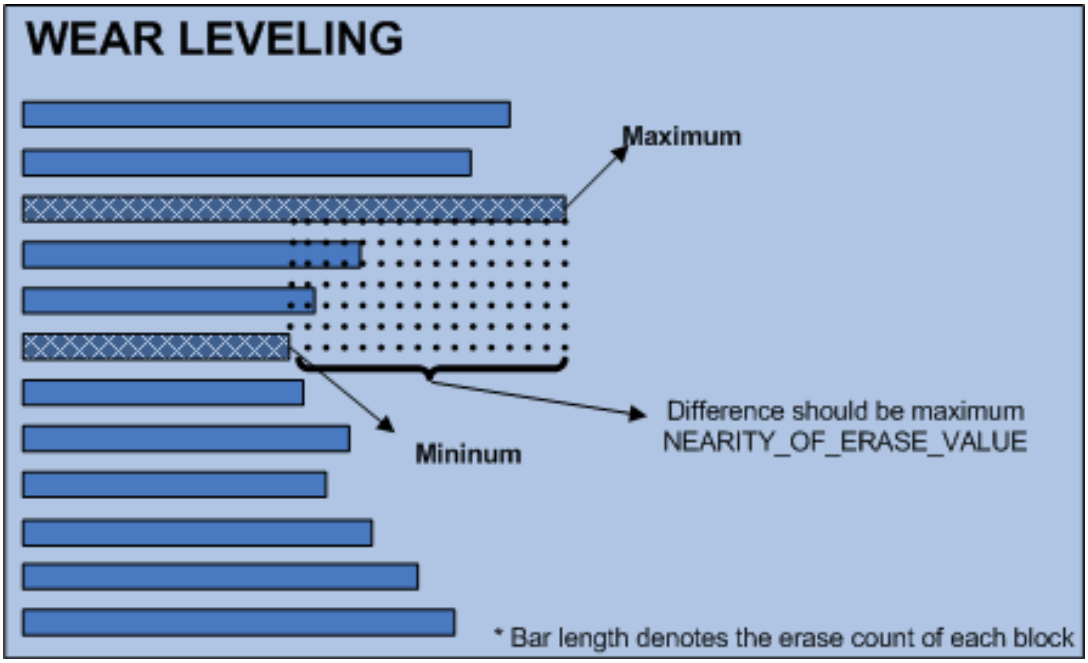

Figure 3.14: Wear Leveling

In figure 3.14, the longest bar denotes the most erased block. The least erased block is also shown. The first step is to find the bar pairs with the longest difference. FTL ensures that this length will never be longer than the global variable *NEARITY_OF_ERASE_VALUE*. This guarantees that no block will be erased too much compared to the others.

In Flash Memory, blocks are erased normally during the usage of Flash Memory. FTL also keeps a global variable named *Max_Count_Of_Block_Erases*, this shows the erase count of the most erased block. By having this value in mind, whenever a block is erased, FTL inspect the erase count of the block. If this value is not much far away from *Max_Count_Of_Block_Erases* (not much far means less than *NEARITY_OF_ERASE_VALUE*), then we can conclude that this block is used more than the others and it has to be replaced with another block with number of erase count more less. There are two ways for finding a less erased block.

First one is to find an empty and less erased block. This is straight forward, FTL finds the block from the empty blocks with less erase number, then does the exchange of blocks with it.

Second one is, if FTL could not find an empty and less erased place, it looks for used valid blocks. Because these blocks are already in use, they may have log blocks associated with them. Therefore, before the exchange mechanism, a garbage collector should work.

FTL Wear Leveling ensures an evenly distribution of blocks on point of view of their erase counts.

3.7 Garbage Collector

If FTL tries to update the pages in data block, it writes the updated pages to the log block. Once the log block is used, it cannot be used for another data block any more, until it is erased. This internal erase operation is the garbage collector.

Garbage Collector is called in two ways.

Firstly, during usage, all the pages in log block could get used. From this point, any update to this log block will call eventually garbage collector. This case is shown in figure 3.15. In this figure, not all of the pages in data block are occupied, but all of the pages in log block are either valid or invalid, meaning there is no empty page in log block. Therefore, any update to the data block, which is going to be placed in log block, calls Garbage Collector.

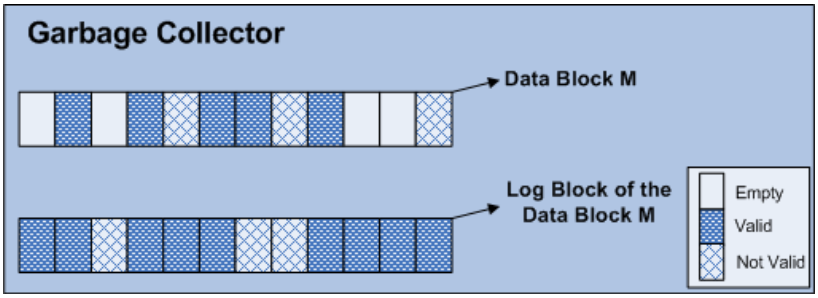

Figure 3.15: Garbage Collector

Secondly, another data block could want to use the same log block. Since by design, a log block could be used only by a single data block, again garbage collector should be called.

The garbage collector merges the log block and the corresponding data block by using the reserve block. Every valid page from data block and log block is written to the reserve block. After that, the reserve block gets all the valid pages, FTL changes its type from 'E' to 'D', it is now the data block. At this time there are two empty blocks, the old log block and the old data block. The old data block is turn into reserve block, the old log block is again the log block. When the data block and log block both have some valid pages, then the merge will be a full merge as in 2.3.4.3. The merge mechanism is depicted in figure 3.16.

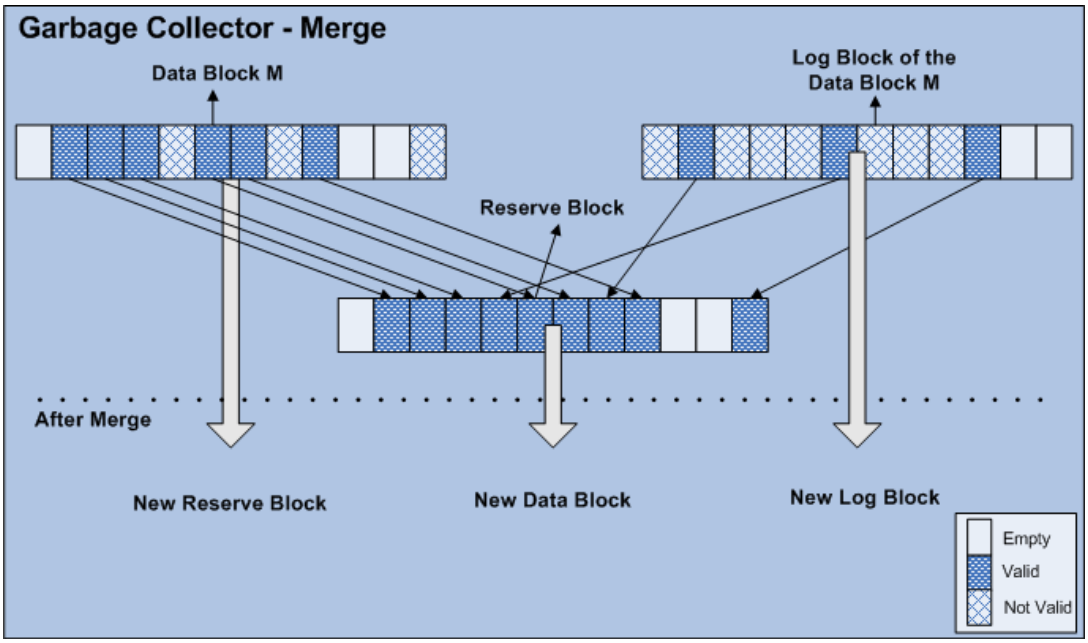

Figure 3.16: Garbage Collector - Merge

At the end, there is a data block with all the filled pages valid, there is a full empty log block and a full empty reserve block.

3.8 Simulation Environment

The thesis is implemented in C language on Ubuntu operation system by using Eclipse Galileo IDE. There are 1284 lines of codes with 7 source code files and 2 header files in the implementation. In figure 3.17, the screen shot of the simulation environment is given.

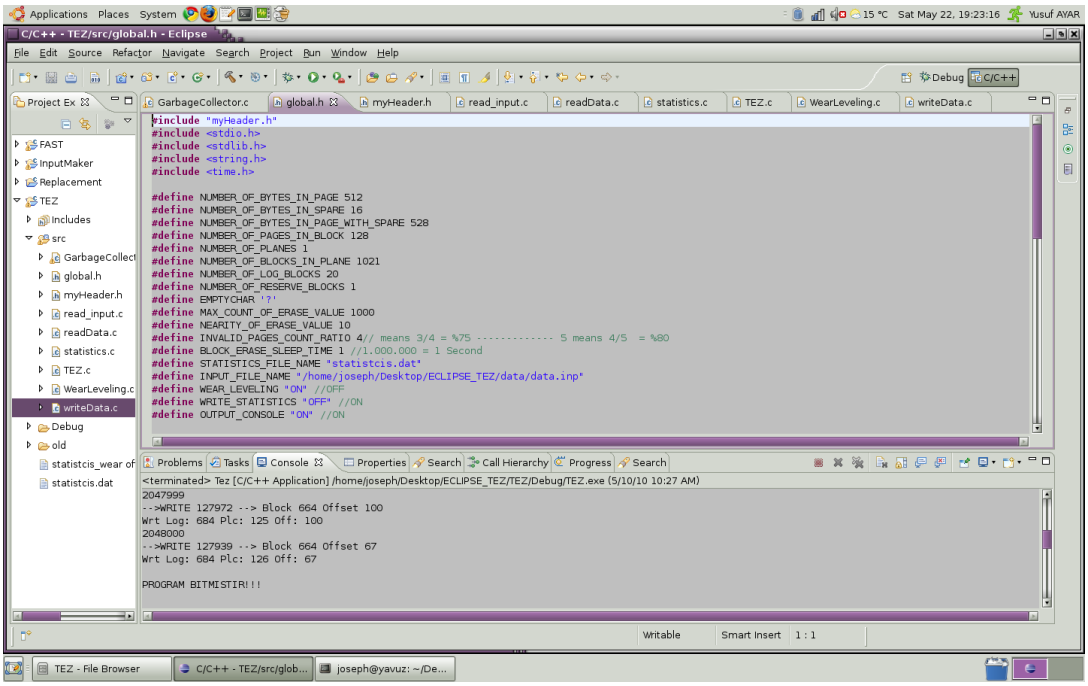

Figure 3.17: Simulation Environment

On the screen shot, "'global.h"' file can be partially seen. With this file, we can easily adjust Flash Memory's properties. The number of bytes in a page, the number of log blocks are all determined with this file at the beginning. The current condition on this screen is for a 64 MB size Flash Memory. Moreover, we can turn or off the wear leveling algorithm or have the statistics of any run in a file.

In "myHeader.h" file, there are function declarations.

With "read$_{i}nput.c$", $theinputisreadfromafileandeveryinputissenttothedesign.$

The file "TEZ.c" contains the main function.

With "readData.c", we conduct the READ operation in Flash Translation Layer.

With "writeData.c", the WRITE command is handled.

With ”WearLeveling.c”, wear leveling mechanism in Flash Translation Layer is implemented.

With ”GarbageCollector.c”, garbage collection mechanism in Flash Translation Layer is implemented.

With ”statistics.c”, we can obtain statistics from runs. The erase count of each block, the type of each block, total number of erase and garbage collector operations can be obtained via this statistics.

CHAPTER 4

EXPERIMENTAL RESULTS

It is important to assess an algorithm in order to see the strong and weak sides of the algorithm. To assess our BAST Flash Translation Layer algorithm, we conduct some runs with our algorithm. First the effect of wear leveling is examined. Then log block fullness, a criterion about garbage collector mechanism, is examined. At last, we will compare our BAST algorithm with other existing ones.

4.1 Wear Leveling

Wear leveling plays a vital role in Flash Translation Layer. Without wear leveling, some blocks will wear out too soon and they cannot be used anymore. Eventually the lifetime of Flash Memory will shorten. In the upcoming experiments, some input data is run with and without wear leveling algorithm in Flash Translation Layer. The effect of wear leveling will be shown.

4.1.1 Case I

The Case I is done with sequentially ordered data. The input size is 500 MB. A 5 MB sequential written file is written 100 times to the same location. Every write command tries to write a 512 Byte data. One little portion of the data is shown in figure 4.1.

Figure 4.1: Sample Input Data

Part A

In the Case I Part A, the input is run without wear leveling algorithm. The result is shown in figure 4.2. The figure 4.2 tells us that, blocks with block number around 200 and blocks with block number around 1,000 are too much used and erased in this run. The reason, why blocks around block number 200 are too much erased is that, the input is normally targeted to block numbers around 200. The blocks around block number 1,000 are erased too much, because of the Flash Memory design. In our design, the log blocks are located between block number 1,000 and 1,020. Therefore these blocks are erased too much because of updates to same location.

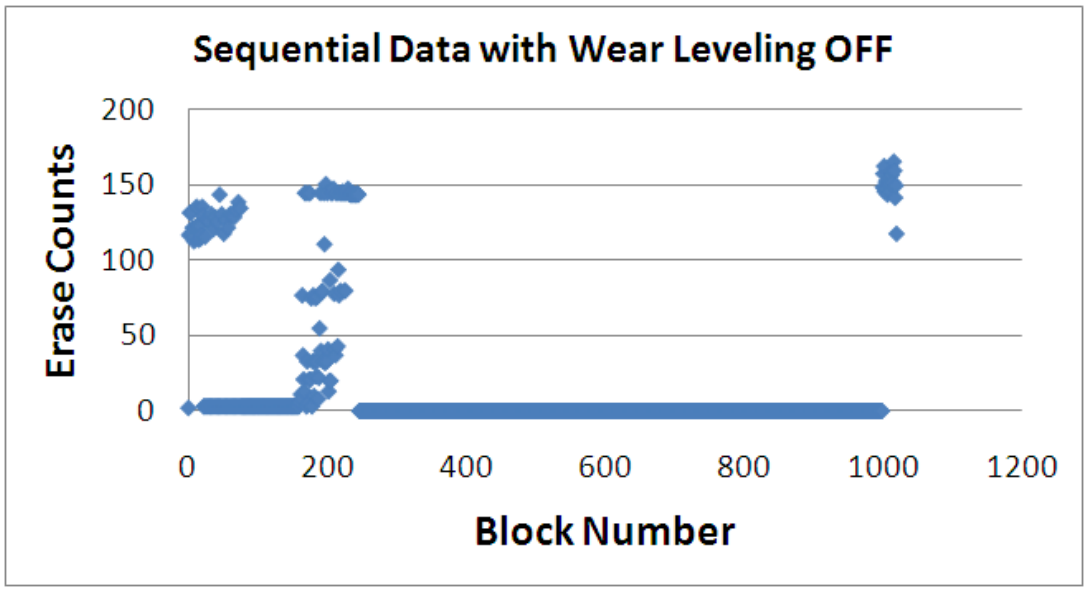

Figure 4.2: Input I - Part A

Without wear leveling, the log blocks are used too much, when there are multiple updates to a place. Also the type of the blocks is not changing into each other.

Part B

In the Case I Part B, the input is run with wear leveling algorithm. The result is shown in figure 4.3.

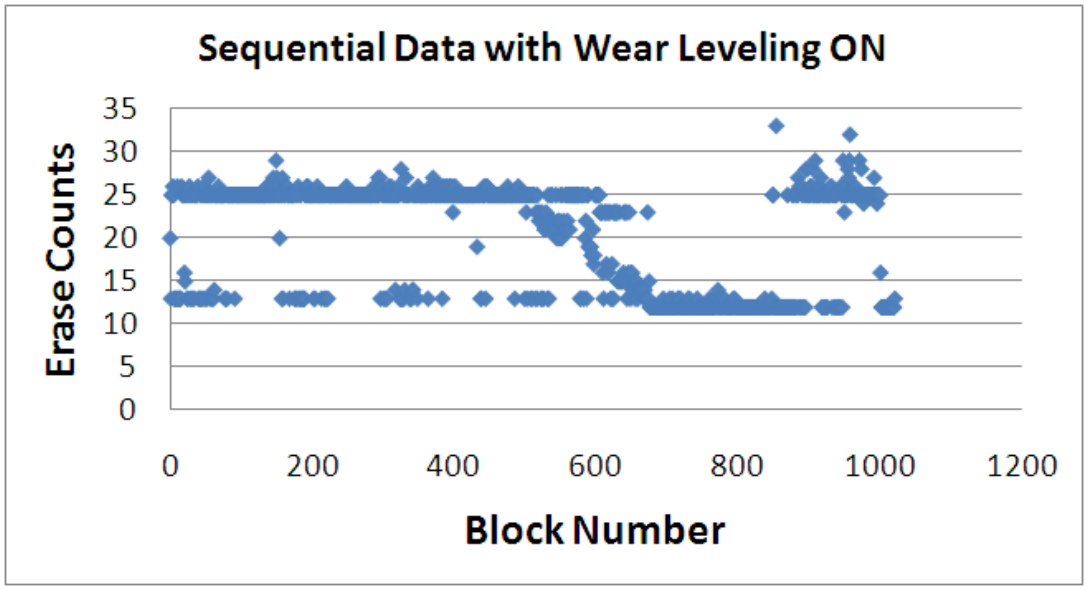

Figure 4.3: Case I - Part B

With wear leveling algorithm running on Flash Translation Layer, the erase count of all blocks in the Flash Memory are close to each other. Their erase count varies between 10 and 35. There is no accumulation of much erased log blocks like in the Part A. With wear leveling, the type of the block may change. In other words, if a log block is getting used too much, it will change into a data block and a data block with few erased count changes into the log block. In this run, the log blocks are no more between 1,000 and 1,020 because of the wear leveling algorithm. They are spread between data blocks.

Moreover, the empty block, which is used to merge the data block and the corresponding log block, is also used too much if there are much garbage collector calls. The type of this empty block may also change during usage of the Flash Memory.

4.1.2 Case II

The Case II is done with 400 MB of data with over 1 Million of write commands in it. It is a randomly constructed input, which is done with random function in C language.

Part A

In the Case II Part A, the input is run without wear leveling algorithm. The result is shown in figure 4.4.

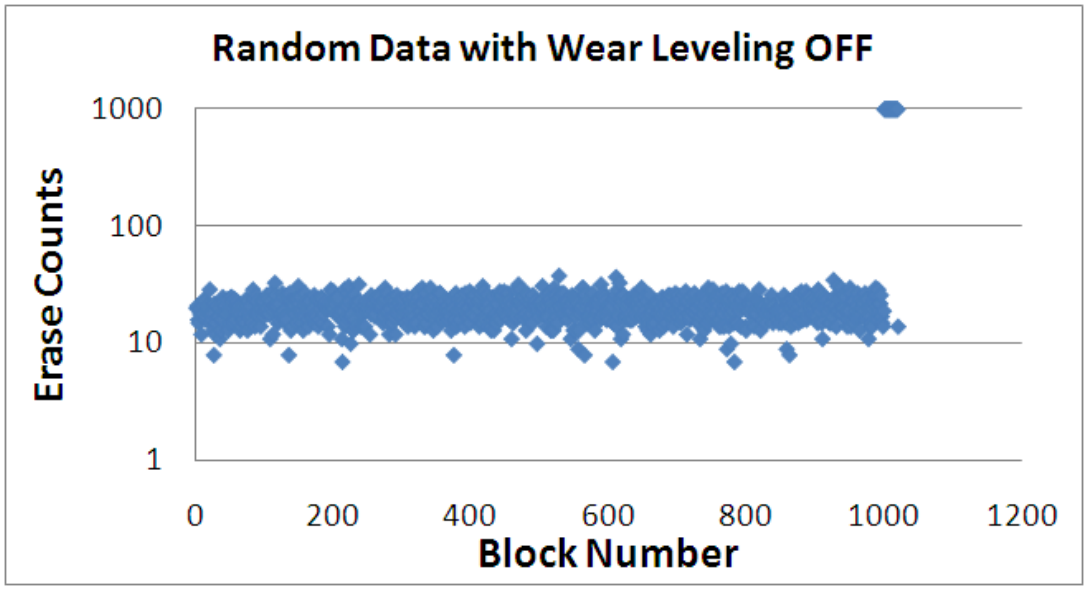

Figure 4.4: Case II - Part A

Bad block management in Flash Translation Layer marks blocks as bad, as their erase counts reach to a specific limit. This limit is 1,000 in our thesis. Therefore in Part A , the run has canceled before the input has ended, because there is a block with erase count 1,000. The log blocks with block number between 1,000 and 1,020 are erased too much and finally when a log block reaches the erase limit, the run has canceled.

In this run without wear leveling, the types of the blocks are not changing. Their types remain the same during the execution of Flash Translation Layer.

Part B

In the Case II Part B, the input is run with wear leveling algorithm. The result is shown in figure 4.5.

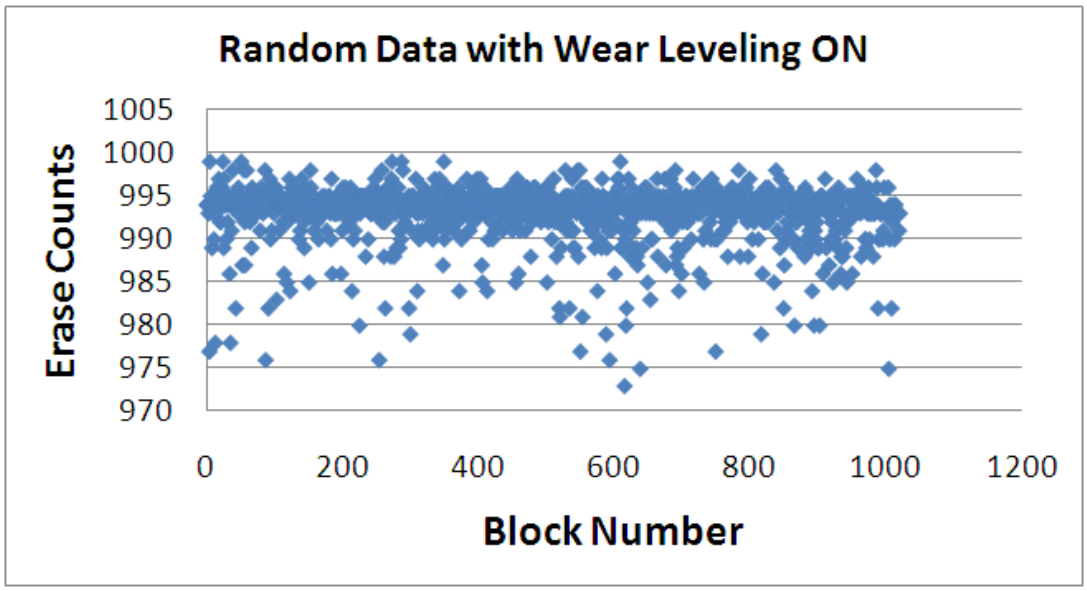

Figure 4.5: Case II - Part B

Again with wear leveling algorithm running on Flash Translation Layer, the erase count of all blocks in the Flash Memory are close to each other. Their erase count varies between 975 and 1,000. The program has run until the input has finished, because no block reaches the erase limitation. Like Case I Part B, the type of blocks has interchanged between each other.

4.2 Log Block Fullness

When FTL runs the garbage collector mechanism, the log block and the corresponding data block are merged. In some occasions, garbage collector merges some log blocks with not much more valid pages. In this situation, we say that the log block fullness is low. It is a bad property for garbage collector to call log blocks with lower log block fullness. If garbage collector mergers log blocks with more valid pages, then the log block fullness is higher.

In two different workload, the log block fullness of the current design is depicted. The first workload is composed of random data, whereas the second workload is composed sequential data. The random data is constructed with C random function. The sequential data is

constructed by writing sequentially ordered data. The result of the runs is shown in figure 4.6.

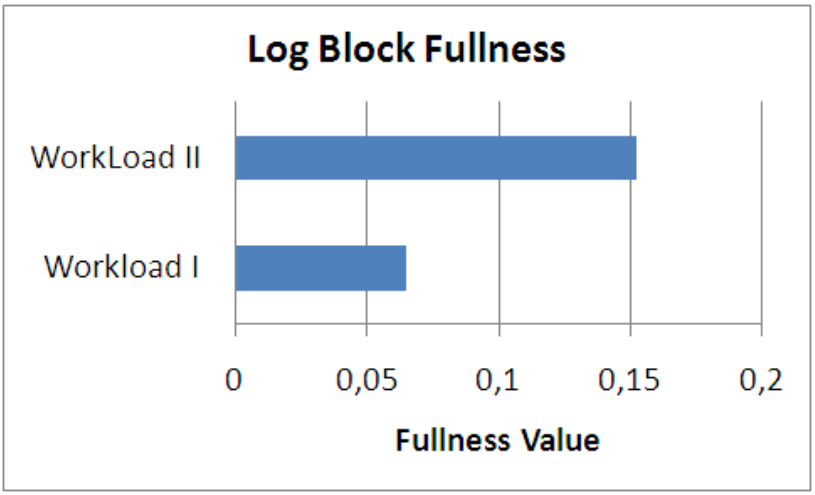

Figure 4.6: Log Block Fullness

In Case I, the data is random, the sequentiality of the data is low, therefore log blocks are merged before they are filled well. For every update to a random block, Flash Translation Layer has to obtain a new log block. When there are no more log blocks left, then the existing used log blocks have to used. Since these log block are not much used, they have few valid pages in it. Garbage collector merges this log block and provides an empty log block. At some point, every update to the Flash Memory will result a garbage collector call, which heavily effects the performance of Flash Memory.

In Case II, the sequentiality of the data is higher compared to Case I. Therefore the fill factor of the log blocks is higher, meaning the log blocks have more valid pages compared to the log blocks in Case I, while a garbage collector is called. In sequential data pattern, consecutive updates hit to the same block of Flash Memory. Because these hits use the same log block, the garbage collector is not much called.

From the log block fullness experiment; we can state that, in a sequential data, the log block fullness is high and there are less garbage calls. On the other hand, when the input data is random, the log block fullness is low and the number of garbage collector calls rises. Eventually the performance of Flash Memory decreases.

4.3 Comparison with Other Flash Translation Layers

It is crucial to have some competitors in order to compare something. To compare our Block Associative Sector Translation (BAST) design and assess its advantages and disadvantages, Replacement Block Algorithm and Fully Associative Sector Translation (FAST) Algorithm are partially implemented. Only the meta data operations are implemented in these algorithms. Therefore, we only able to assess the number of specific operations like erase operations or garbage collector calls. We cannot compare the algorithm in terms of the execution time, because the other two algorithms are not fully implemented.

Our BAST algorithm, Replacement Block and FAST algorithms differ from each other in terms of their address translation schemes. To understand their differences, we summarize their address translation schemes.

In our BAST algorithm, every log block is dedicated to a data block. Once a log block is used, it cannot be used for any other data block, until garbage collector merges it. The page offsets of the updated pages are not preserved in BAST design. There are extra data structures to keep this information. The updated page is written to the next empty position in the log block. In BAST design a log block may not have other log blocks connected to it.

In Replacement Block algorithm, a log block is dedicated to a data block. The updated page's offset of the data block has to be preserved in the log block, in other words the page offset never changes. A log block may also have other log blocks like a link list. In Replacement Block, every update to the same page results allocating a new log block, as long as there are free log blocks.

In FAST algorithm, a log block is not dedicated to a single data block like others. A log block can shared by multiple data blocks. Also like BAST, the page offset of the updated pages are preserved by keeping extra data structure. Therefore updated pages could be written to the next empty position in the log block.

Before the experimental results, first the workloads are going to be clarified.

The first workload is a sequentially ordered data. The second workload is a random ordered data and the last workload is a semi random ordered data in order to simulate the daily usage. The randomness of data is obtained by the rand function in C language.

4.3.1 Case I

In Case I, we intend to compare the total number of garbage collector calls of the three different algorithms. The number of the garbage collector calls is important, because garbage collector process is a heavy job for Flash Memory. It has to conduct all of the read, write and erase operations together in order to finish garbage collection.

In workload I in figure 4.7, the workload is a sequentially ordered data. The total numbers of garbage collector calls are near to each other, but Replacement Block algorithm is a bit ahead. Our thesis BAST algorithm has the highest garbage collector count.

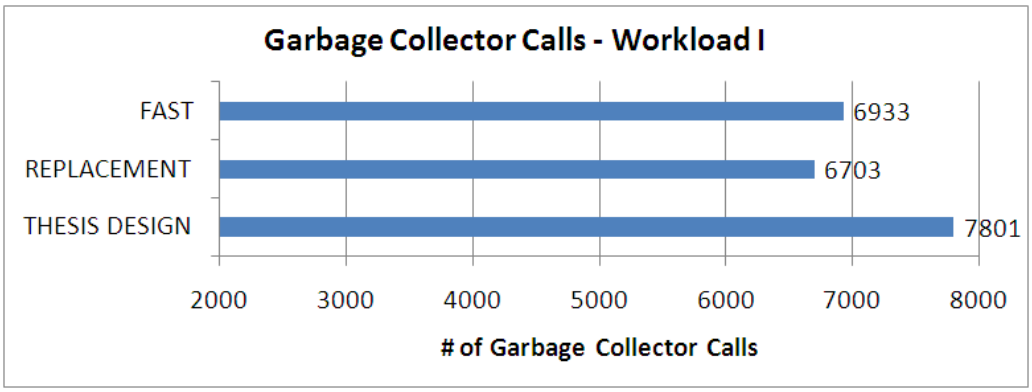

Figure 4.7: Garbage Collector Calls - Workload I

In workload II in figure 4.8, the workload is a random ordered data. FAST algorithm outperforms the other drastically. The Replacement Block has the worst performance, our BAST algorithm has a quite bad performance compared to FAST, but is better than Replacement Algorithm.

In workload III in figure 4.9, the workload is a semi random ordered data. FAST Algorithm is still better than the other algorithms and the Replacement Block has the worst performance.

If Case I is carefully inspected, we can state that, FAST has nearly the least garbage collector calls and has the best performance among others in terms of garbage collector mechanism. When the input data is random ordered, than the performance between FAST and the other algorithms increases. The reason behind this is, in FAST algorithm a log block is shared by many data block, allowing random updates have higher probability to hit to the same log block.

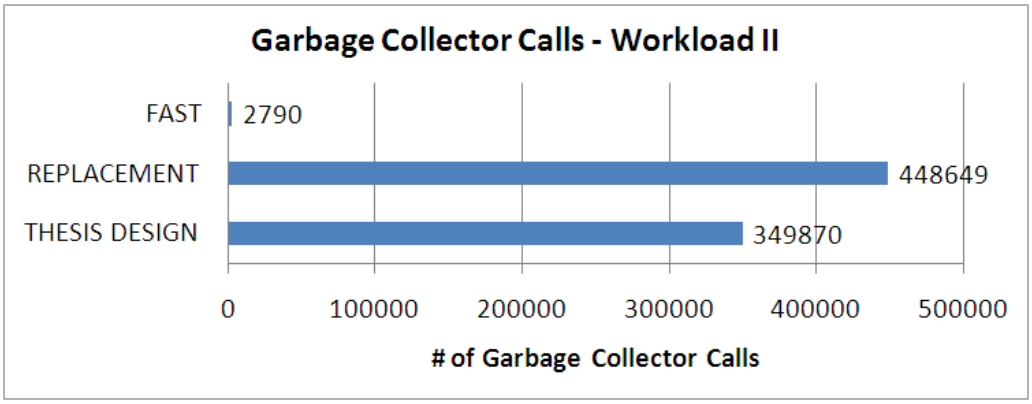

Figure 4.8: Garbage Collector Calls - Workload II

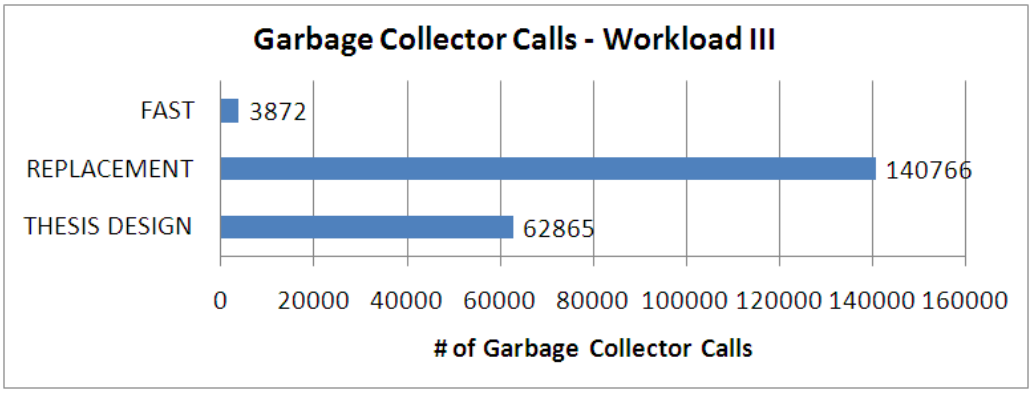

Figure 4.9: Garbage Collector Calls - Workload III

Our thesis BAST algorithm has a poor performance in workload II when compared with FAST. But in workload III, BAST's performance comes closer to FAST performance. Whenever the input gets more sequential, the BAST algorithm gets advantage.

4.3.2 Case II

Since an erase operation is much more longer than the other operations in Flash Memory, it is highly desired to have a minimum number of erase count. In Case II, we inspect the three algorithms in terms of the erase count of all the blocks in Flash Memory after workloads has run.

In figures 4.10, 4.11, 4.12 the total numbers of erase counts are depicted.

In workload I in figure 4.10, the workload is a sequentially ordered data. FAST algorithm has the maximum number of erase count that means it has the worst performance in terms of total number of erase counts. The Replacement Block algorithm is better than FAST algorithm and our BAST design outperforms the others.

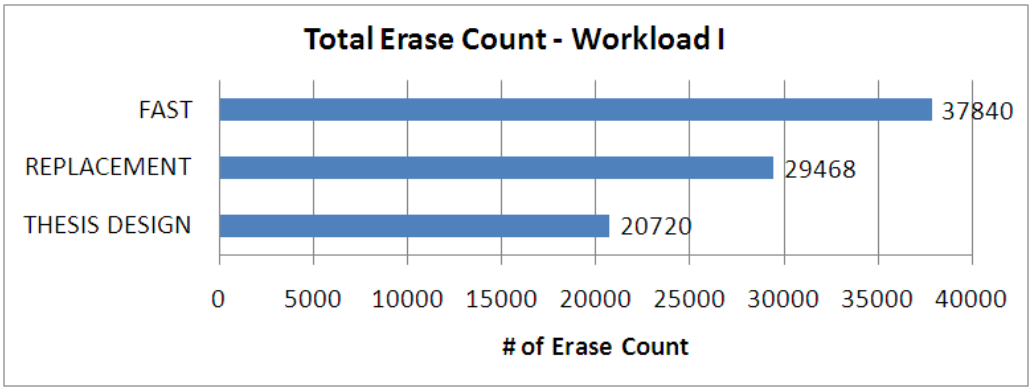

Figure 4.10: Total Erase Count - Workload I

In workload II in figure 4.11,the workload is a random ordered data. FAST algorithm outperforms the BAST design and Replacement Block design. The Replacement Block has the worst performance, meaning the total number of erase count of blocks is the highest.

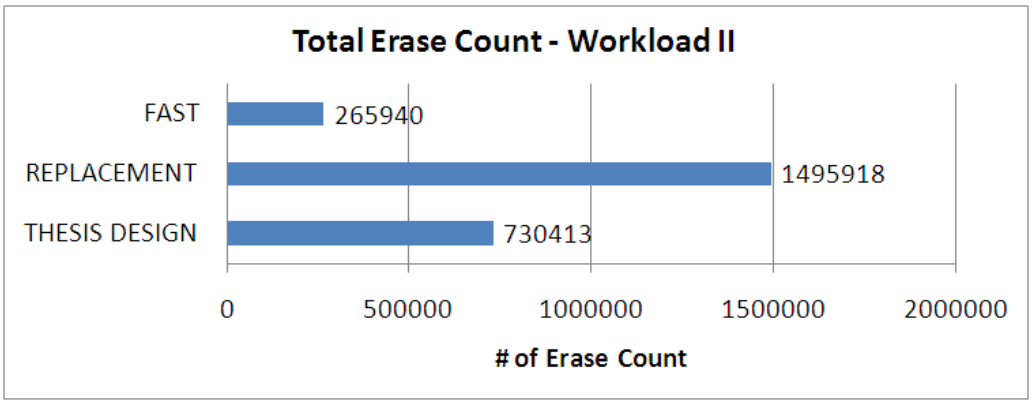

Figure 4.11: Total Erase Count - Workload II

In workload III in figure 4.12, the workload is a semi random ordered data. In this workload, again FAST Algorithm has a quite good performance compared to others. Since multiple data blocks in FAST share a log block, the possible garbage collector calls are reduced especially in a random workload. BAST Algorithm has a better performance than Replacement Block Algorithm.

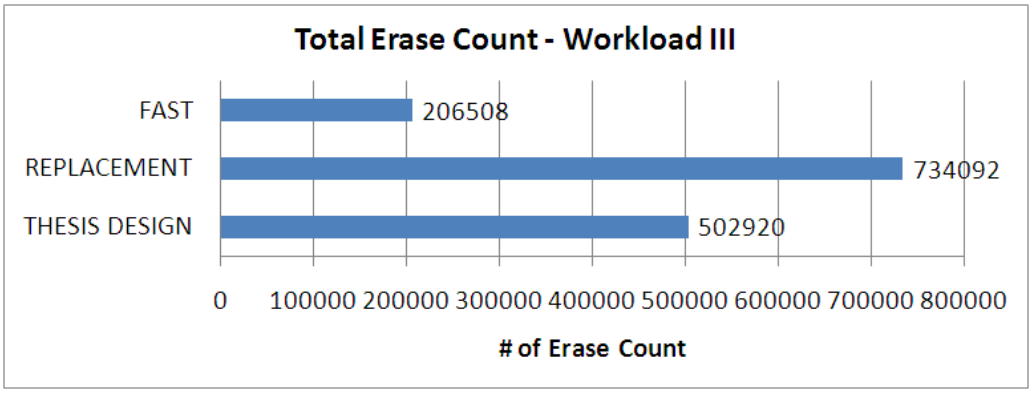

Figure 4.12: Total Erase Count - Workload III

The total erase count of all blocks can be a criterion to measure the performance of Flash Memory, since erase operations in a Flash Memory take longer time when compared to the other commands. In workload I, when the input is sequentially ordered, the BAST algorithm outperforms the others. On the other hand, in the second and third workloads, FAST beats the others. In second workload, the difference between BAST and FAST algorithms is higher than compared to workload III. Like in the workload II, FAST gains performance if the input gets more random because of its log block design. FAST algorithm can handle random updates

more easily than thesis BAST and Replacement Block algorithms.

4.3.3 Overall Comparison

When comparing the algorithms with an overall perspective, FAST can be seen more advantageous than Replacement Block and our BAST algorithms. The daily usage of a USB Flash Memory of an ordinary computer user has randomness in it, therefore FAST is more suitable for daily usage.

On the other hand, there are some usage areas, where Flash Memory will work better with BAST design. For example, in some cases, only sequential read and write patterns are available for the Flash Memory like reading a whole program from Flash Memory. In these cases, BAST will work with a higher performance than FAST algorithm. In addition to this, BAST has a simpler design and few data structure when compared with FAST design.

It is not right to say, FAST outperforms the other algorithms in every way. It is important to know the capabilities of the algorithms and assess their advantages. Therefore it is a wise choice to select an appropriate algorithm for the intended usage.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Flash Memory is non-volatile computer memory that is used almost everywhere in our lives. It is non-volatile, it consumes low power, it has a high shock resistance and most important it is mobile.

Besides the advantages, Flash Memory has some disadvantages because of its nature. Some of these problems are address mapping issue, erase-before-write constraint, power-off recovery, bad-block management and wear leveling. These limitations effect the performance of Flash Memory heavily. For example, without wear leveling, a Flash Memory will wear out too soon than we expected. Also when power-off recovery is not implemented, then any power shortage will result data loss.

At this point, Flash Translation Layer is an answer for these problems. It is a software module running inside the Flash Memory conducting all the operations between Flash Memory and operating system and other operations inside the Flash Memory.

In this study, we first give some information about Flash Memory. Then we try to explain the role of the Flash Translation Layer. Then we give information about various Flash Translation Layer's design in the market. We try to analyze their advantages and disadvantages. Also we have a comprehensive view to the Flash Translation Layer's main duties, such as address translation, bad block management, wear leveling.

Finally we come up with our design. We examine the structures and the algorithms used in the design. The address translation, bad block management, wear leveling and garbage collector designs of the design are depicted.

After that, some empirical study is presented. This empirical studies first show the effect of Flash Translation Layer in terms of wear leveling. Without Flash Translation Layer, some blocks may be useless in a short time. Without wear leveling, the most used log blocks or most targeted blocks will be erased too much and eventually their erase count will reach to the limit too soon. From this point on, these block could not be used and the overall capacity of Flash Memory will decrease. On the other hand, when wear leveling algorithm is used in the Flash Memory, the erase count of blocks will be close to each other, meaning no block will wear out too soon when compared with other blocks.

Secondly, the log block fullness is inspected in various workloads. If garbage collector runs well on filled log blocks, the log block fullness is good. If we run the algorithm on different workloads, we can state that, in a random workload, the log block fullness is quite poor. On the other hand, the log block fullness gets better if the input gets more sequential. By inspecting the log block fullness values of log blocks before the garbage collector, we can increase the performance of Flash Memory by calling the log blocks with low log block fullness value in idle time of Flash Memory.

Then our designed BAST Algorithm is compared with other Flash Translation Layer Algorithms, FAST and Replacement Block Algorithms. The total garbage collector call count and the total erase count of blocks are inspected in order to compare the algorithms.

Total garbage collector call count is a good benchmark for Flash Translation Layers. Since a garbage collector call is a time consuming job for Flash Translation Layer, it is better to have minimum number of Garbage Collector calls.

In general Replacement Block Algorithm has the worst performance among the three. The Replacement Block Algorithm has a simple design. This simplicity yields in bad performance in Flash Memory.

FAST Algorithm outperforms in most cases, but BAST comes closer to FAST in some runs. As seen in Workload II and Workload III, whenever the input gets random, FAST outperforms against BAST. FAST Algorithm can handle random inputs more easily, because random hits in FAST will eventually result less garbage collector calls. The reason behind this is that in FAST many data blocks share a log block together. That brings flexibility to FAST algorithm, especially when input is random.

BAST Algorithm has a simpler design than FAST Algorithm. A log block is associated to a unique data block in BAST. Therefore BAST's garbage collector mechanism is also simpler and faster than FAST's.

In this thesis, we design and implement a BAST Flash Translation Layer Algorithm. This study gives us full control over a Flash Translation Layer Algorithm. The performance in case of speed or read-write throughput can be changed via implementation of the code or global variables. That means, we could easily change the algorithm according to our needs.

While implementing our BAST design, we note some points that could help enhance the performance of the Flash Memory. These points could yield better Flash Translation Layer algorithms in the future.

- We are going to reduce the size of the meta data structure, *All_Blocks_Physical* structure. The *logPages[]* structure can be removed from this structure and can be assembled in to spare area of the pages. This will reduce the size of *All_Blocks_Physical*.
- In current design every log block is dedicated only to a single data block. In random writes, this feature may lead to an overhead in garbage collection. We will dedicate a log block to a bunch of data blocks. This will eventually decreases the garbage collector overhead.
- In Garbage Collector mechanism, if both data and log blocks have valid pages, then full merge is used during merging the blocks. In case all the pages in log block are valid, meaning there are sequential writes to the data block and there is no valid block in data block, then switch merge in 2.3.4.1 will reduce the overhead of extra erase operations. We will have a control mechanism to detect the occasions for switch merge and try to have more efficiency.
- With time, there could arise many invalid pages in a block. For example, many updates to the same page will result invalid pages in the corresponding log block. In future, we will inspect the invalid page ratio of each block. If this ratio reaches to a specific point, we will call the garbage collector in idle time of Flash Memory. This will enhance the read-write throughput of the Flash Memory.

REFERENCES

[1] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, p. 18, 2007.

[2] *Application Note for NAND Flash Memory (Revision 2.0)*. SAMSUNG, 1999.

[3] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "System software for flash memory: A survey," in *Embedded and Ubiquitous Computing*, vol. 4096 of *Lecture Notes in Computer Science*, pp. 394–404, Springer Berlin / Heidelberg, 2006.

[4] "Ces 09: Asus 512gb ssd laptop; world first - http://www.tomshardware.com/news/asus-ssd-laptop,6771.html, last visited in 10/03/2010."

[5] J. Brewer, *Nonvolatile Memory Technologies with Emphasis on Flash*. Wiley-IEEE Press, 2008.

[6] Y.-L. Tsai, J.-W. Hsieh, and T.-W. Kuo, "Configurable nand flash translation layer," in *SUTC '06: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06)*, (Washington, DC, USA), pp. 118–127, IEEE Computer Society, 2006.

[7] Y. Wang, D. Liu, M. Wang, Z. Qin, Z. Shao, and Y. Guan, "Rnftl: a reuse-aware nand flash translation layer for flash memory," *SIGPLAN Not.*, vol. 45, no. 4, pp. 163–172, 2010.

[8] A. M. A. V. Roberto Bez, Emilio Camerlenghi, "Introduction to flash memory," in *Proceedings of The IEEE, Vol. 91, No. 4*, pp. 489–502, IEEE Computer Society, 2003.

[9] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for nand flash memory," in *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, (New York, NY, USA), pp. 161–170, ACM, 2006.

[10] S. E. Wells, "Method for wear leveling in a flash eeprom memory." US 5,341,339, 8 1994.

[11] T. Kgil and T. Mudge, "Flashcache: a nand flash memory file cache for low power web servers," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, (New York, NY, USA), pp. 103–112, ACM, 2006.

[12] C. Park, W. Cheon, Y. Lee, M.-S. Jung, W. Cho, and H. Yoon, "A re-configurable ftl (flash translation layer) architecture for nand flash based applications," in *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, (Washington, DC, USA), pp. 202–208, IEEE Computer Society, 2007.

[13] "Bad block management in nand flash memories - http://www.numonyx.com/documents/application notes/an1819.pdf, last visited in 08/04/2010."

[14] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, 2007.

[15] C.-H. Wu and T.-W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," in *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, (New York, NY, USA), pp. 601–606, ACM, 2006.

[16] H. jin Cho, D. Shin, and Y. I. Eom, "Kast: K-associative sector translation for nand flash memory in real-time systems," in *DATE*, pp. 507–512, 2009.

[17] A. Ban, "Flash file system." US 5,404,485, 4 1995.

[18] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A spaceefficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, pp. 366–375, 2002.

[19] S.-I. J. Soo-Young Kim, "A log-based flash translation layer for large nand flash memory," in *In Proceedings of the 8th International Conference Advanced Communication Technology (ICACT)*, pp. 1641–1644, IEEE Computer Society, 2006.

[20] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "Last: locality-aware sector translation for nand flash memory-based storage systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, 2008.