NAND FLASH TRANSLATION LAYER (NFTL)

A PROJECT REPORT

Submitted by

TEJAS PATEL (080110107044)

SUDEEP BORA (080110107008)

HARIKRISHNA RANPARIYA (080110107047)

In fulfilment for the award of the degree

Of

BACHELOR OF ENGINEERING

In

Computer Engineering

G.H. Patel College of Engineering and Technology

Bakrol Road, Vallabh-Vidhyanagar-388120 (Gujarat)

Gujarat Technological University, Ahmedabad

May, 2012

G.H. Patel College of Engineering and Technology

Department of Computer Engineering

ACKNOWLEDGEMENT

An effort to develop a research project like this is a group activity and calls for generous help and support from a lot of people and we would like to acknowledge the contribution of certain distinguished people, without their support and guidance this research work would not have been completed.

We take immense pleasure in thanking **Prof. Hetal Gaudani**, **Mr.Tejas Vaghela**, Manager of System Level Solutions (SLS) Corp. PVT Ltd. and **Mr. Pratik Patel,** Employee at SLS for having permitted us to carry out this project work. We wish to express our deep sense of gratitude to our internal guide, Prof. Hetal Gaudani for her able guidance and useful suggestions, which helped us in completing the Project-2, work in time. We would also like to thank Mr.Tejas Vaghela and Mr. Pratik Patel who had been a source of inspiration for their timely guidance in the conduct of our project work and for all their valuable assistance in the project work.

Finally, yet importantly, we would like to express our heartfelt thanks to our beloved parents for their blessings, our friends/classmates for their help and wishes for the successful completion of this Project-2.

Acknowledgements and thanks are also extended to all the authors whose articles/conference papers/White papers have been referred in this project work.

TEJAS PATEL (080110107044)

SUDEEP BORA (080110107008)

HARIKRISHNA RANPARIYA (080110107047)

ABSTRACT

*Flash memory is non-volatile computer storage chip that can be electrically erased and reprogrammed.**Solid-State Drives (SSDs)** are data storage devices that use NAND Flash memory to store persistent data. SSD provide an increasingly attractive alternative to Hard Disk Drives for laptop, desktop, server because of its features like low-power consumption, non-volatility, high random access performance, shock resistance and high mobility. Due to the certain limitations of NAND Flash memory like erase before write operation, asymmetric speed of operations, small and lightweight form factor and limited endurance, straightforward replacement of magnetic disks (HDD) with SSD is difficult. Researchers have tried to solve these inherent difficulties by putting an effort to design high performance SSD and lots of researches have been carried out. It is important to develop a software layer that can deal with complex read/write pattern which are normally found in high computing devices in a large scale. Normally workloads of high computing devices are characterized by a large number of small, random and skewed operations. So, to improve the performance of NAND Flash memory for random Read/Write request and to make effective use of the advantages of flash memory, while overcoming its constraints (Limitations), system software termed **Flash Translation Layer (FTL)** has been introduced for Flash memories. Most existing FTL schemes are optimized for some specific access patterns or bring about significant overhead of merge operations under certain circumstances. In this project, we propose a novel FTL scheme named GFTL that exhibits low response latency and at the same time, eliminates the overhead of merge operations completely.*

2012

CERTIFICATE

Date: 23-05-2012

This is to certify that the Project entitled "NAND Flash Translation Layer (NFTL)" has been carried out by **TEJAS PATEL(080110107044), SUDEEP BORA(080110107008) and HARIKRISHNA RANPARIYA(080110107047)** under our guidance in fulfilment of the degree of Bachelor of Engineering in Computer Engineering (8^{th} Semester) of Gujarat Technological University, Ahmedabad during the academic year 2011-2012.

Prof. Hetal Gaudani
Internal Guide

Prof. Maulika S. Patel
Head of the Department

Mr. Tejas Vaghela
External Guide

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS, ABBREVIATIONS AND NOMENCLATURE

Abbreviation	Description
FTL	FLASH TRANSLATION LAYER
HDD	HARD DISK DRIVE
SSD	SOLID STATE DRIVE
MLC	MULTI LEVEL CHIP
SLC	SINGLE LEVEL CHIP
MSB	MOST SIGNIFICANT BIT
LSB	LEAST SIGNIFICANT BIT
OOB	OUT OF BAND
RAM	RANDOM ACCESS MEMORY
GO	GROUP OFFSET
LBN	LOGICAL BLOCK NUMBER
LGN	LOGICAL GROUP NUMBER
PBN	PHYSICAL BLOCK NUMBER
PMT	PAGE MAPPING TABLE
LPN	LOGICAL PAGE NUMBER
PPN	NEXT PHYSICAL PAGE NUMBER
ONFI	OPEN NAND FLASH INTERFACE

Chapter 1

INTRODUCTION

1.1: BRIEF INTRODUCTION OF THE PROJECT:

In the past few years flash memory became more and more important because of its small size, low power consumption, storage density, shock resistance and low cost compared to other EPROM. In fact Flash memory is also a non-volatile memory. It has capability to hold and store data even when the power is turned off. Flash memory is the choice when solid state non-volatile memory is needed, especially in mobile devices. It is an excellent storage solution for many applications such as MP3 players, USB drives, removable storage cards, cell phones, and an endless number of other applications where mobility, power use, speed, and size are key factors. But as the price for flash memory is rapidly decreasing and the storage density of flash memory chips is growing, it becomes feasible to use flash memory even in notebooks, desktop computers and servers. It is now possible to use flash memory as main storage device. Such a device is called a Solid State Drive (SSD).

A solid state disk is a data storage device that uses non-volatile memory to store persistent data. Most of them are referred to as NAND-based flash solid state storage device. In terms of function SSD's serve the same purpose as typical hard disk drive (HDD), but are much faster than your typical hard drive. SSD addresses a lot of issues that a hard disk drive (HDD) has with the magnetic disk.

Comparison with HDD:

- A solid state drive has no mechanical parts. It contains only flash memory chips. Since there are no mechanical parts and no cooling fans required in NAND flash memory. So a solid state drive does not make noise.

- In a HDD the hard drive has an actual magnetic disk that needs to spin and it has a moving read/write head. So hard disk drives are slower because they need to wait for these moving parts to get into place before they can save information to the hard drive or before they can read files. In contrast, solid state drives have no moving parts and they store information in microchips, much like a USB flash drive and they can instantly start saving information or reading files, without having to wait for any moving parts to get in place. SSD can offer much less mechanical failures and generate much less heat.
- The density of the flash memory chips can be very high. Therefore it is possible to produce solid state drives which are generally smaller and lighter than common hard drives.
- Flash memory consumes less power than hard drives. That is one of the reasons why flash memory is often assembled in mobile devices.

All these advantages of SSD mention here give out better bandwidth and I/O performance. Nowadays, a lot of commercial applications use NAND-based flash memory SSD to improve their products' performance. Not only USB sticks and camera memory cards appear on today's market, the NAND-based flash disks that can provide interfaces like IDE and SATA disks are also becoming more and more popular. These solid state disks approximate interfaces of the traditional rotating disks, but they have much better random read performance owing to their electronic nature. Their random write performance is also comparable to those rotating disks after design optimization.

A NAND flash memory chip is composed of a fixed number of **blocks** and these blocks are in turn made of the large or small number of **pages**. NAND flash memory does not support in-place update. Once a page is written, it should be erased before the subsequent write operation is performed on the same page. Since read and write (or program) operations are executed on a page basis while erase operations on a much larger block basis, NAND flash memory is sometimes called a write-once and bulk erase medium. A page that is the basic unit for read/write operations contains a user area and a spare area, where the user area is for data storage and the spare area stores the house-keeping information such as error detection/correction codes,

status flags, and Page Mapping Tables (PMTs) etc. The following limitations lead to problems by increasing overhead in operations.

Flash Memory has typically three main limitations:

1) Write Once
2) Asymmetric operations speed
3) Limited number of erase/program cycles

1)Write Once:

Blocks of flash memory need to be erased before they are rewritten. Hence these pose a problem while writing a page on invalid page. Pages inside flash memory are usually classified into valid page, invalid page and free page. Once the page is written and if we want to write on same page again we need to first erase that page than only we can write again on it. Hence this poses an overhead in write during page update operation.

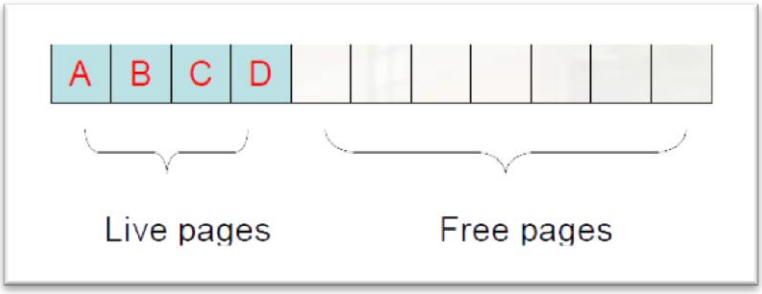


Figure 1.1: Live and free pages in a block.

From Figure 1.1 we can note that updating pages A and B leads to writing them into another set of pages and invalidating the previous versions of the pages. These happened because of the basic property Write once of Flash memory.

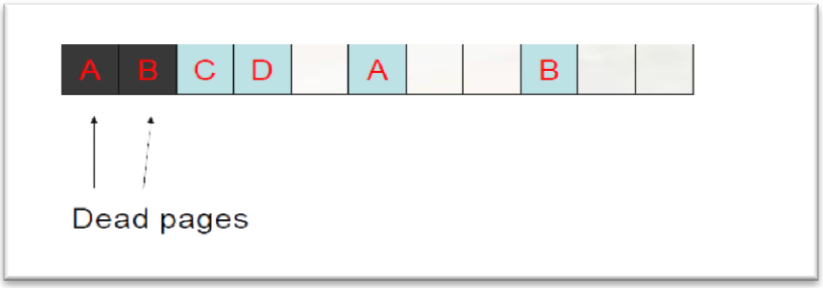


Figure 1.2: Dead pages in a block

The invalidated page (Figure 1.2) can now be used only after the entire block is erased (because erase operations are performed on block level), hence these leads to an overhead.

2) Asymmetric operations speed:

There are three possible operations that may be performed on a flash memory device. They are read, program (write) and erase. Read and write operations are performed at page level and erase operation is performed at block level. The read operation is the fastest of the three. Write operation is comparatively slower than read operation though it is performed at page level. The erase block operation is the longest and may take as long as 3 milliseconds for NAND flash memory. Time required to erase a block (specified in milliseconds) are usually much greater than write and read time (specified in tens or hundreds of microseconds). Thus, it is required to decrease the number of erase operations as much as possible.

3)Limited number of erase/program cycles:

In NAND flash memory the number of program/erase cycles for a block is limited to about 100,000 to 1,000,000 times depending whether the Flash Memory is MLC or SLC respectively. Hence these restrict the random page writing in a block.Hence if a particular page is written again and again then the life of the page decreases and slowly and steadily the different parts of flash memory gets dead permanently. Thus, the number of erase operations should be minimized not only to improve the overall performance but also to extend the lifetime of NAND flash memory.

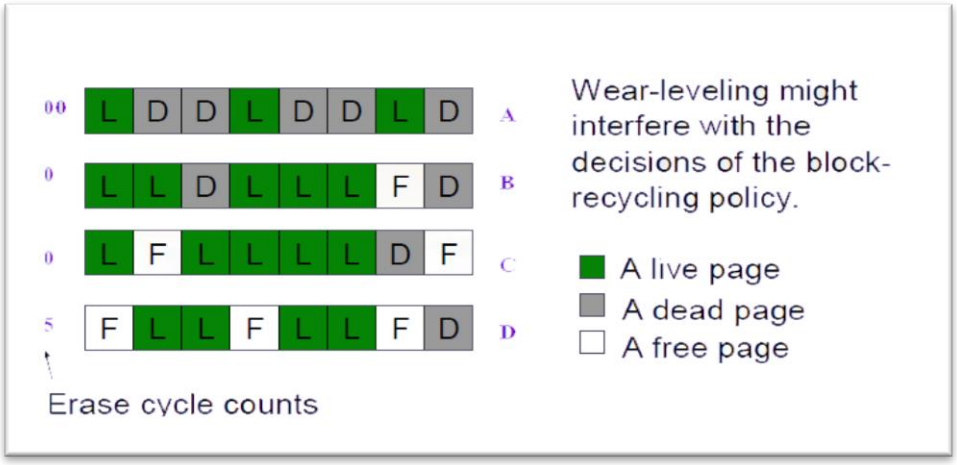


Figure 1.3: Wear levelling in flash memory.

It can inferred from the Figure1.3 that Block A has been erased most of the time and Block B has been erased least number of times. Hence Block A will get wear out fast as compared to other blocks. These will surely decrease the life cycle of the flash memory.

Hence in order to increase the life time of the Flash Memory the erase operations should be decreased to the minimum value possible and Wear-leveling i.e. to maintain the number of erase operations of all the blocks, should be kept approximately equal.

Due to the limitations of NAND Flash memory, software layer called FTL is used between file system and NAND Flash memory. FTL performs the task like wear levelling, garbage collection, and most importantly logical to physical address mapping of the pages. Most of the FTL algorithm uses the hybrid mapping technique.

1.2: PROJECT BACKGROUND:

Flash memory is a type of Electronically Erased Programmable Read Only Memory (EEPROM). The Flash memory was invented by Dr Fujio Masuoka while working for Toshiba in 1984. According to Toshiba, the name 'Flash' was suggested by Dr Masuoka's colleague, Mr Shoji

Ariizumi, because the erasure process of the memory contents reminded him of a flash of a camera. Dr Masuoka presented the invention at the IEEE 1984 Integrated Electronics Devices Meeting held in San Jose, California. Intel is the first company to introduce commercial (NOR type) flash chip in 1988 and Toshiba released world's first NAND-flash in 1989[7].

Flash memory is non-volatile computer storage chip that can be electrically erased and reprogrammed. Flash memory has a grid of columns and rows with a cell that has two transistors at each intersection. Flash memory cells are made up of floating gate transistors to store information. Each bit of data in a flash memory device is stored in a transistor called a floating gate. Flash works using an entirely different kind of transistor that stays switched on (or switched off) even when the power is turned off.

Flash memory is used in SSD which can be used as an alternative to the hard drives. Flash memory can be classified into 2 types on the basis of the gates used in their construction. The first type is NOR-flash memory which was first launched by INTEL for commercial use in 1988. The second type is NAND flash memory which was first launched by TOSHIBA in 1989.

Table 1.1: Comparison of NOR Flash and NAND Flash

	NOR Flash	**NAND Flash**
High-Speed Access	YES	YES
Page-Mode Data Access	NO	YES
Random Byte Level Access	YES	NO
Typical Users	Cell Phones, BIOS Storage for PC's ,Networking Device Memory	PDA's ,Digital Cameras,MP3 Players ,**SOLID STATE DISK DRIVES**(SSD) ,SET-TOP Boxes, Industrial Storage

Raw NAND is available in two types: Single-Level Cell (SLC) and Multi-Level Cell (MLC). The basic difference between the two is the number of data bits each memory cell holds. For SLC, exactly one bit of information is held in each memory cell. MLC devices store two or more bits per cell, though this is not without trade-off. By storing more bits per cell, the storage capacity of the NAND can be doubled, quadrupled, or even more for roughly the same cost to produce the die as compared to SLC. Looking at it another way, MLC can produce the same storage capacity as SLC but using a smaller die, and therefore at a lower cost.

1.2.1: SLC (SINGLE-LEVEL CHIP) NAND FLASH MEMORY:

As the name suggest, SLC Flash stores one bit value per cell, which basically is a voltage level. The bit value is as shown in table 1.2.

Value States:

Table 1.2: SLC Levels

Value	State
0	Programmed
1	Erased

Since there are only two states, it represents only one bit value. As seen in Table1.2, each bit can have a value of "programmed" or "erased".

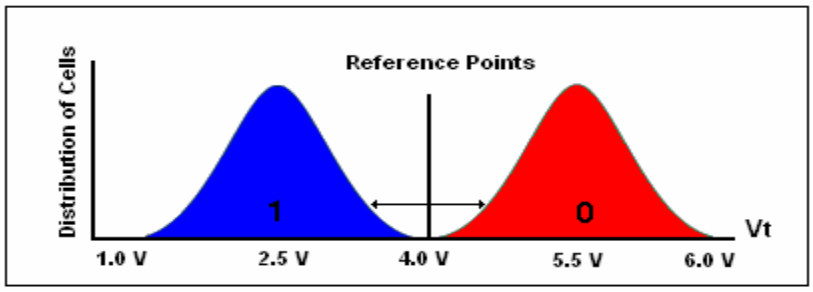


Figure 1.4: Voltage Reference for SLC[12]

A "0" or "1" is determined by the threshold voltage V_t of the cell. The threshold voltage can be manipulated by the amount of charge put on the floating gate of the Flash cell. Placing charge on

the floating gate will increase the threshold voltage of the cell. When the threshold voltage is high enough, around 4.0V, the cell will be read as programmed. No charge, or threshold voltage < 4.0V, will cause the cell to be sensed as erased.
SLC Flash is used in commercial and industrial applications that require high performance and long-term reliability. Some applications include industrial grade compact Flash cards or Solid State Drives (SSDs).

1.2.2: MLC (MULTI-LEVEL CHIP) NAND FLASH MEMORY:

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

A NAND flash memory chip consists of multiple blocks, and each block is composed of a fixed number of pages. Block is the basic unit for erase operations while page is the minimum unit for read/write operations. A page contains the data area and the spare area known as *Out Of Band (OOB)* area. The OOB area is used to store the house-keeping information (e.g., error correction code).

In NAND flash memory, when one page is written, it cannot be updated (rewritten) until the block with this page is erased. Therefore, “out-place update” is adopted so that the data are written into another new free page. The page that contains the old version of data is an invalid

page. With write operations propagating in flash memory, the free space will become lower, and invalid pages scattered over blocks should be reclaimed (called "garbage collection"). In order to hide the time consuming garbage collection and to overcome the "out-place update" constraint, a software layer called flash translation layer is designed to emulate the flash memory as a block device so that it can provide transparent storage service.

1.2.3: Comparison between SLC and MLC:

Now the differences between SLC and MLC have been explained, let's compare their specifications to help further make a distinction between the two grades.

Table 1.3: Comparison between SLC and MLC [12]

	SLC	MLC	
Density	16 Mbit	32 Mbit	64 Mbit
Read Speed	100 ns	120 ns	150 ns
Block Size	64 Kbyte	128Kbyte	
Architecture	x8	x8/x16	
Endurance	100,000 cycles	10,000 cycles	
Operating temperature	Industrial	Commercial	

Let's compare each characteristic in table 1.3. Using the same wafer size, you can double the density of the MLC Flash by using the charge placement technology. Thus, MLC has greater densities.

The read speeds between SLC and MLC are comparable. Reading the level of the Flash cell compares the threshold voltage using a voltage comparator. Thus, the architecture change does not affect sensing. In general, the read speeds of flash are determined by which controller is used.

The endurance of SLC Flash is 10x more than MLC Flash. The endurance of MLC Flash decreases due to enhanced degradation of Si. This is a main reason why SLC Flash is considered industrial grade Flash and MLC Flash is considered consumer grade flash.

Higher temperatures cause more leakage in the cells. Combined with the increased sensitivity required to differentiate between the levels, this leakage will cause the sensors to read the wrong level. As a result, the operating temperature of MLC spans only the commercial range. Leakage is not significant in SLC flash and thus, it can operate in an industrial temperature range.
Table 1.4again summarizes the advantages and disadvantages of SLC flash and MLC flash.

Table 1.4: Pros and cons of SLC and MLC [12]

	SLC	MLC
High Density	NO	YES
Low Cost per bit	NO	YES
Endurance	YES	NO
Power consumption	YES	NO
Write/Erase speed	YES	NO
Operating temperature	YES	NO
Write/Erase Endurance	YES	NO

1.2.4: Different Addressing Schemes for FTL:

In the past decade, three types of addressing schemes have been proposed for flash translation layer (FTL):-
1) Page-level mapping
2) Block-level mapping, and
3) Hybrid-level mapping.

1) Page-Level mapping

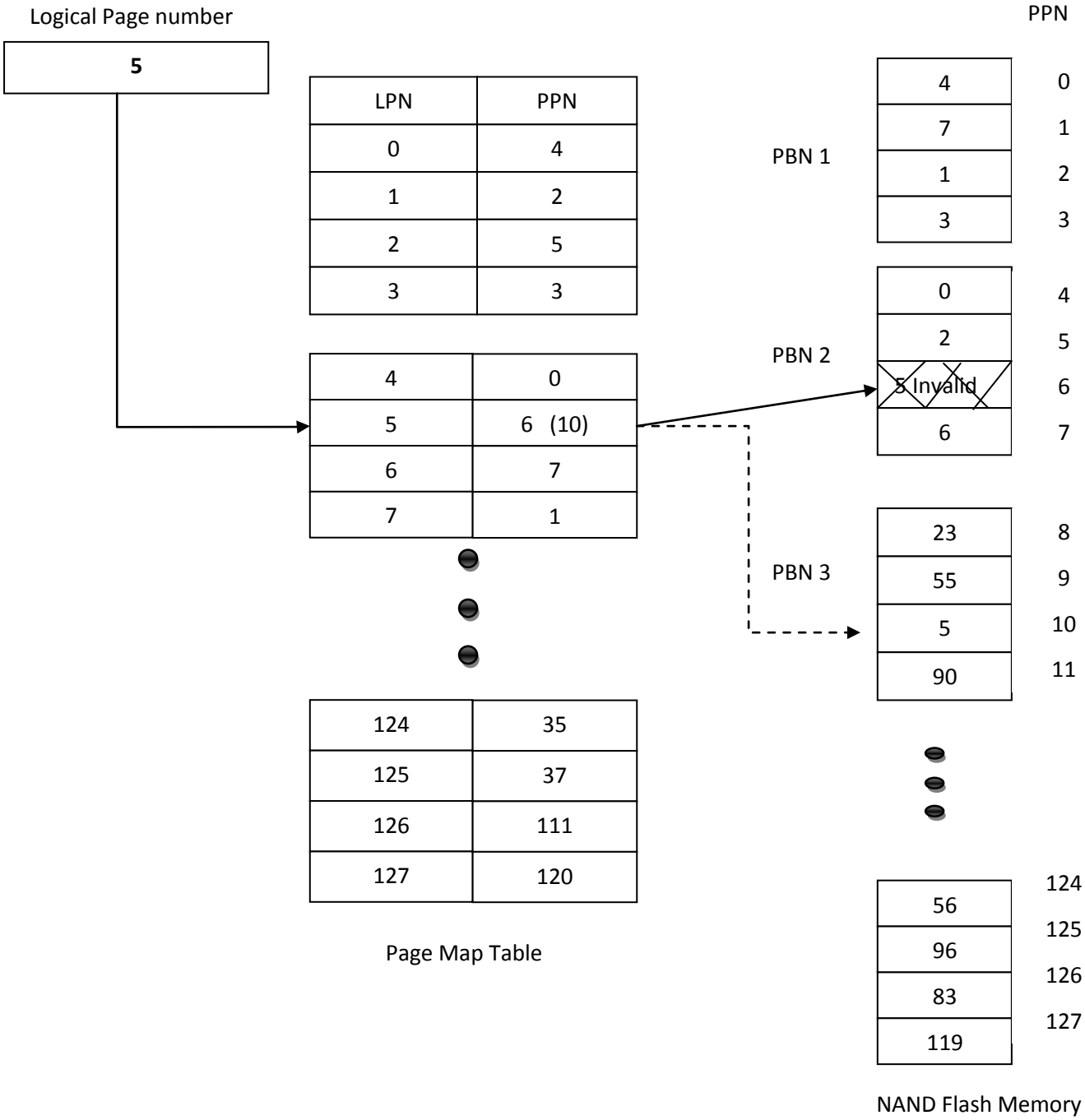


Figure 1.5: Page Level Mapping

In page mapping, NAND flash memory is managed on a page basis. Page mapping scheme maps each logical page to physical page. Here a page map table is constructed and maintained in RAM. A map table entry consists of an LPN (Logical Page Number) and a PPN (Physical Page Number). When a write request arrives physical page number associated with logical page number is searched form the page. If the page found in mapping table and is already written with some value, then data cannot be written to the same page. Requested data is written to the one of the free page and entry in the page table is modified corresponding to new page to correctly read data and previous page is invalidated.

For example (see Figure 1.5), when a write request to the logical page address 5 arrives to the FTL, the FTL first searches for the corresponding physical page number by using the logical page number as index of table and it find physical page number 6 in NAND Flash memory meaning of it is that the physical page number 6 is already written to. Hence, the requested data should be written to page 6. It should be written to a free page in flash memory. Here the third page of the physical block number 3 (physical page number 10) is free, the data is written to that location. At the same time, the corresponding map entry is updated to point to the new page with valid data. The page mapping scheme has the advantage that it writes data to any free page in flash memory, which enables more flexible storage management

Page-level FTL schemes can allocate the pages within a block sequentially without recording the page status (valid or invalid) in its spare area. Therefore, it is still usable to MLC flash. However, page-level FTL is unsuitable for large sized MLC flash due to the large address mapping table. How to reduce the size of the address mapping table is a crucial issue. Based on page-level FTL, DFTL scheme, stores the address mapping table in flash memory and only caches a small amount of active mappings in RAM. It effectively reduces the RAM cost, however, it incurs extra valid page copies when maintaining the address mapping table in the flash memory.

2) Block Level Mapping

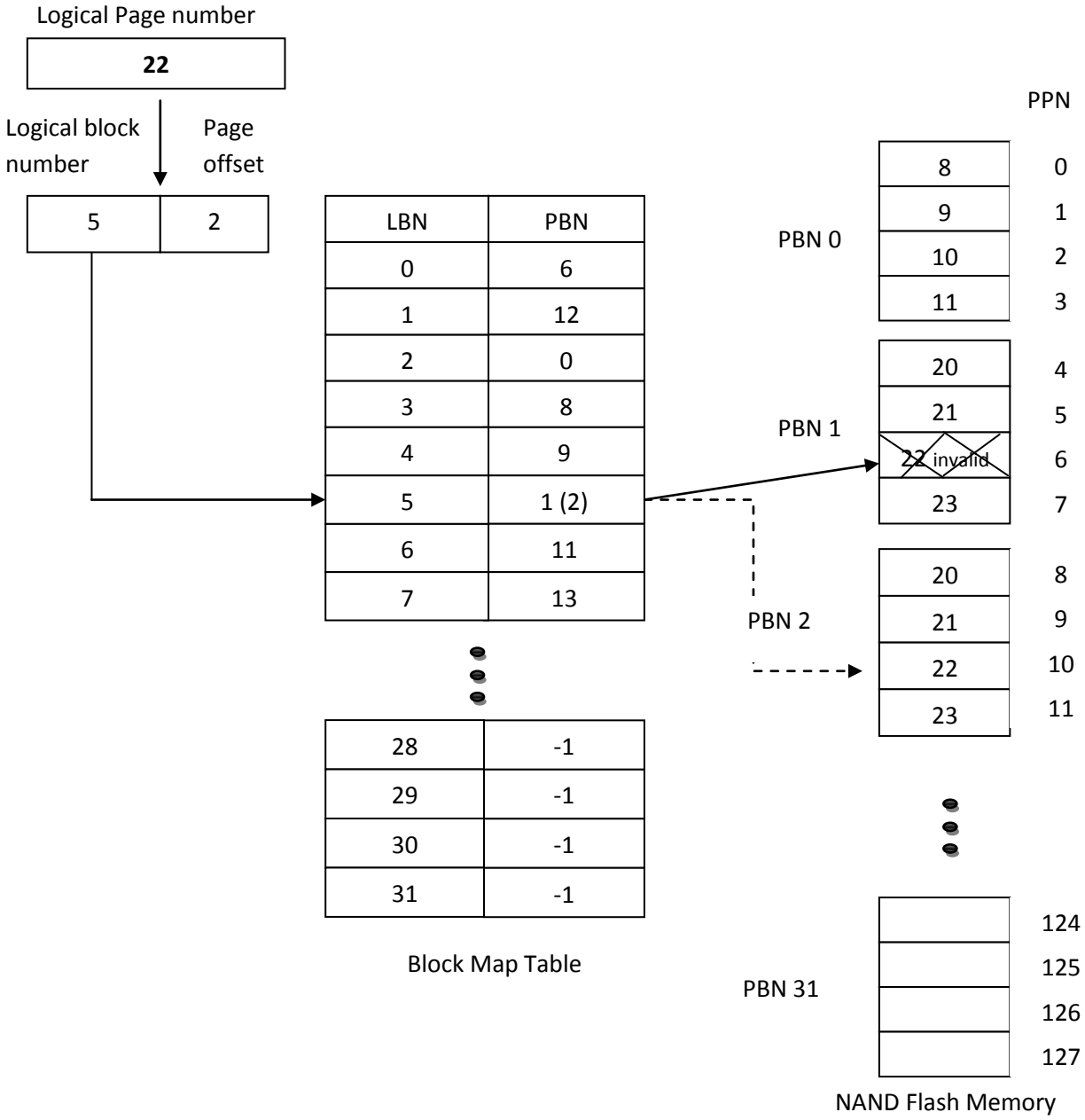


Figure 1.6: Block Level Mapping

In block mapping, the logical page address is divided into a logical block number and a page offset. When any read or write request arrive it first find logical block number and offset using equation 1.1 and 1.2. The logical block number is used to find physical block associated with

that logical block and the page offset is used as an offset to locate the free page in the corresponding block. As the map table consists of block number entries, its size can be reduced.

$$\textit{Logical block Number} = \textit{logical address} / \textit{no of pages per block} \quad (1.1)$$

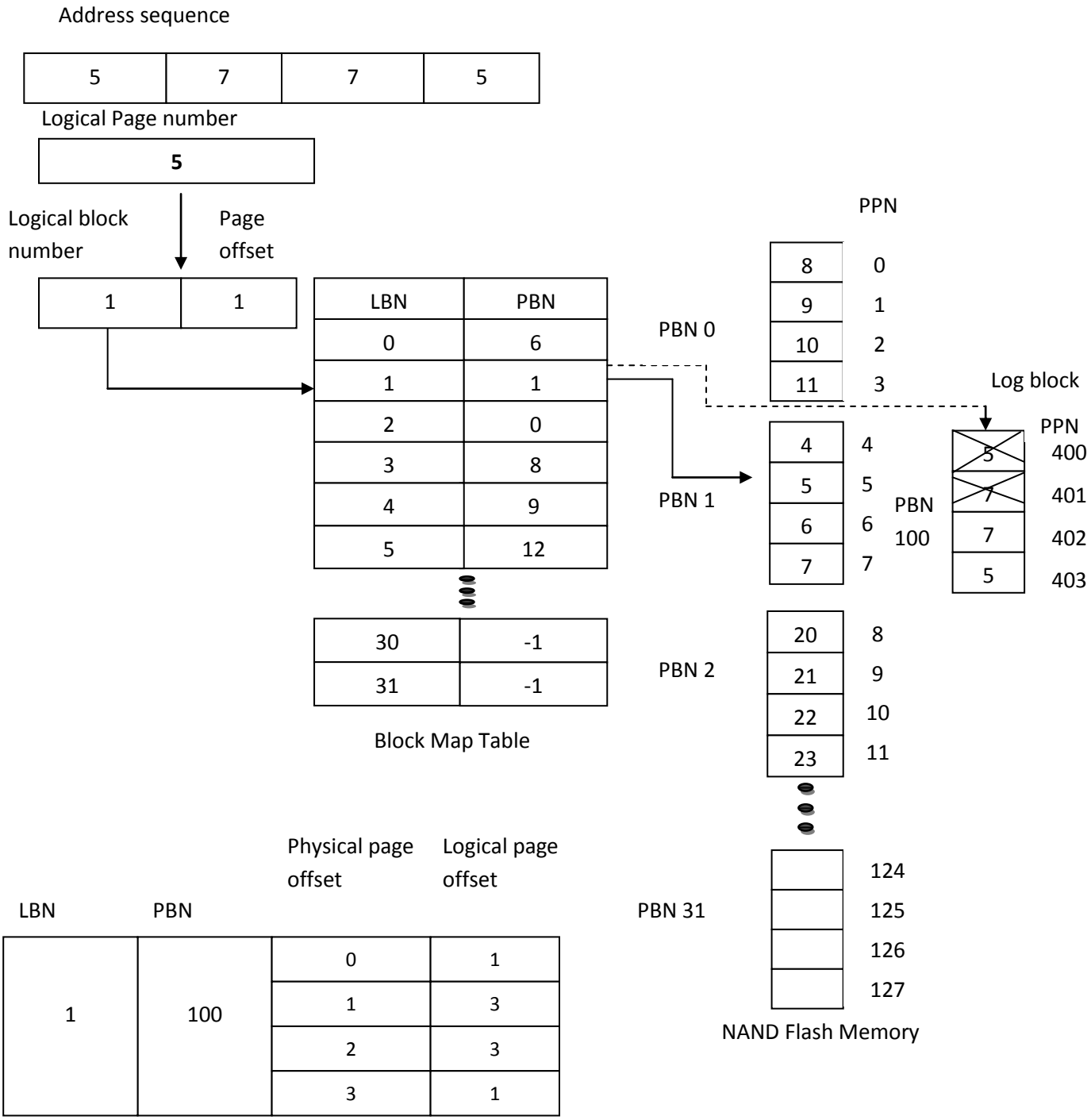
$$\textit{Page offset} = \textit{logical address} \,\%\, \textit{no of pages per block} \quad (1.2)$$

However, block level mapping require that each logical page to be written to a fixed offset of a physical block, and thus limits the flexibility of the page placement. This results in lower space utilization of flash blocks, which is defined as the ratio of the number of occupied (i.e., non-free) pages in a block to the total number of pages in a block when the block is going to be erased. As a result, every overwrite operation to the same location may incur a frequent block-level copy operation. For example (see Figure 1.6), when a write request to the logical page address 22 is arrives to the FTL, the logical page address 22 is divided into a logical block number 5 and a page offset 2. The physical block number 1 for the corresponding logical block number 5 is found from the mapping table. After the corresponding physical block number 1 is matched, the logical page offset is added to the physical block number found, and hence the incoming data is to be written to the physical page number 2. However in our case, the physical page number 2 in physical block number 1 is already written to. Therefore, the data is written to a free block in our case physical block number 2 is free. So update data is written at page 2 in physical block 2.

Block-level FTL schemes use the block offset to locate the pages within a block, and the pages may be programmed randomly within a block. Therefore, block-level FTLs may not be applicable for MLC flash.

3) Hybrid Level Mapping

In Hybrid mapping technique few blocks from the NAND Flash memory serve as log blocks. Here log blocks use the page-level mapping scheme and the data blocks are handled by the block level mapping. The maximum number of log blocks is given to limit the size of page-level mapping table. A hybrid mapping scheme known as the log block scheme was first presented by Kim et al. [2002].



Page level mapping table for log block

Figure 1.7: Hybrid Level Mapping

The key idea of the log scheme is to maintain a small number of log blocks in flash memory to serve as write buffers for overwrite operations. When write request arrive it first find logical block and offset using equation 1.1 and 1.2. The physical block number for the corresponding logical block number is found from the mapping table. After the corresponding physical block number is matched, the logical page offset is used to find page from block and check whether data are already written there or not. If data are already written to that page the log block scheme allows the incoming data to be appended continuously into log block as long as free pages are available in the log blocks in order of its arrival. For example, as shown in Figure 10, the physical block number 1 contains the data of logical page numbers 4, 5, 6, and 7. When the upcoming write requests are to logical page numbers 5, 7, 7 and 5 arrives, they are written to the log block number 100. As a result, only the last requests to logical page numbers 7 and 5 are valid for logical block number 1. These requests are represented as 5" and 7" in the figure 1.7. So here log block contain four pages and their mapping information stored in page map table as shown in figure 1.7.

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

Chapter 2

Literature Survey

2.1 Previous Works:

FTL schemes have been proposed to improve the performance of flash memory. There have been many researches on log block-based FTL. Ban proposed the replacement block scheme based on the concept of a replacement block [7]. In this scheme when update request arrives new log block is allocated and page is written into the log block. This is in-place scheme in which the logical page number is always equal to the physical page number in the block. When an update request arrives it allocates a new log block if the write cannot be accommodated in the existing data block and log blocks. Replacements block scheme use block level mapping scheme. When no log blocks are free to accommodate upcoming write request, garbage collection choose the data block which has the largest number of log blocks and all the valid pages of data block and log blocks are copied into the last log block. Than last log block become new data block and all other blocks are erased to create free space to accommodate upcoming write request. Problem with this approach is that when merge operation is performed all the log blocks are not fully occupied. Only some of pages are frequently updated in that case log block contains only frequently updated pages and rest of space of log blocks are unutilized, due to that it suffer from poor space utilization of log blocks**.** In fact if update request are random, it perform frequent merge operation which degrade performance of FTL.

To overcome the problem of replacement block scheme, Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min and Yookun Cho have proposed *block associative sector translation* (BAST) scheme [6] that use dedicated log block for storing updates of data block until log block become full. In BAST page are written in log block in an order of it arrival. It generally uses the block mapping to translate the logical address, and uses the page mapping with log scheme for frequently updated block. When log block is full it performs merge operation. If all pages in log

blocks are written in a place then it perform switch merge operation otherwise it perform full merge operation. Here one log block is associated with one data block and for each log block page table is maintain. Problem with this approach is that only few blocks are used as log blocks.

When update request arrives and no more log block is free to accommodate incoming request than garbage collector select victim log block and merge it with corresponding data block and free one log block to accommodate update request. This approach suffers from two performance problems. For example assume that there are four pages per block and there are four data blocks and two log blocks. If the write sequence "P11, P5, P8, P2, P10, P7" arrives form upper layer. Here we assume that all data blocks are filled. Note that P11, P5, P8, P2 all are belong to different data blocks because we have four pages per block. Here write request for P11 and P5 are written in two separate log blocks because they belong to different data blocks. But when request for P8 is arrive one of the log blocks must be freed to accommodate write request of page P8 because it is not belong to any of the present log block. But at the same time current log blocks contain only one page when merge operation is performed. Here we cans say that for above write sequence BAST will perform expensive merge operation on every write request. Such a phenomenon where most of the writes requests invoke block merges is called log block thrashing. Moreover every victim log block contain only one page and other three remain empty. BAST would show very low space utilization. Here we can conclude that when request pattern is random 1:1 mapping scheme of BAST show poor performance.

To solve the problem of BAST, the fully associative sector translation (FAST) scheme has been proposed by Lee in 2007[2]. In this approach log block is shared by all the data blocks. Whenever any update request arrives for any block it is written in a current log blocks which effectively improve storage utilization of log blocks and delay merge operation much longer. FAST prevent log thrashing problem and even low space utilization problem of BAST. Here in this approach log blocks are divided into two areas: one log block is used for sequential write and other log blocks for random writes. If data having sequential write pattern they are stored in sequential log block. And if data having random write pattern then they are stored in random log blocks. Here main aim of dividing block into sequential and random log block is that work load arrive form

upper layer may contain large number of sequential writes; in that case it is likely that many switch operation arise. However in FAST one log block is shared by all data block so it cannot take an advantage of switch merge unless sequential writes are handling separately from random writes. So to handle sequential write FAST allocate dedicated special log block for sequential request.

Whenever update request arrives page is directed to sequential log block if page offset calculated from equation 1.2 is zero (i.e. first page in a block) or log block is filled with the pages which are sequentially written from beginning offset and upcoming page is next in sequence**.** If update request is not candidate for sequential log block it is written into log block for random request. If no more empty space found in either of the log block merge operation is performed. If no more space found in sequential log block Switch merge operations performed by just changing the log block with data block and erasing old date block because it has invalid data. Switch operation required just one erase operation. If no more space found in log block for random request it select one of the log block as victim and merge victim with its corresponding data blocks and erase victim log block and all the data blocks whose corresponding pages found in victim log block. If in victim log block contain pages of n different data block, total erase operation performed are n+1. FAST scheme maintain block level mapping table for mapping of logical block to physical block. Problem with FAST is high block associativity. When log block is merge with corresponding data blocks it copy all the valid pages form log block and corresponding data block to newly allocated free block. If there are N pages per block and in worst case it may happen log block contain the all pages form different block, then we can say log block is associated with N data blocks. In that case maximum merge cost for log block L is as follows:

$$N*N.*C_c + (N+1)*C_e \qquad (2.1)$$

Here C_c is cost of copy and C_e is cost of erase and N is number of page per block. So in worst case N^2 copy and (N+1) erase operation are required. In general we can say if total K numbers of data blocks are associated with log block then merging cost would be

$$K*N*C_c + (K+1)*C_e \quad (2.2)$$

Here we can say as associativity of data blocks in log blocks increase, numbers of merge operations are also increase. Main Reason for increasing associativity is small random belonging to different data blocks.

In FAST scheme large associativity of data blocks with log block degrade performance of FTL, to reduce associativity of data blocks with log block set associative sector translation (SAST) scheme have been proposed by researchers [8][9][6]. In the SAST scheme [8][9] set of log blocks are associated with set of data blocks. For example set of N log block can be used by set of data block consist of K number of data blocks. In superblock scheme [8] several adjust logical blocks are combined to construct a superblock. For every superblock K numbers of log blocks are allocated where value of K is changed dynamically. Block level mapping table is used for superblock but it allows logical pages within a superblock to be freely allocated to any of the allocated physical block so it required page mapping also. It uses three levels in page mapping table. Superblock scheme also use spare area to store page mapping information to reduce RAM requirement for mapping information. When update request arrives log block is allocated and data are written in a log block in order of arrival. Superblock scheme exploit block level spatial locality by combining consecutive logical blocks into a Superblock. It maintains page-level mappings within the superblock which exploit block level temporal locality. Superblock scheme can easily control degree of sharing by adjusting size of super block according to block level spatial locality. Here SAST scheme [8] [9] is compromised version of BAST scheme and FAST scheme, so it cannot solve block thrashing and high block associative problem completely.

Table 2.1: Comparison of Merging cost of various block level FTL

Algorithm	Merging Cost in Worst Case	Merging Cost in Best Case
BAST	$N*C_c+2*C_e$	$1*C_e$
FAST	$N*N*C_c+(N+1)*C_e$	$1*C_e$
KAST	$K*N*C_c+(K+1)*C_e$	$1*C_e$

Hyunjin Cho, Dongkun Shin, Young IkEom have proposed Log Buffer based Flash Translation Layer using K-Associative Sector Translation(KAST)[4]. The KAST scheme basically minimizes the associativity of log block which is full in case of FAST and 1:1 in case of BAST. Here when write request of different logical block arrives they are distributed among different log blocks. And each log block enforced to associate only with the K number of data block at maximum. As a special case sequential log block is also associated with data block like in case of FAST. When write request arrives first it check whether it is for random log block (RLB) or sequential log block (SLB). If the page offset within data block is zero than request is candidate for SLB. But there is possibilities other pages for same blocks are already written in RLB. There are two possible way to handle this is either write a page to RLB without wasting SLB or allocate a new SLB. For new SLB either copies of all the pages which are in RLB for this particular block in to SLB and invalidate them into RLB or keep those pages in RLB and wait for update request for the pages. If page offset is not zero but it is same as next expecting page number of SLB, in that case also page is written in SLB. But again here it is possible that there is a gap between last written page and current page. If gap is small then fill the gap and append the page at end. But if gap is large, KAST use two policies it perform partial merge on SLB or second it transfer SLB into RLB by assuming that there is little possibility of sequential write request. It is better to use log block for on-going write request instead of performing partial merge operation. If page is not candidate for SLB it is written in RLB. To write page in RLB first it search log block which contain page for the corresponding block and write page in that log block. If no log block found for corresponding block than page is written in any block which is having a free space whose block associativity is less than K. If RLB has free page but associativity is not less than k in that case associativity may increase also. When no more free space found in log block merge operation is performed. Here block which is having a low block associate is selecting as victim and perform merge operation. Here every log block is associated with normally K data blocks so it require (k+1) erase operation and (k *N) copied operation where N is the number of block associated with log block. When value of k=1 then KAST is similar to BAST but it avoid thrashing problem of BAST. And when value of k=T where T is block associated with log block and T>K, it behave like FAST, but cost of merge operation is less compare to FAST. Here we can conclude that KAST is better that FAST and BAST. Comparison of merging cost of BAST, FAST and KAST is given in Table 2.1.

Chapter 3

Design Approach and Implementation

3.1 DESIGN OF GroupFTL

In this section, an efficient page-level NAND flash translation layer, called GroupFTL, is discussed which can be used with both MLC as well SLC Flash Memory. In our scheme, group of pages is formed in which concentrated mapping is deployed and limited RAM space is taken. Detailed write and read operations in our GroupFTL are presented based on the page-level address mapping scheme (Section 3.1.1).

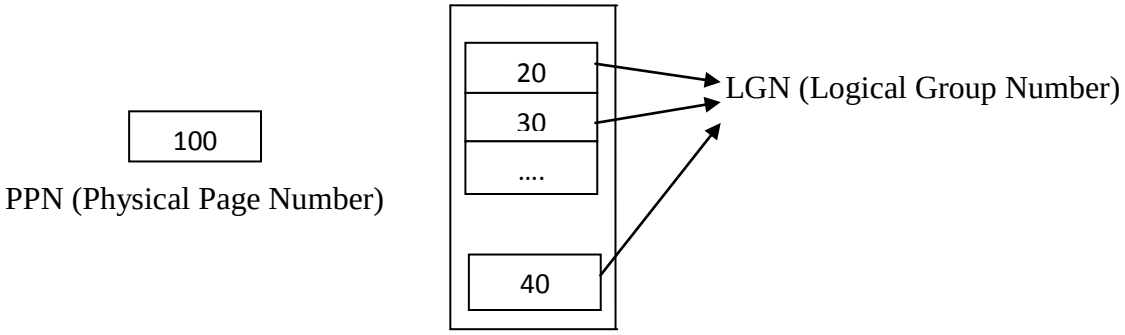
3.1.1 PAGE LEVEL ADDRESS MAPPING FOR GroupFTL

In GroupFTL, each logical page number (LPN) is translated to a logical Group number (LGN) and a Group offset (GO) as shown in Figure 3.1. Each logical Group can be mapped with *any number of physical* blocks as in the case of Pure Page level FTL. GroupFTL adopts the page-level mapping and uses page as unit to manage mapping entries. It exploits sequentiality in the write requests to create a concise representation of in-cache mapping table pages. As GroupFTL has page level mapping it gives freedom to map any logical page to any physical page. We need to record the mapping for any logical page in cache (SRAM/DRAM).

In GroupFTL we have made groups of logical pages and these each group is stored in the spare area of the physical page in Flash Memory when data is written in the Data area of the physical page, and the page containing the sub-table (i.e. the group of pages stored in spare Area) is stored in the GMT (Global Mapping Table) stored in the Cache (RAM).

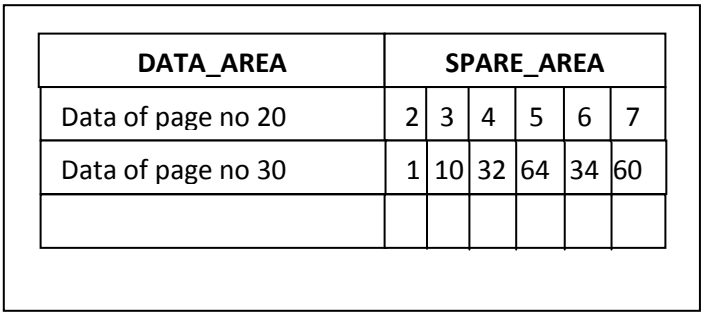
Hence if we have 100GB Flash Memory then No of pages is 26214400 and if we form a group of 50 Logical Pages then the GMT has 524288 entries in RAM. These entries contain the pages which contain the Physical page number whose spare area has the sub-table for that group.

In GroupFTL the sub-table is written in spare area along with the data which is written in the Data area of the page, hence writing the spare area does not include any cost of overhead. The LGN of the LPN can be found out by dividing the LPN by Group size (e.g., 50 entries).After getting the LGN we can read the sub-table from Flash memory from the physical page number obtained by us using LGN from GMT. From the sub-table read, the required physical page number is obtained using the GO (Group Offset) which is obtained as remainder while dividing the LPN by Group size.



GMT (GLOBAL MAPPING TABLE)

Figure 3.1(a)



NAND Flash Memory

Figure 3.1(b)

Figure 3.1: Mapping scheme of GroupFTL.

In this way, we can get the actual physical page number directly by reading the spare area of the physical page present in the GMT while limited RAM space is taken when doing address translation and we can exploit all the benefits of Page Level FTL, the most optimal one.

3.1.2 READ AND WRITE OPERATIONS

A write request issued from file system is represented by a data and a logical page number (LPN) and number of pages to be written, e.g., write (35, 2, AB). Given the LPN, when divided by the Group size, the quotient obtained is the logical Group number (LGN), and the remainder is the Group offset (GO). After the translation from logical page number to logical group number, the first write to a given logical page is written to the page next to the page pointed current page pointer (PPN).PPN is the Global variable which points to the physical page in Flash memory on which next data can be written. Initially the pointer (PPN) is NULL, hence a physical block is assigned and first page number of that physical block is copied into PPN. If the pointer (PPN) is not NULL, the write operation is performed on the page next to the page pointed by PPN. When PPN value crosses the last page of the currently assigned physical block a physical block is assigned. Thus once a physical block is mapped, pages in the assigned physical block are allocated sequentially no matter it is a write operation or an update operation.

1. *Algorithm 1: write(lpn,num_pages,data)*
2. *while(num_pages> 0)*
3. *global_entry = GMT[lpn / PMT_ENTRIES].PMT;*
4. *pmt_offset = lpn % PMT_ENTRIES;*
5. *if(global_entry == MAX(unsigned long int))*
6. *create new spare-table*
7. *else*
8. *read the spare area of the page specified by global_entry*
9. *pages_in_gmt_entry = PMT_ENTRIES - pmt_offset;//free pages in the currently assigned block*

10. *pagecnt = (pages_in_gmt_entry>= num_pages) ? num_pages : pages_in_gmt_entry;*
11. *find the next physical page to write the data*
12. *firstpg = nextpg;*
13. *if((nextpg % PAGES_PER_BLOCK + pagecnt-1) >= PAGES_PER_BLOCK)*
    a. *pagecnt = PAGES_PER_BLOCK - nextpg%PAGES_PER_BLOCK;*
14. *for(spare_offset = pmt_offset; spare_offset<pagecnt + pmt_offset; spare_offset*
15. *If there was an entry in sub-table array previously then decrement the valid page count of that physical page*
    i. *Increment valid page count of the current block.*
    ii. *Update the sub-table array in spare area obtained at step*
    iii. *Write the data of num_pages = pagecnt from the page specified by firstpg with the updated spare area at last page*
    iv. *Update the GMT(GLOBAL MAPPING TABLE)*
16. *lpn = lpn + pagecnt;*
17. *dataoffset+=pagecnt;*
18. *num_pages = num_pages - pagecnt;*

Figure 3.2: Algorithm for Write operations in GroupFTL

After *N (i.e. NO_OF_PAGES_PER_BLOCK)* number of writes, the physical block becomes full (step 13 in Figure 3.2), a new free physical block will be allocated. When a new page is mapped, the latest version of page mapping sub-table (which includes the requested GO) will be read out from the spare area of the page pointed to by pointers LGN. The corresponding mapping slot will be updated and then written to the spare area of the new page, together with the requested data written in the data area. The time taken for a write request is one spare area read and one page write i.e. (T*_sparerd + T_pagewr)*, if a free page in currently assigned physical block is available.

Consider an example for write operation in GroupFTL. Assume each Group has 8 pages. Initially Global mapping table is empty. For the first write request write (45, 2, AB), the corresponding LGN and GO are 5and 5 respectively. A new free block, suppose PBN=11, is allocated, and the

data A is written into the data area of the first free page PPN=88 and data B is written into data area of PPN=89 .Page Mapping Sub-table is updated by writing the PPN at the GO obtained above and is stored in the spare area of page PPN=89 with data area which is filled with Data B. The Next page pointer PPN in mapping table is incremented after writing the data. After 8 writes, the physical block PBN=11 becomes full, a new free block PBN=17 is allocated, and the data are written sequentially into the pages of the Block 17.

Garbage collection is invoked to free some of the blocks when SSD is idle or when certain threshold conditions are attained. A Garbage collector chooses a block from all the used ones and the one which has least number of valid pages, copies them to another block and performs the erase operation on the chosen block. The freed block is then attached to the list of free blocks.

Now Consider a Read request issued from file system e.g. *Read (45, 2,* ***Data)*** which is represented by a logical page number (LPN). This LPN is then converted into corresponding LGN and GO. The LGN obtained will be first searched in the Global mapping table. The page mapping sub-table is read from the physical page number obtained from LGN. With the requested GO we can be get the physical page number which contains the requested data. The time overhead for one read request is one Spare Area read and one Page read: i.e. (*T_sparerd+ T_pagerd).* For the above example Read (45, 2, data), we get LGN = 5 and GO =5.As GO=5 we read Spare Area from the LGN entry 5 and required Physical Page Number = 88 can be obtained from the Sub-table read using the GO = 5.

Then we increment the GO and check whether the next page (i.e. 46) which we want to read is written after the previous page (i.e. 45), if it is the case the we increase the repeat and continue the procedure until the num_page to read is greater than zero or the sequentiality is broken, In our case we obtain Physical Page number 88 from Sub-table at offset 5 and 89 at offset 6. Hence we send request for two pages to read from page number 88 to the Flash memory. We obtain the valid data *AB* copied into the buffer **Data** after reading data from page number 88 and 89.

1. *Algorithm 2: read(lpn,num_pages,data)*
2. Whilenum_pages> 0
3. global_entry = GMT[lpn / PMT_ENTRIES].PMT;
4. pmt_offset = lpn % PMT_ENTRIES;*pba=ppns/(NO_OF_PAGES_PER_BLOCK);*
5. *Read spare area of the page specified by global_entry.*
6. *ppn = physical page number from sub table array with index specified by the pmt_offset.*
7. *Initialize repeat with 1.*
8. *Increment the pmt_offset.*
9. *tpn = ppn++.*
10. *While pmt_offset< PMT_ENTRIES && repeat <num_pages*
11. if(ppn == sparebuffer.subtable[pmt_offset])
    i. repeat++;
12. else
    i. break;
13. ppn++;
14. pmt_offset++;
15. *Read the data from the number of pages= repeat, specified by tpn.*
16. lpn = lpn + repeat;
17. num_pages = num_pages - repeat;

Figure 3.3: Algorithm for Read operations in GroupFTL

3.2DESIGN OF CACHEFOR GroupFTL

In this section, a caching scheme implemented for GroupFTL is discussed. The cache uses the uthash library in C for performing hashing functions. This library performs hashing on the cached data so that the searching of a page in the cache require less time. This increases the overall time required for the read and write operations.

3.2.1 CACHING USING Uthash [11] library

The caching layer lies above the Flash Translation Layer(FTL). The requests coming from the System first goes to the cache buffer. In case of read requests, the page is searched in the cache. If the page is found in the cache the data is returned from the cache. If the page is not found, the page is read from the Flash memory by issuing the corresponding ftl_read request to the FTL. In case of write requests, the page is searched in the cache and if the page is found, it is updated in the cache itself. If the page is not found, the page is added in the cache until the number of pages in cache exceeds the maximum cache size. When the number of pages in cache exceeds maximum cache size, the part of the cache is flushed. In flushing the cache, the number of pages that are flushed is in multiple of number of pages per block of the flash memory. By doing so, a new block can be assigned and all the pages that are flushed can be written in newly assigned block in a single write request. Thus whenever the cache is flushed, the number of ftl_write requests going on to the flash memory is equal to the quotient of division of number of pages to be flushed by the number of pages per block of flash memory.

Hence, if 256 pages are flushed and there are 128 pages per block, then the number of write requests going on the flash memory is 256/128=2. Thus, as the number of write requests going on the flash decreases due to cache, the cache increases the overall speed and efficiency of the flash memory.

We have seen that the cache decreases the input/output requests passing on to the flash memory thus increasing its efficiency. But each time the request comes, the requested page is searched in the cache. The time required to search the page is proportional to the size of cache. When the size of cache increases, the time required to search the page increases. Thus the caching algorithm must take care that the time required in searching the cache is not too large.

In order to address this issue, we have performed hashing of pages stored in the cache. The uthash library, which we have used, provides routines for hashing. The pages in the cache contain two fields namely logical page number (LPN) and the data. The logical page number

(LPN) is used as index (key) to perform hashing and searching. The following functions of uthash library are used by us.

Table 3.1: Functions in Uthash Library

NAME OF FUNCTION	FUNCTIONALITY
HASH_ADD_INT()	Add a page in the cache
HASH_FIND_INT()	Find a page in cache on the basis of lpn
HASH_DELETE()	Remove a page from the cache
HASH_CLEAR()	Remove all the pages from cache
HASH_ITER()	Iterate in the cache
HASH_COUNT()	Returns the number of pages in cache

These functions performs hashing of data stored in cache using the logical page number (LPN) as index (Key) and reduces the search time of the cache.

When any request comes HASH_FIND_INT() is called. In case of read operation, if the page is found, then the page is returned. If page is not found, page is read from flash and is returned. In case of write operation, if the page is found, the data is updated, the page is removed and updated page is added again using HASH_ADD_INT(). If the page is not found, the page is added using HASH_ADD_INT(). If the HASH_COUNT is greater than MAX_CACHE_SIZE, pages are deleted from cache using HASH_DELETE and are flushed on to the flash translation layer (FTL). The flash translation layer writes these pages on the flash memory. The ftl_write function is called once, when pages equal to number of pages per block are to be flushed.

Thus our caching algorithm reduces the number of input/output requests passing on to flash memory resulting in increasing the efficiency and life of flash memory. Moreover hashing of data results in decreasing the searching time of cache and thus increasing the speed of serving the read write requests than general searching methods.

Chapter 4

Analysis

Analysis is very important in evaluating the performance of any algorithm. Proper analysis makes the way for devising an algorithm that is better and efficient than the existing one.

We have done analysis of various existing algorithms like BAST, FAST and MNFTL as FTL algorithm and we compared these existing algorithms with the one proposed here GroupFTL (Flash Translation Layer).

Below we present the evaluation methodology and metrics used for the evaluating the algorithm.

4.1METHODOLOGY OF EVALUATION

In developing an FTL, there are two main issues that should be considered carefully. One is a correctness issue that addresses the functional correctness of the designed FTL and second is performance analysis. In order to verify the correctness of FTL, test based on formal verification based methods can be used. In test case based method test cases are generated and given to a flash storage to check the correctness of FTL. Here the test cases (or traces) can be synthetic that means they don't need to be realistic because they are used for the correctness verification. In that respect, the random generation of test cases (or traces) seems to be adequate and can be generally used even though completeness of testing (test coverage) cannot be guaranteed. To verify the correctness we have generated many synthesized traces and run on our simulator and check content of mapping table and other parameters. For example, the read content of a sector must be identical with the data which are written to the sector if a power failure does not occur during the write. The NAND flash storage can be a real-system or a simulated system which simulates the flash memory states based on flash memory operations and faults like a power failure. The other issue in developing an FTL is concerned about performance enhancement

which has been main theme in flash memory research. In order to compare the performance, realistic traces which reflect the real world usage are necessary. We have also used realistic traces to evaluate performance of proposed algorithm.

It is found that workloads in general computing system and in enterprise is very complex. These workloads contain very complex access patterns. They may contain sequential request as well as random request patterns and may also contain many random requests which are interposed between sequential requests. IBM Researcher found that the I/O traffic in servers and personal computers are bursty and appears to exhibit self-similar characteristics. Workloads may contain temporal locality and sequential locality. To support this fact we have analyse distribution of LSN with reference to logical time for first 10,00,000 I/O request of different traces is as shown in figure 11. Access pattern general purpose application running on computer is shown in figure12. From the figure 11 & 12 workloads are very complex and there is strong temporal locality in work load also. From the diagram we can also see that there exists a spatial locality too.

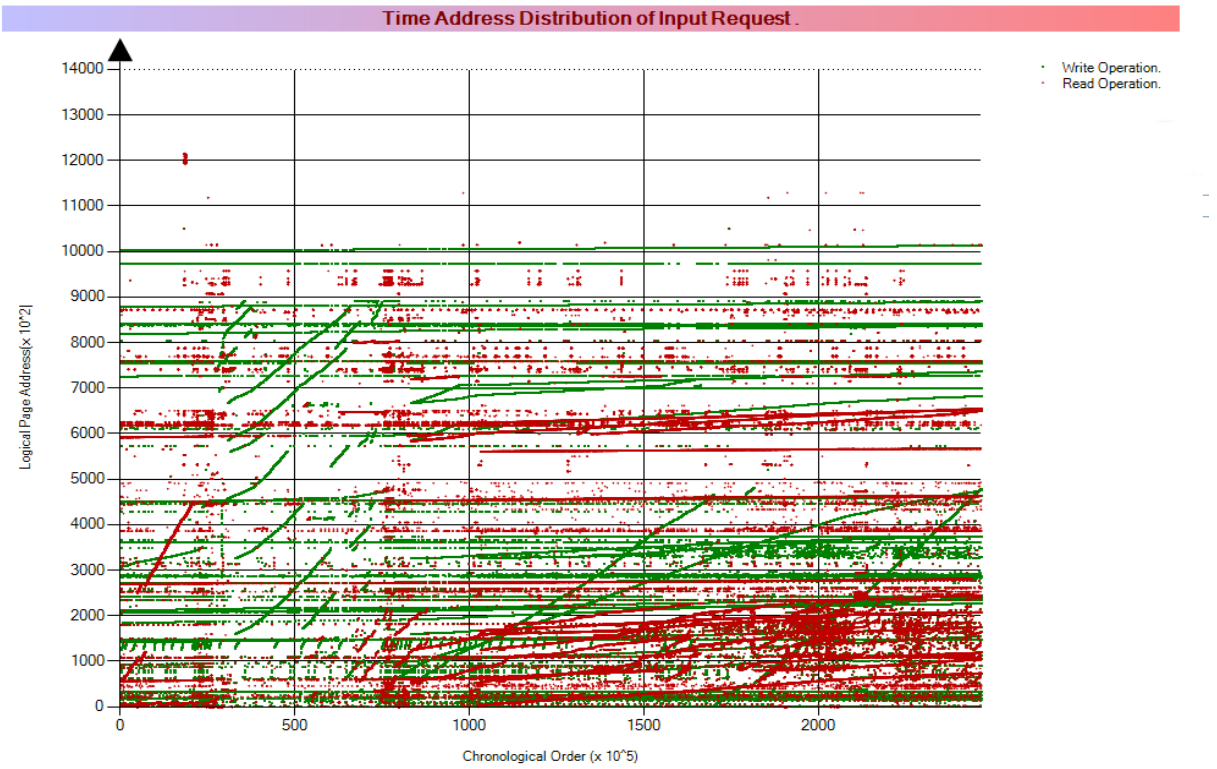


Figure 4.1(a): Financial1

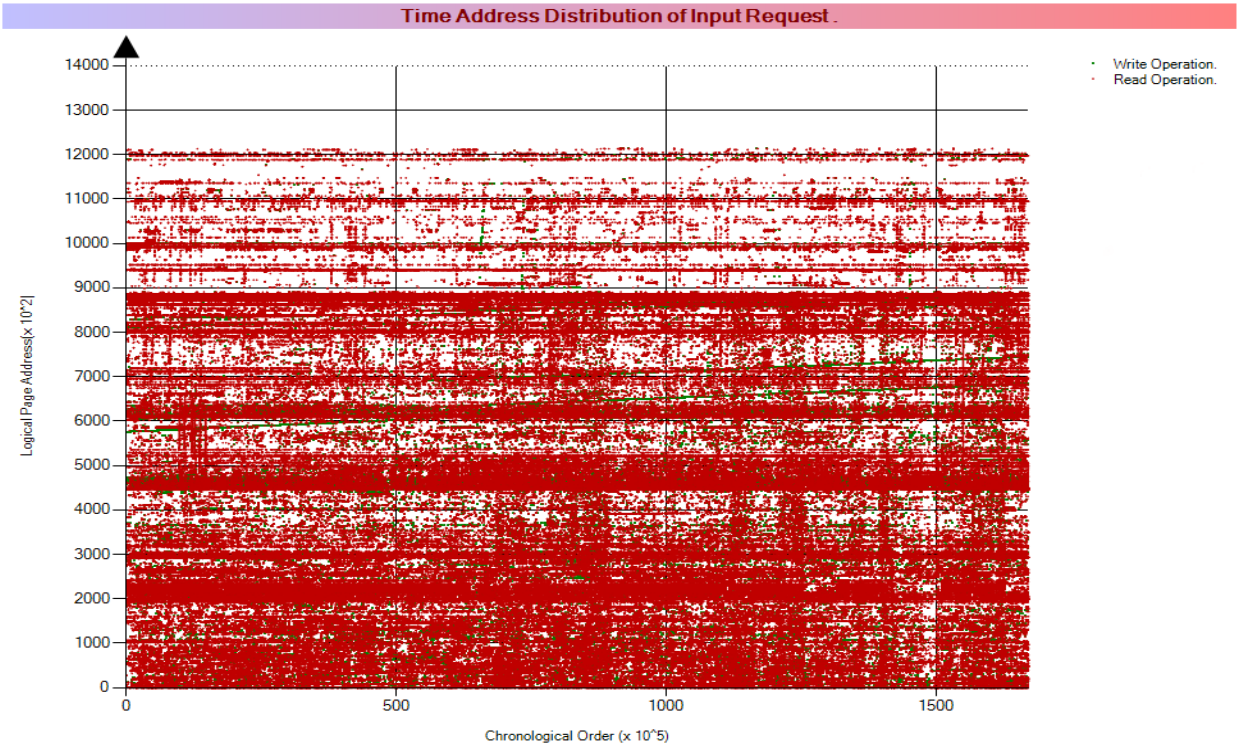


Figure 4.1(b): Financial2

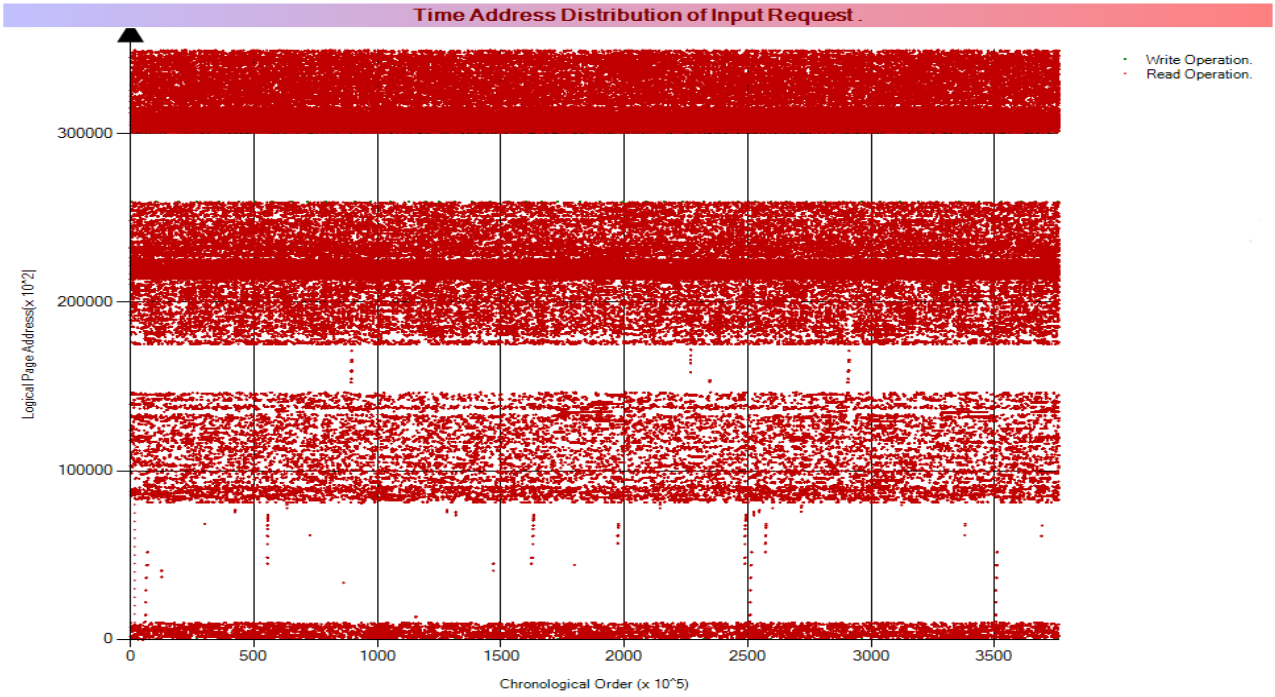


Figure 4.1(b):Web search1

Figure 4.1: Time Address distribution of input Requests

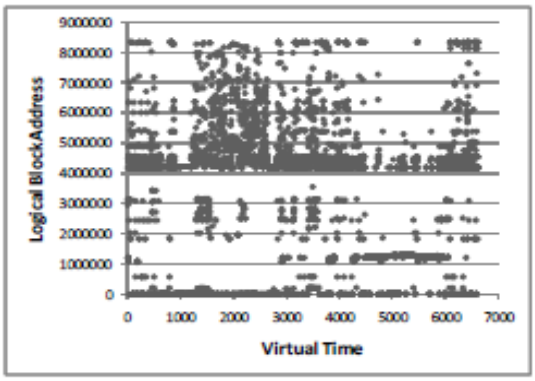


Figure 4.2 General-purpose applications

Table 4.1 Characteristics of Selected REAL-WORLD Workloads

Trace	Read/write	Unique pages	Address updated more than once	Locality%/%	Max page number
Financial 1	23%/77%	57%	13%	80%/20%	1215634
Financial 2	79%/21%	18%	8%	61%/30%	1213864
Web search1	100%/0%	25%	15%	70%/35%	34967796

We have used real-world and synthetic traces to study the impact of existing FTL. We have received traces namely Financial 1, Financial 2 and Web search1, from an OLTP application running at a financial institution made available by the Storage Performance Council (SPC), henceforth referred to as the financial trace [10]. Here read/write ration represent the percentage of read and write requests present in the traces. The locality expression 80/20 means that 80% of total number of accesses call is intensively performed in a certain 20% of the NAND flash area.

4.2 PERFORMACE METRICS AND EXPERIMENT RESULTS

We will focus on four parameters, the block utilization which indicates the average number of pages that have been used when a block is erased, the average response time, the write response

time, and the number of erase operations which is the costliest operation hence degrading the response time and which will also directly affect the life span of the flash memory.

1) Block Utilization

Block utilization Metric indicates the average number of pages that are used when a block is erased. As BAST and FAST are Block Level FTL the block utilization is less compared to the Optimal FTL (The Ideal Page Level FTL). As shown in the Figure 4.3 the Block utilization of the GroupFTL is same as the Optimal FTL, because GroupFTL is also Page Level FTL. Block Level FTL's have less block utilization as they write pages in the block at the block offset and hence situations may arise that when a block is to be reclaimed by the Garbage Collection and all the pages are not written in the Block.

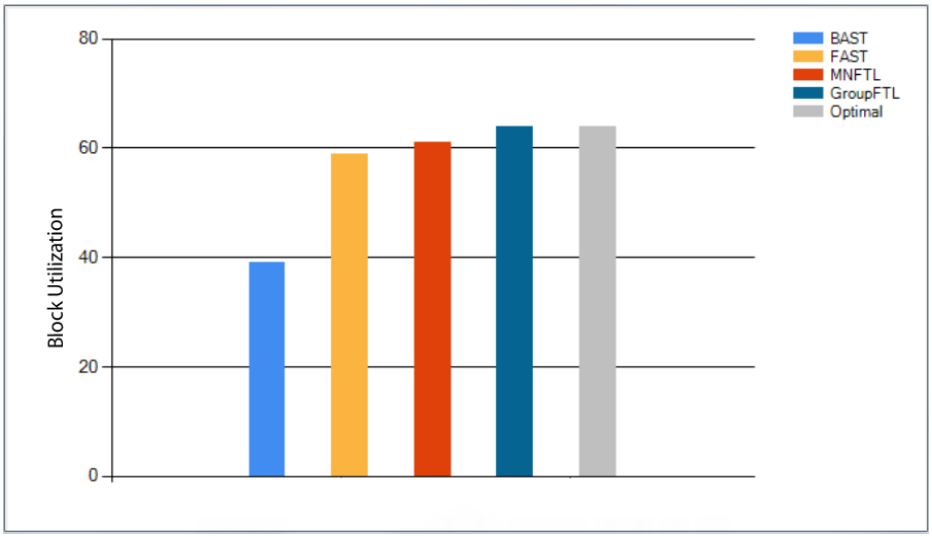


Figure 4.3: Block Utilization

2) Average Response Time

Average response time metric indicates the average time required by the FTL (Flash Translation Layer) to fulfil the input request. Here the average response time of various algorithms like BAST, FAST, MNFTL, GroupFTL and Optimal.

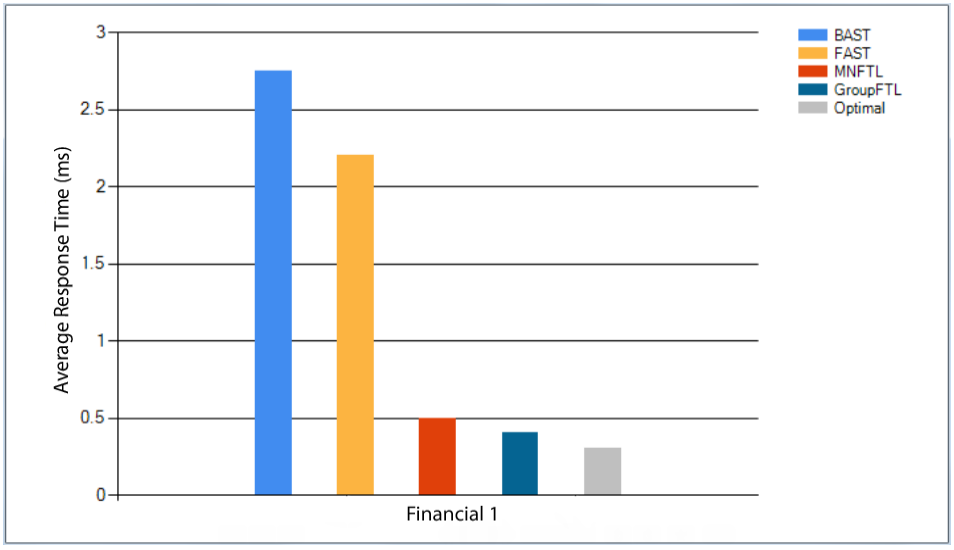


Figure 4.4: Average Response Time

3) Erase Operations Performed

Erase operations are performed on block Level, as the granularity for Erase operation is Block. Hence the erase operations play an important role in deciding the effectiveness and efficiency of the Flash Translation Layer. If the erase operations are performed in large numbers than it can be said that FTL is not efficient and also the increase in erase operations leads to increase in overall system response time degrading the effectiveness of the FTL.

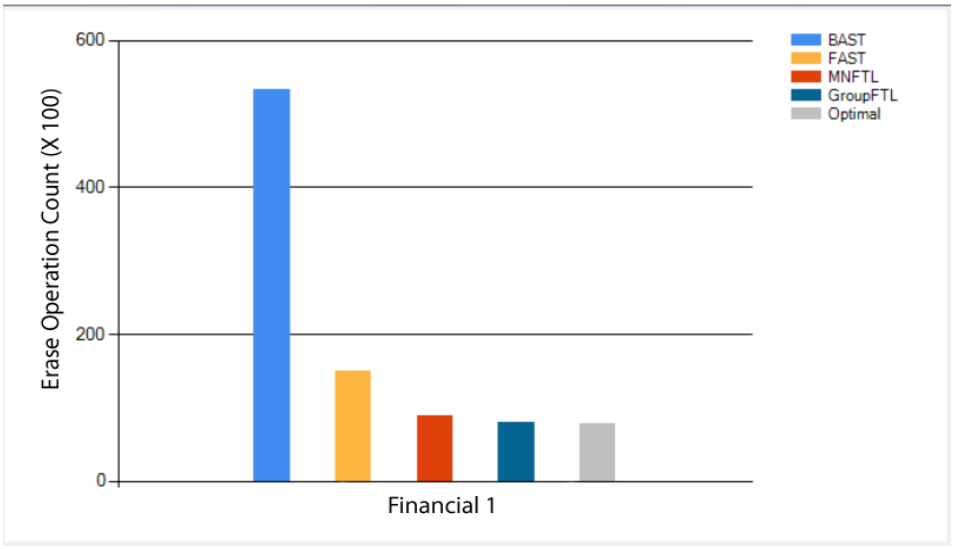


Figure 4.5: Number of erase operations

4.3 EXPERIMENTAL SETUP

To evaluate the performance of algorithm we have developed trace driven GUI simulator which simulates SSD with the parameter describes in TABLE 4.1 and generates the results in form of Graphs. In our experimental tests, each SSD block consists of 128 pages of size 2k by default and the buffer size ranges from 8M to 256M bytes.

Table 4.2: Configuration parameters of SSD ONFI Chip

Capacity	4 GB
Page Size	2046 B
Spare area size	224 B
No of Pages/block	128
Read Time	25micro sec
Write Time	200micro
Erase Time	700

Initially to check the correctness of FTL we have also use synthesize traces .We apply different traces to simulator and verify the content of memory after completion of operation .We have also verify the mapping information using synthesize traces. We have also use Diskmon and blktrace utilities to collect read/write request for hard drive of personal computers. Once we get log file of hard disk access pattern we have slightly modified and giving that file as an input to the simulator to record required parameters through simulator.

We have tested FTL algorithm on a real proto type hardware board having **Altera Cyclone IV E EP4CE75F29C7N** Device with **80MHz NIOS II** processor, **1 Gbit DDR2 SDRAM , 64 Mbit CFI Flash, 64 Mbit Mobile SDRAM (4 M x 16) and Five NAND Flash** Device which is designed by System Level Solutions(SLSCorp). ONFi Flash chip having their own **ONFi 2.3 controller IP** which controls the SLC ONFI flash on the board plus some more peripheral blocks design on board.

Figure 4.6: SSD Hardware with USB 3.0(Courtesy SLS corp.)

We have complete System on Programmable Chip (SoPC) design which is fitted in the board and act as a Solid state drive. The board has 8GB ONFi Flash memory whose page size is 2048 bytes and 128 pages inside the each block. We have implemented a Mass Storage protocol on the board which acts as external removable storage media. System Level Solution has provided their USB3.0 device IP code. This transfers the USB 3.0 device traffic on ONFi memory. On the host side we have used 2.0 GHz Core 2 Duo CPU with 2.5GB DDR with windows XP professional. When we connect board with PC, it will detected as USB drive and OS will send a logical page address to the disk and the FTL and buffer technique method designed inside the board will perform the functionality correctly.

Chapter 5

TESTING AND CONCLUSION

5.1 FTL Simulator

Here we have designed a GUI based FTL Simulator for execution and analysis of NFTL algorithm so that it can be easy to interface NFTL algorithm rather than by using command prompt. There are many parameters that are to be set for execution and testing of NFTL algorithm. It is more time consuming task to set them manually. For that reason we have chosen a GUI based simulator to make the process of setting the parameters of NFTL easy. If we use Command prompt then we have to open each and every files of our algorithm to change parameter according to requirement of analysis at every time. But if we use GUI (Graphical User Interface) base simulator then it is easy as well as it takes less time to change parameter of each and every files of our algorithm. For this, we have used **Microsoft Foundation Classes (MFC)** to design FTL Simulator. As the MFC classes are based on C++, we have used Microsoft Visual Studio for the implementation of Simulator.

Here in MFC classes have many classes like main class is 'CObject' and from this class many other classes are derived like 'CCmdTarget' , 'Exception' , 'File Services' , 'Graphics Drawing' , 'Arrays' etc. Similarly from 'CCmdTarget' class one another class derived is 'CWnd' class which is use for development of Window services. Here we want to develop window application for design GUI base simulator so we have to use 'CWnd' classes in which many other classes are available like 'Frame Windows', 'Dialog Boxes', 'Views' , and 'Controls' by using all these classes we can develop GUI base Simulator. But this simulator support only to Microsoft Windows type OS.

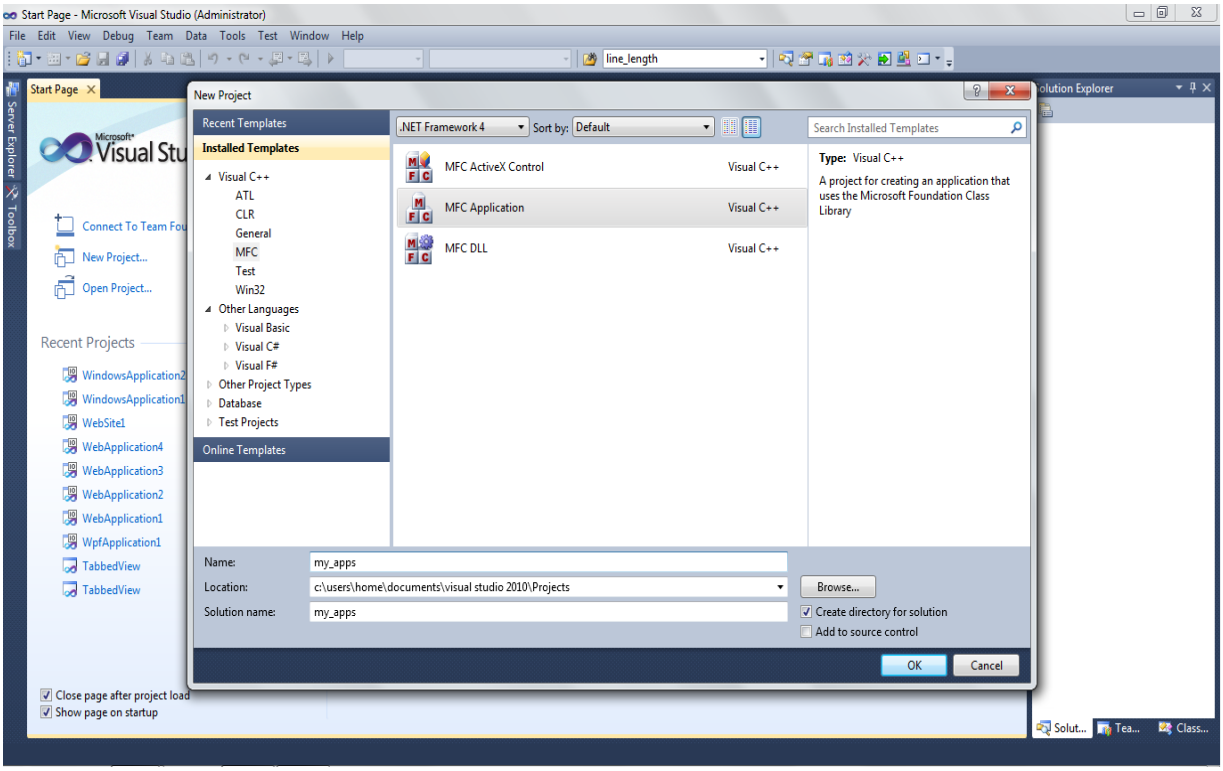


Figure 5.1: How to create new MFC application

Suppose we want to execute same algorithm with different-different parameters as well as in different-different phases like with cache or without cache and after executing with all those parameter we want to analyse their result through graph. If we use Command prompt then we have to open each and every files of our algorithm to change parameter according to requirement of analysis every time. Also if some things are done in one programming language and other things are done in other programming languages, than it make tedious task. But if we use simulator for all these things than it not required to remember all commands and flow of their execution. We only need to select option, write the name of file and push button. Thus we have chosen a GUI based simulator to make the task of analysis easy and user friendly.

In our Simulator we have kept following five tabs:

1) FTL Configuration
2) Cache Configuration
3) Analysis

4) Graph
5) Input Analysis Graph

FTL Configuration: - In FTL configuration we have kept Dialog-boxes, Text-Field and Button. By using Dialog-boxes and Text-Field we can set parameters of FTL algorithm such as flash capacity, spare area, number of block erased, etc. and set these parameters by pushing 'Save' button. After pushing 'Save' button small confirmation dialog-box show us that our data has been saved.

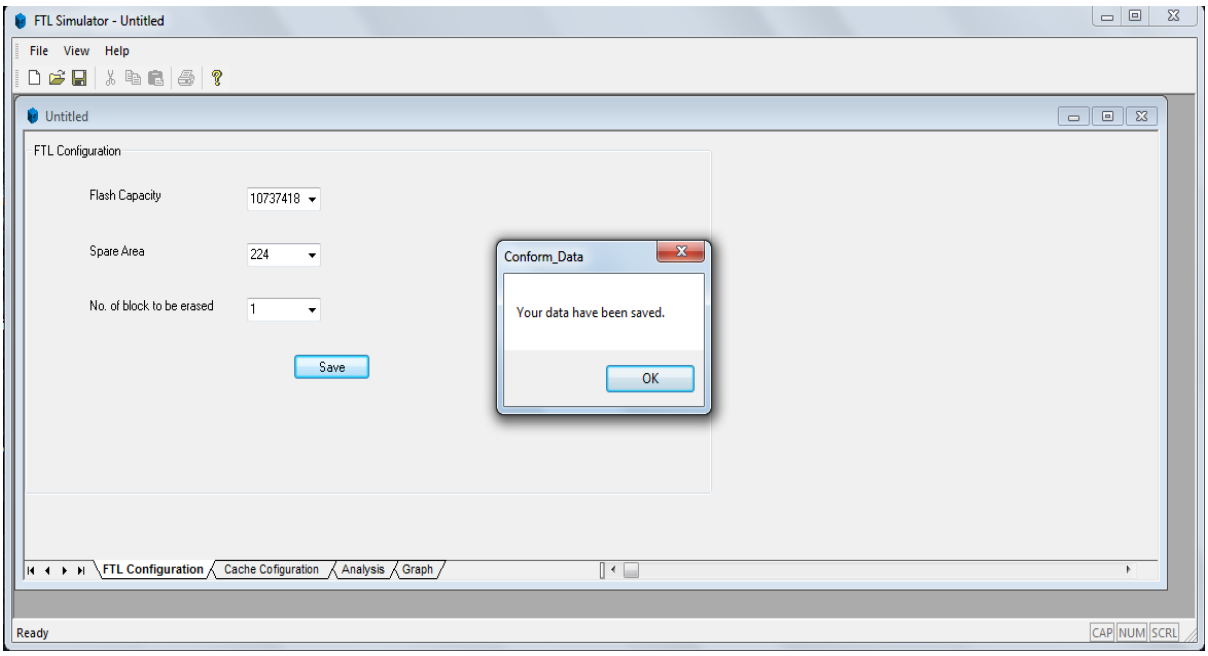


Figure 5.2: FTL configuration.

Cache Configuration: - Here we have kept Dialog-boxes and Buttons. By using Dialog-boxes and Text-Field we choose parameters of Cache algorithm such as size of cache, etc. and set these parameters by pushing 'Save' button. It contains 'Cache Configuration' group box and 'Execution phase' group box. In which we have to choose that either we want to execute with format memory or without format memory and write one input trace file name in text filed and last choose option that if either we want to execute with cache or without cache, At last 'Run' button is to be pushed to execute FTL algorithm along with Cache algorithm on the flash memory.

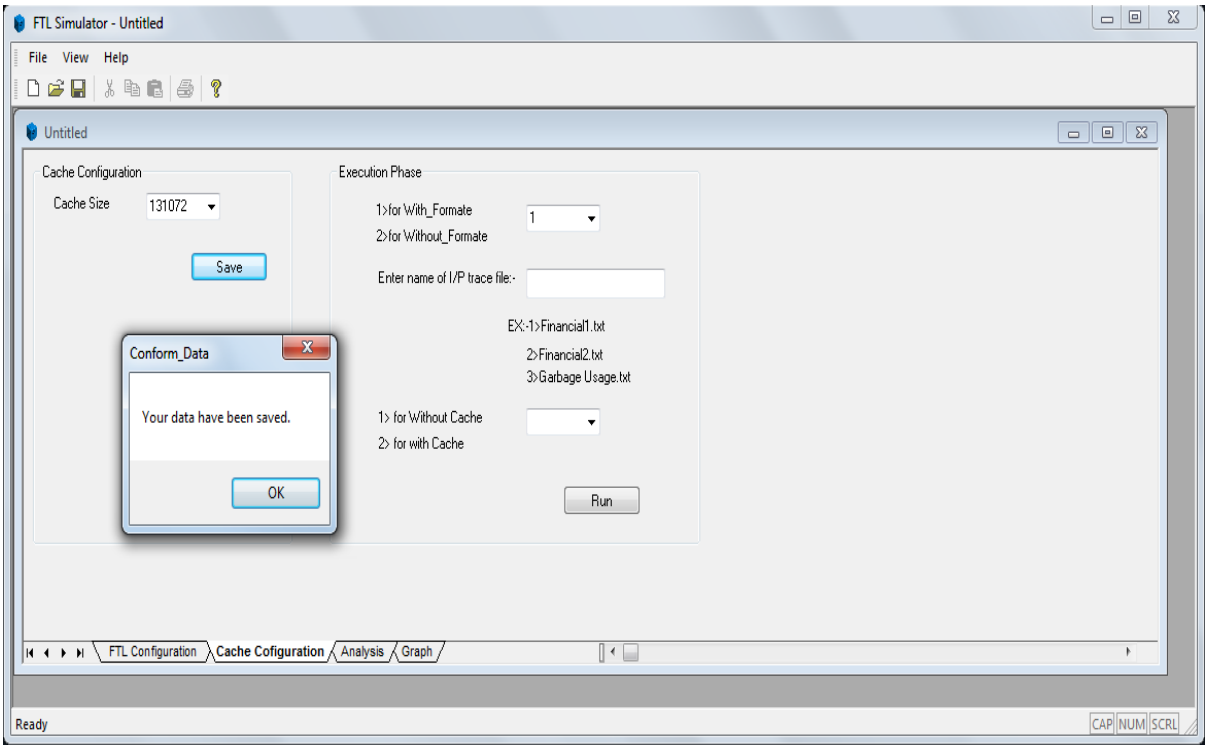


Figure 5.3 (a)

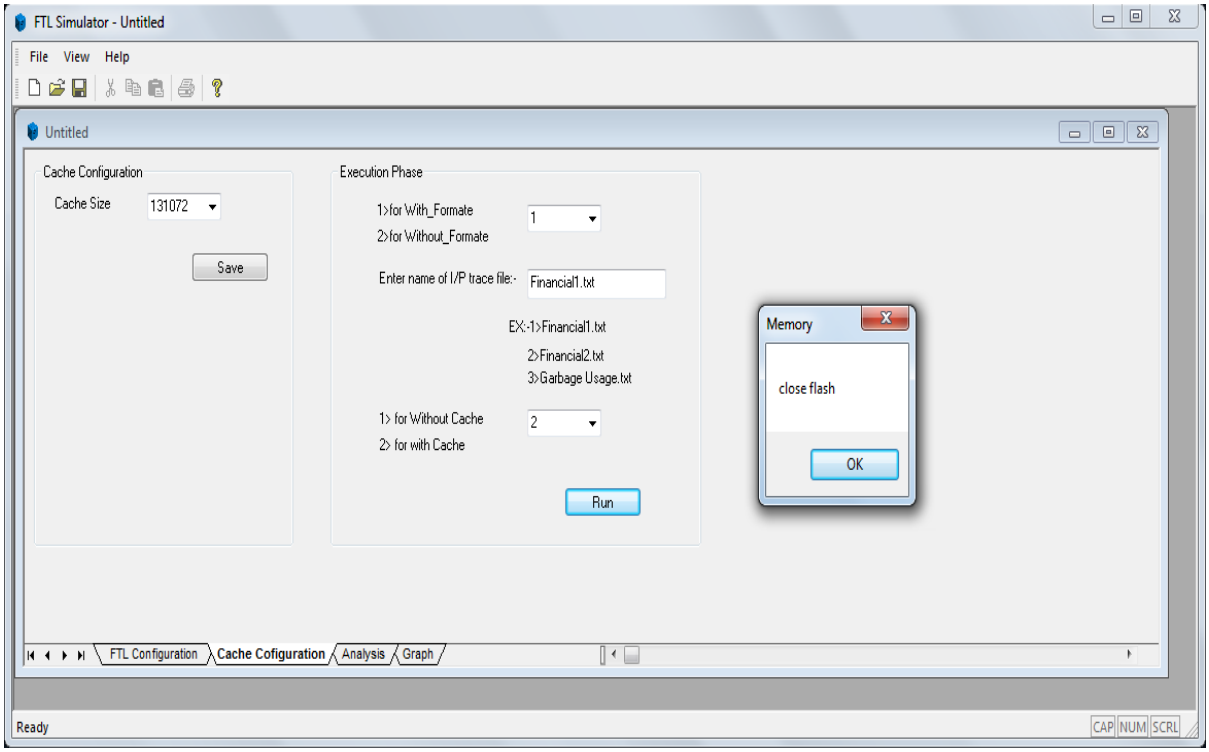


Figure 5.3 (b)

Figure 5.3: Cache configuration.

Analysis: - This module of simulator is used to analyse the results of the caching and FTL algorithm. It gives a graphical analysis of various performance metrics such as number of hits on cache buffer, number of erase operations, number of write operations passed on to FTL, etc. This is useful on further improvement in the algorithm or to devise a more efficient algorithm.

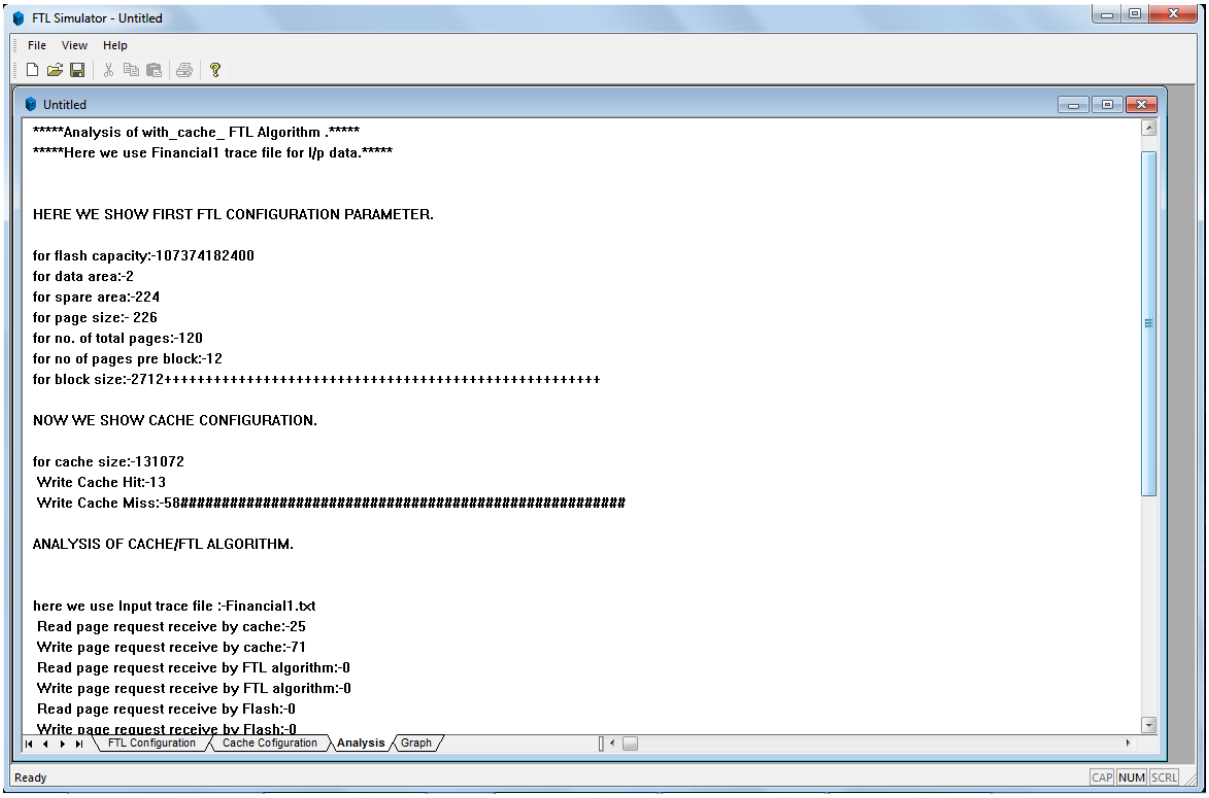


Figure 5.4: Result analysis as shown by GUI simulator

Graph: - This module of simulator is used to show analysis of FTL algorithms in a graphical manner. After executing algorithm based on their result we can compare with it other result of same input data for other FTL algorithm or various parameters by using graphical view. This is better way to compare results of various FTL algorithms.

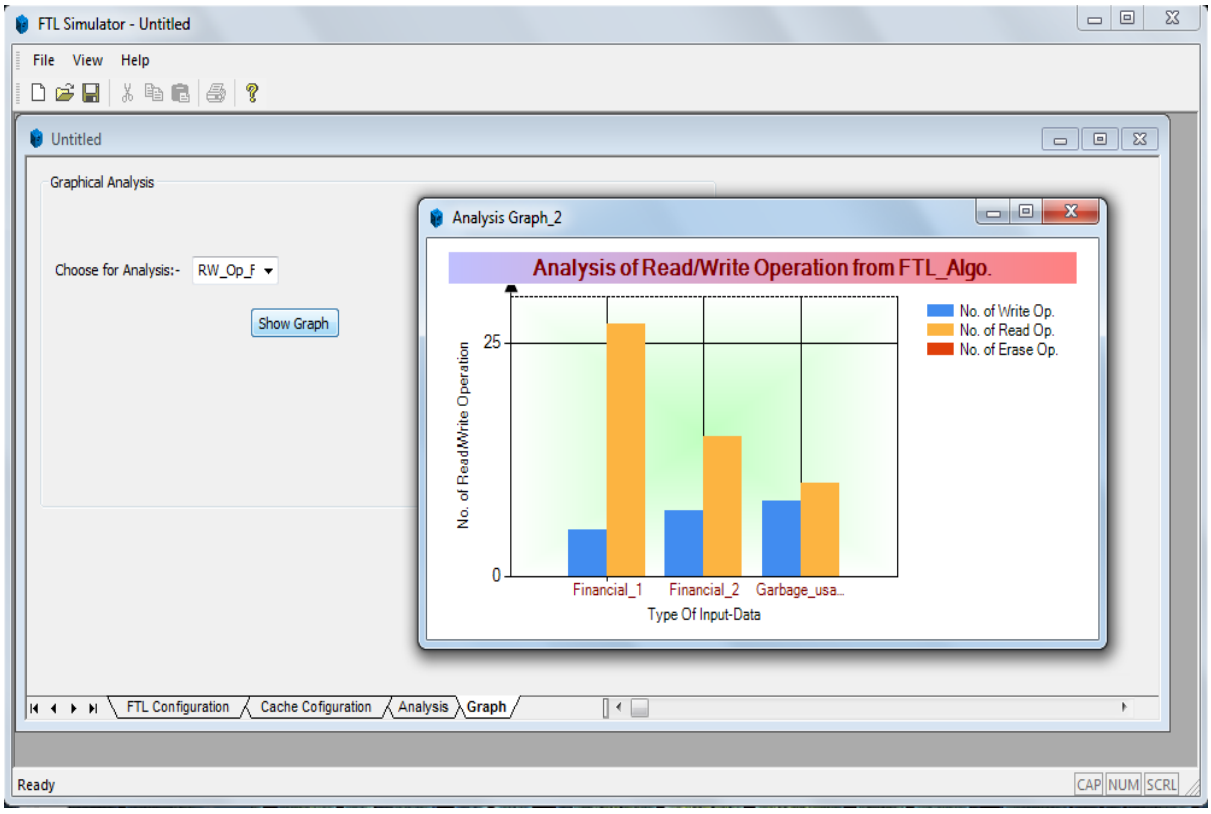


Figure: - 5.5 Graph Generation

Input Analysis Graph: - This module of software is used to analyse the input trace files which are used for the analysis of FTL algorithms. Various type of input pattern like web application where 90% - 99% request come from user side for only read operation. The input trace files may contain extensive read operations or extensive write operations and may contain a balance of both. Input Analysis module helps in understanding these trace files with graphs. In Figure: 5.6 we have designed one application in which one text filed, in which we write input trace file name and push 'Show graph' button to generate the graph for the provided trace file name. In Figure: 5.7 it shows us the graph generated by it. In the graph, Green colour dots indicates 'Write' request and Red colour dots indicates 'Read' request.

Figure: - 5.6(a)

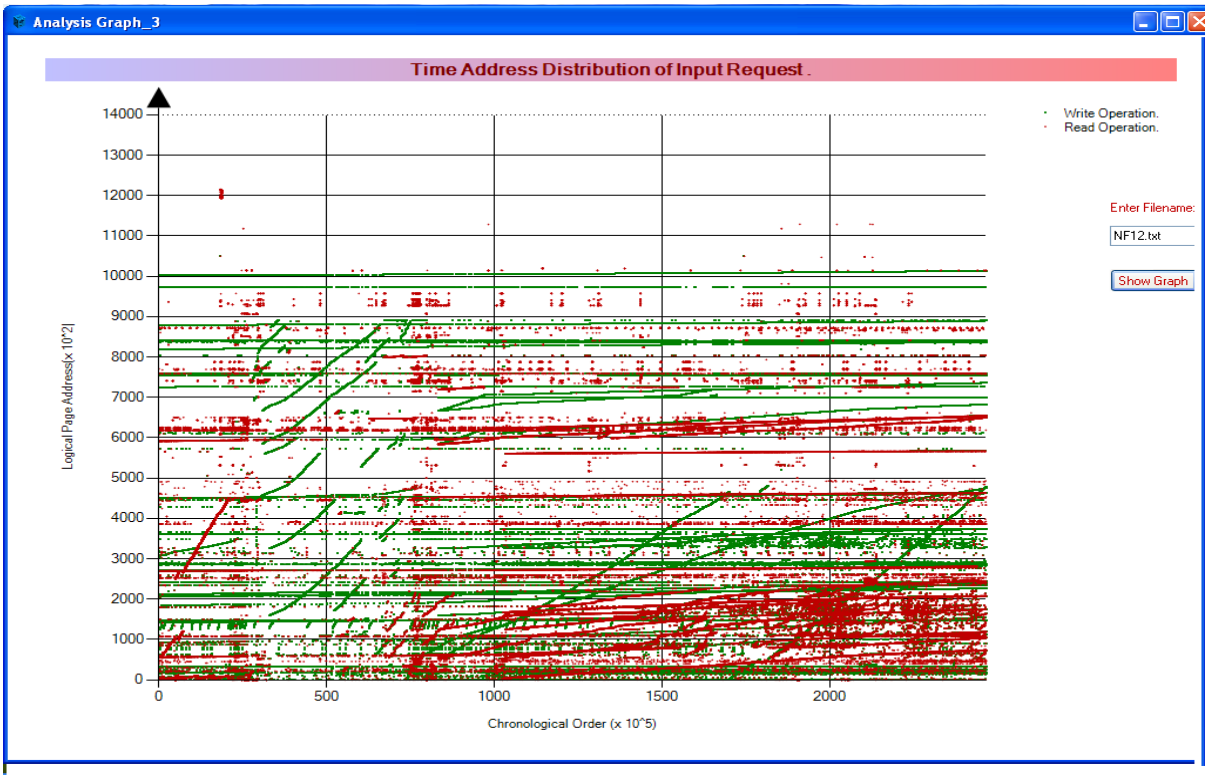


Figure: - 5.6(b)

Figure: - 5.6 Time Address Distribution

Thus our GUI based simulator helps us to set the input parameters for FTL algorithm, caching algorithm easily and perform the analysis and comparison of performance metrics of various algorithms.

Pros of GUI Simulator:-

- We can check and get results for different-different values for different parameters easily.
- We can change mode easily from with cache to without cache or vice-versa at runtime.
- We can specify the name of Trace files at runtime.
- We can generate graph during Runtime just selecting options.
- We can generate a text file containing complete analysis of algorithm.
- We can execute command prompt commands through this simulator.
- We can combine such files which have been designed in different-different Language like here we have combine C# and MFC in same application to generate graphs.
- Not require to remember whole flow of execution process because we have to set whole flow of execution during design of simulator.

5.2: Conclusion and future Offshoots

From the results it can be inferred that GroupFTL reduces space demand of mapping table in cache and exploits only readily available spatial locality. GroupFTL does not impose any strict mapping rule and hence does not impose additional garbage collection cost. With help of GroupFTL we can write entire block, if data is available to write, and hence a cache can be used to flush data equal to the data which can be written in one block.

To exploit this facility efficiently an efficient caching with good page replacement policy is required. In Project-2 we have implemented an LRU cache over GroupFTL which flushes data equal to the data that can be written in one Block.

GUI simulator developed can be used to perform the analysis of the FTL and Caching algorithms. With help of GUI simulator be can generate graphs to show the analysis of the algorithms. One module of the GUI simulator can be used to generate time address distribution of the requests in the Trace files used to perform the analysis.

In future an efficient Caching algorithm can be implemented which provides better response time and efficient page replacement policy.

REFERENCES:

1. Zhiwei Qin, Yi Wang, Duo Liu†, Zili Shao, and Yong Guan, "MNFTL: An Efficient Flash Translation Layer for MLC NAND Flash Memory Storage Systems", June 2011.
2. S.W. LEE, D.J Park, T.S Chung, D.H Lee, H. Song, " FAST " A Log Buffer-Based Flash Translation Layer using Fully-Associative Sector Translation", July 2007.
3. S. Lee, D. Shin, Y. J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," ACMSIGOPS Operating Systems Review, vol. 42, no. 6,Oct. 2008, ,pp. 36-42.
4. Hyunjin Cho, Dongkun Shin, Young IkEom," KAST: K-Associative Sector Translation for NAND Flash Memory in Real-Time Systems", DATE09 © 2009 EDAA.
5. A. Gupta, Y. Kim, B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," Pro. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems, pp. 229-240, March 2009.
6. Jesung Kim et al. "A Space-Efficient Flash Translation Layer for Compact Flash Systems", IEEE Transactions on Consumer Electronics, Vol. 48, No. 2, May 2002
7. Ban, "Flash file system," US Patent 5,404,485, Apr. 1995.
8. J.-U. Kang, H. Jo, J.-S. Kim and J. Lee. ,"A superblock-based flash translation layer for nand flash memory. In Proc. of International Conference on Embedded Software", (EMSOFT), 2006, pages 161 – 170.
9. S. Y. 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 Proc. of International Workshop on Rapid System Prototyping, 2007, pages 202–208.

10. U.T.Repository, "OLTP Application I/O." [Online]. Available at: http://traces.cs.umass.edu/index.php/Storage/Storage

11. Uthash.h header file from http://uthash.sourceforge.net by Troy D Hudson.

12. SLC vs. MLC: An Analysis of Flash Memory by Super Talent Technology http://www.supertalent.coms