University of Massachusetts - Amherst
ScholarWorks@UMass Amherst

Masters Theses 1896 - February 2014 | Dissertations and Theses

2011

Enhanced Search And Efficient Storage Using Data Compression In Nand Flash Memories

Shruti S. Vyas
UMass, vyas.shruti@gmail.com

Follow this and additional works at: http://scholarworks.umass.edu/theses

Vyas, Shruti S., "Enhanced Search And Efficient Storage Using Data Compression In Nand Flash Memories" (). *Masters Theses 1896 - February 2014*. Paper 548.
http://scholarworks.umass.edu/theses/548

This Open Access is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Masters Theses 1896 - February 2014 by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

ENHANCED SEARCH AND EFFICIENT STORAGE USING DATA COMPRESSION IN NAND FLASH MEMORIES

A Thesis Presented

by

SHRUTI VYAS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2011

ELECTRICAL AND COMPUTER ENGINEERING

© Copyright by Shruti Vyas 2011

All Rights Reserved

ENHANCED SEARCH AND EFFICIENT STORAGE USING DATA COMPRESSION IN NAND FLASH MEMORIES

A Thesis Presented

by

SHRUTI VYAS

Approved as to style and content by:

____

Sandip Kundu, Chair

____

Russell Tessier, Member

____

Maciej Ciesielski, Member

____

C. V. Hollot, Department Head
Electrical & Computer Engineering

ACKNOWLEDGEMENTS

I would like to thank my thesis Adviser Professor Sandip Kundu for trusting me with this project and giving me an opportunity to work on this topic. I really appreciate the constant guidance and different ideas given by him in the process of my research and accepting my ideas too. I would like to thank Professor Russell Tessier and Professor Maciej Ciesielski for accepting to be a part of my thesis committee.

I would like to thank my labmates Aswin and Lokesh for participating in the several brainstorming sessions that we had in KEB 306.

I would also like to thank Abhinav, my husband, for supporting and trusting me throughout my research. Lastly I would thank my family and friends for having faith in me and standing by me always.

ABSTRACT

ENHANCED SEARCH AND EFFICIENT STORAGE USING DATA COMPRESSION IN NAND FLASH MEMORIES

FEBRUARY 2011

SHRUTI VYAS, B.E., EE, BITS PILANI, RAJASTHAN, INDIA

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS, AMHERST

Directed by: Professor Sandip Kundu

NAND flash memories are popular due to their density and lower cost. However, due to serial access, NAND flash memories have low read and write speeds. As the flash sizes increase to 64GB and beyond, searches through flash memories become painfully slow. In this work we present a hardware design enhancement technique to speed-up search through flash memories. The basic idea is to generate a small signature for every memory block and store them in a signature block(s). When a search is initiated, signature block is searched which produces reference of possible blocks where data might be contained, reducing the total number of read operations. The additional hardware has no impact on read access times or sequential write times but increases the random write times by an average of 8-9%. Simulation experiments were performed for flash memory of size up to 16Gb. Simulation results show that the performance of searches improve by 2000X by using the proposed technique. The signature-based technique is used to find exact matching data. A discrete cosine transform based technique is used when partial matching of data is required. The same setup is also used to increase storage efficiency of data by performing data deduplication on the flash memory. The hardware implementation of the search technique results in 0.02% increase in area, 3.53% increase in power and can operate at a maximum frequency of 0.47GHz.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT v

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1. INTRODUCTION 1

2. BACKGROUND AND RELATED WORK 6

2.1 NAND Flash Design 6
2.1.1 NAND Flash Transistor 6
2.1.2 NAND Flash Architecture 7
2.1.3 NAND Flash Commands and Addressing 8
2.1.4 NAND Flash Command Operation 9
2.2 Multiple Input Shift Register (MISR) 11
2.3 Discrete Cosine Transform (DCT) 12
2.4 Related Work 13

3. MISR-BASED ARCHITECTURE 15

3.1 Introduction 15
3.2 Scheme 15
3.3 Hardware Overhead 18
3.4 Experimental Framework 19
3.4.1 Platform 19
3.4.2 Experiments 21
3.5 Results and Analysis 22

4. DCT-BASED ARCHITECTURE 25

4.1 Introduction 25
4.2 Scheme 25
4.3 Hardware Overhead 27
4.4 Experimental Framework 28
4.5 Results and Analysis 28
4.5.1 DCT on an Image File 28
4.5.2 Timing Analysis 30
4.5.3 Spare Block Estimation 30

5. EFFECT ON PRODUCT LIFETIME 33

6. EFFICIENT STORAGE USING COMPRESSION 34

6.1 Introduction 34
6.2 Data Deduplication 34
6.3 Data Deduplication for Flash Memory 36
6.3.1 Implementation 36
6.3.2 Assumptions 38
6.4 New Memory Operations 39
6.4.1 New Write 39
6.4.2 New Read 41
6.4.3 New Erase 42
6.5 Results and Analysis 42
6.5.1 Data Deduplication and Storage Gain 42
6.5.2 Effect of Data Deduplication on Performance 44
6.5.2.1 Write Time 45
6.5.2.2 Read Time 46
6.5.3 Data Deduplication with varying memory size 47

7. HARDWARE IMPLEMENTATION 48

7.1 Verilog Implementation 48
7.1.1 Structure of Design 48
7.1.2 Finite State Machine Description 49
7.2 Results and Analysis 51
7.2.1 Simulation Results 51
7.2.2 Area Calculation using Design Compiler and CACTI5.3 56
7.2.3 Power Estimation using Design Compiler 59
7.2.4 Timing Analysis using Design Compiler 60

8. CONCLUSIONS 62

9. BIBLIOGRAPHY 64

LIST OF TABLES

Table Page

1. NAND Flash Operations 8
2. 2Gb SLC Addressing Scheme 9

LIST OF FIGURES

Figure Page

1. Price Trend of Flash vs HDD [1] ... 2
2. Transistor-level NAND Flash design ... 3
3. Floating Gate Flash Transistor ... 6
4. NAND Flash Design ... 7
5. Timing Diagram of the Program (Write) Command Operation ... 10
6. Timing Diagram of the Modified Write Command Operation ... 11
7. Timing Diagram of the Proposed Search Operation ... 11
8. Three Input MISR with $P(x) = x^3 + x + 1$ ... 12
9. Eight Input MISR Design with $P(x) = x^7+x^4+x^3+x^2+1$ ... 15
10. MISR-Based Architecture [11] ... 17
11. Pseudo Code of TurboNFS [11] ... 20
12. Search Gain with TurboNFS ... 22
13. Comparison of Sequential Write with and without TurboNFS ... 23
14. Penalty on Random Write due to TurboNFS ... 24
15. Effect of Signature Length ... 24
16. DCT-based Architecture ... 26
17. Image written into the Flash memory ... 29
18. (a) Search Request which is part of image (b) A distorted search request ... 30
19. Increase in Spare blocks with number of DCT coefficients ... 31
20. Effect on Spare blocks with the size of data on which DCT is performed ... 32
21. Example of Data Deduplication [13] ... 35

22. 2Gb Flash memory Structure with Data Deduplication Scheme 37
23. Header Page Format 38
24. Pseudo Code for New Write and Read 40
25. Flowchart for New Scheme with Data Deduplication 41
26. Increase in storage gain with duplication 43
27. Number of pages required to write File A and File B 44
28. Effect of Data Deduplication on Write Time 46
29. Effect of Data Duplication on Read Time 47
30. Finite State Machine for Flash Operations 49
31. Module Level Diagram of the Flash Hardware Design 52
32. Modelsim Waveform for READ Operation 54
33. Modelsim Waveform for WRITE Operation 55
34. Modelsim Waveform for SEARCH Operation 55
35. Using CACTI 5.3 for Memory Area Calculation 56
36. Area, Power and Timing Numbers for 2Gb Memory 57
37. Area, Power and Timing Numbers for Cache 57
38. Area Calculation for Logic using Design Compiler [18] 58
39. Power Analysis for Logic using Design Compiler 60
40. Timing Report showing Clock and Slack Numbers 61

CHAPTER 1

INTRODUCTION

Memories form an inseparable part of computers, laptops, mobile phones, digital cameras and most electronic consumer appliances today. Memories are classified majorly as volatile and non-volatile. SRAM and DRAM are classified as volatile memories since the data is retained only till the power is supplied. Non-volatile memories however do not require maintained power supply for retaining data. Flash memories are classified as non-volatile memories that are reprogrammable and electrically erasable. Flash memories are further classified as NOR flash and NAND flash due to their structure. NAND flash memories are generally used for data storage and NOR flash for code storage due to the fact that NOR flash is good for random reads and NAND flash for serial access of data.

NAND flash has enjoyed a phenomenal growth rate in storage capacities as well as a steady decline in pricing during the past few years. These developments have enabled NAND memories to enter and possibly change or displace some traditional storage architectures. NAND flash memories have become ubiquitous in consumer applications. They are known for their dense structure and relatively inexpensive data storage. Flash memories are gaining popularity over traditional HDDs (Hard Disk Drive) due to features like low power consumption, small size and reliability due to lack of moving parts. Figure 1 shows the price trend of flash drive versus HDDs which clearly shows that flash drives are catching up with the hard drives very fast and would gain more market share in the next few years [1]. However, flash memories have lower write, erase and search speeds compared to HDDs. Typical page read time of 25µs, page write time of 300µs and erase time of 2ms have been reported [2]. Searching a flash drive may require reading the entire memory sequentially. Hence, if the memory size is 16Gb, searching it would

take ~26sec which is decidedly slow. The main goal of this research is to speed up data searches through NAND flash to a more acceptable level, to an order of milliseconds.

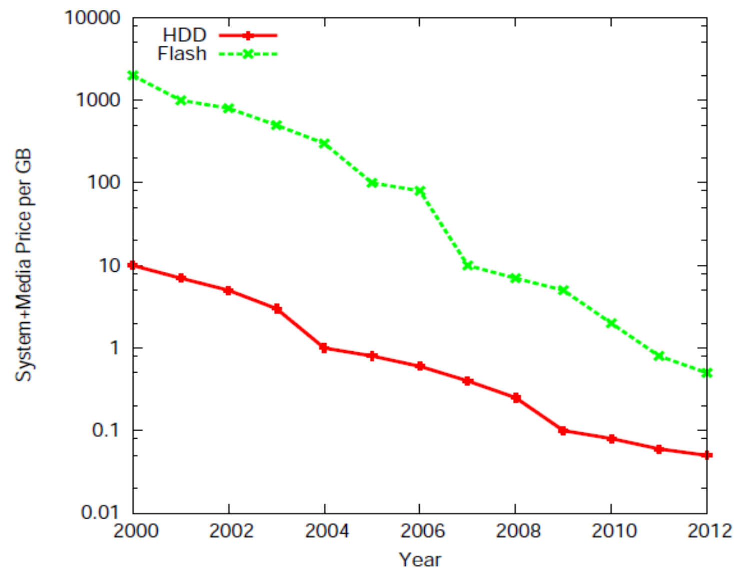

Figure 1: Price Trend of Flash vs HDD [1]

As the memory size increases so does the search time. Hence it is essential to reduce this search time by incorporating some dedicated mechanisms. NAND Flash memories are characterized by a high random read speed. This has contributed to them being used as a replacement for traditional Hard disk based storage. The cost per GB of Flash memory is currently very high but the downward price trend will make them more affordable and ubiquitous in the near future. Some of the important characteristics of NAND flash are:

1. Non-volatile storage
2. Solid state storage, no moving parts
3. Limited write and erase cycles
4. Ensures data persistence for around 10 years
5. Erases are possible on large blocks of data
6. Less fragile and more reliable than hard disk drives

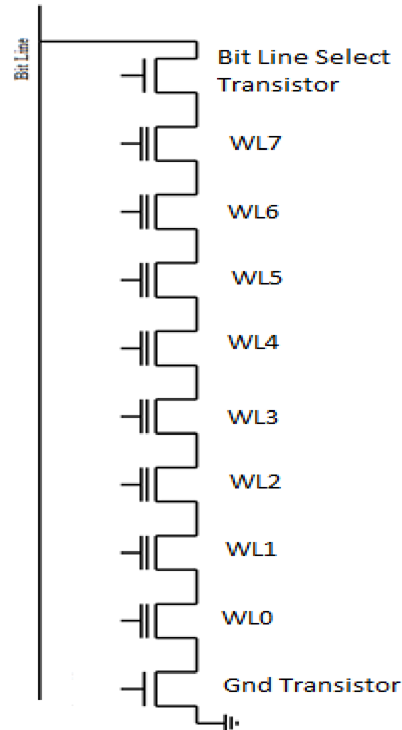

Figure 2: Transistor-level NAND Flash design

A typical transistor-level NAND flash memory is shown in Figure 2. The transistors are connected in series as shown in the figure and hence the structure looks like a NAND gate design. Bit line access is given to only one among several transistors. In a NAND flash device there is no byte by byte access but only pages of data can be accessed. Due to serial access, NAND memories have a very high read speed. However, they have lower write, erase and search speeds compared to HDDs. Also, as the size of NAND flash increases the speed of searching desired data in a flash memory reduces. This is because in order to perform a search, the data needs to be transferred from the flash disk to the CPU, where the search key and the data are matched. This method can be improved upon by keeping the CPU out of the loop and performing the search on the storage media itself. This also has an additional advantage of being able to scale search capability with increased number of flash disks, i.e. there can be as many parallel searches as there are flash disks. This research aims at developing a hardware-based search

technique for fast and efficient flash memory searches. We call this technique TurboNFS (Turbo NAND Flash Search) since it speeds up the flash memory search. This extra piece of hardware will be located on the flash memory itself and will perform search locally instead of assigning the searching task to CPU. This concept is already implemented in RAID Memories [3]. In this work we are trying to extend this idea to NAND Flash memories.

As a proof of concept, we have implemented the search mechanism using two approaches. The first approach focuses on a MISR (Multiple Input Shift Register) based data compaction method. This method is very simple and needs minimal hardware for its implementation. It gives a very good compaction ratio (1:2048). However it gives only exact matches of data. The second approach is based on compaction using discrete cosine transform. This method consumes more hardware but gives partial data matches and is very commonly used in image compression applications.

The MISR-based method finds an exact match of data and is also capable of returning multiple matches. In this method a fingerprint, hash or a signature of every written page of data is stored into a special part of memory called spare blocks. These spare blocks are typically used for bad block management, wear leveling etc. When a search request comes in, instead of reading the whole memory sequentially, we read just the spare blocks for the signatures.

The Discrete Cosine Transform based technique gives partial matches and can be used for searching multimedia or image files. In this method a discrete cosine transform of the data to be written is calculated and the top coefficients in the frequency domain along with their positions are stored in the spare blocks similar to the MISR-based signatures as mentioned in the first method. For an incoming search request, these top coefficients and their positions are compared

and a match if found is returned. Since just the top coefficients are compared this method helps in giving partial matches which are useful in case of multimedia and image files.

The second phase of this research concentrates on using these two compression techniques for efficient data storage in the NAND flash memories. This data compression technique is called data deduplication. It uses the MISR based compression technique to do data compression and use data deduplication methods to save space on a NAND Flash drive.

A flash memory simulator was developed in C++ to evaluate the performance of read, write, erase and search operations of the proposed flash memory. The performance for various memory parameters is evaluated using this simulator. The search technique developed in this study increases the search speed of a 2Gb memory by 2000X and a 16Gb memory by 4000X. Gains are greater for larger memories. The most common command used on a flash drive is a read and this method leaves the read performance unaffected. The signature generation and concurrent signature writes, have no impact on sequential writes. The random writes have 8-9% penalty due to signature generation which goes down when such writes are pipelined. Usually, multiple memory pages are written at once. This helps in hiding the increase in write latency.

The next chapter discusses the background behind this work. Chapters 3 and 4 discuss the MISR-based and DCT-based techniques in detail with all the results and analysis. Chapter 5 talks about the effect of this method on product life time. Chapter 6 discusses the data deduplication based compression and storage technique in detail. The area overhead due to additional hardware is 0.02% and power overhead is 3.53%, calculated using Design Compiler. Hence the search method speeds up the flash memory searches with minimal area and power overhead. The maximum operating frequency of this design is calculated using design compiler and is found to be 0.47GHz.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 NAND Flash Design

This section describes the NAND Flash design, architecture, addressing scheme and operations in detail. In order to understand the TurboNFS method, it is important to know the basics of NAND Flash operation.

2.1.1 NAND Flash Transistor

Figure 3 shows the structure of a NAND Flash transistor which looks different than a regular transistor with a single gate. A NAND Flash transistor has a floating gate which acts as the storing electrode for the cell device. Charge injected in the floating gate is maintained which allows modulation of the threshold voltage. The neutral (or positively charged) state is associated with the logical state “1” and the negatively charged state, corresponding to electrons stored in the FG, is associated with the logical “0”. Charge can be trapped in the floating gate and retained for about 10-15 years if the transistor is not reprogrammed. Thus the floating gate makes the flash transistor non-volatile in nature.

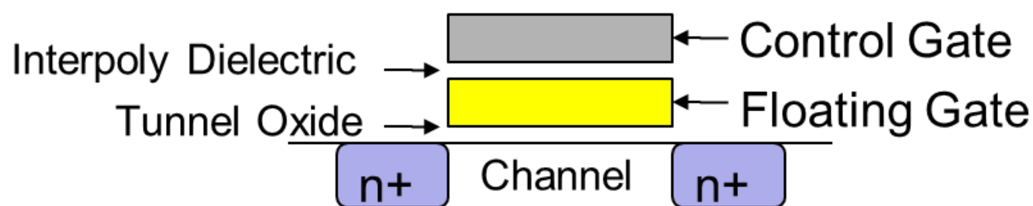

Figure 3: Floating Gate Flash Transistor

In our study we assume the NAND Flash memory to be SLC type. An SLC flash transistor can store only 1 bit of data (0 or 1) as opposed to MLC (Multi Level Cell) structure which can store 2, 4 or 8 bits in a cell. This is done by choosing between multiple levels of electrical charge to apply to the floating gates of its cells. As we move from SLC to MLC flash

memories the density of data storage increases but the write endurance reduces and the error probability increases. Thus it is not recommendable to go beyond 2 or 4 bit MLC designs [4].

2.1.2 NAND Flash Architecture

The NAND Flash design used in our architecture is shown in Figure 4. A 2Gb device consists of 2048 blocks of memory consisting of 64 pages each. Each page is made up of 2112 bytes, 2048 bytes of data and 64 bytes of spare memory for ECC, wear-leveling etc. A register, the size of a page is used to shift data in and out while programming or reading a page. 8 bits of data are shifted per cycle. Although programming and reading the flash memory can be done page by page, only block erasures can be performed. A block erase is quite expensive and takes about 2ms, hence should be used sparingly. The product life of a NAND flash memory depends on the number of erases done per block. Typically, a SLC memory supports 100,000 erases per block. We use a flash memory of 2KB page and 64 pages per block in our design. For memories greater than 2Gb, the number of blocks is increased keeping the page and block size constant.

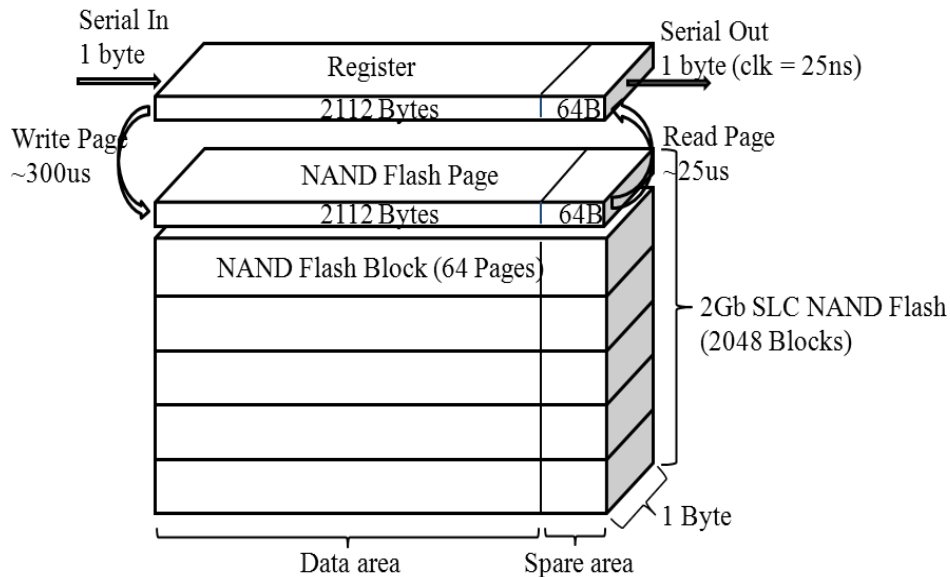

Figure 4: NAND Flash Design

2.1.3 NAND Flash Commands and Addressing

The normal commands used to access the flash memory for reading and writing into it is shown in Table 1. Column 2 represents the name of the command. Column 3 represents the address cycles required to perform that operation. For example reading and programming need 5 address cycles whereas block erase needs only 3. This is because for a block erase, only a block needs to be addressed and not individual pages in that block. Reading and erasing do not need any data cycles but programming a page does. Command cycle 2 is to confirm if the command was executed successfully. In our design we have added a new command called search which is addressed by *99h* command name. It needs data cycles and no address cycles since the command supplies data to be searched and returns addresses of the matches found. The signals needed to perform these operations are read enable, write enable, chip enable, command latch enable and address latch enable. The working of a NAND flash memory can be found in [4].

Table 1: NAND Flash Operations

Command	Command Cycle 1	Address Cycles Required	Data Cycles Required	Command Cycle 2
Page Read	00h	5	No	30h
Program Page	80h	5	Yes	10h
Block Erase	60h	3	No	D0h
Random Data Read	05h	2	No	E0h
Random Data Input	85h	2	Yes	-
Reset	FFh	-	No	-
Search	99h	-	Yes	98h

Table 2: 2Gb SLC Addressing Scheme

Cycle	I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0
I	C7	C6	C5	C4	C3	C2	C1	C0
II	Low	Low	Low	Low	C11	C10	C9	C8
III	B7	B6	P5	P4	P3	P2	P1	P0
IV	B15	B14	B13	B12	B11	B10	B9	B8
V	Low	Low	Low	Low	Low	Low	Low	B16

The addressing scheme of a NAND flash of size 2Gb is shown in Table 2. The I/O bus is 8 bit wide and hence it needs 5 cycles to shift in an address which is 40 bits wide. Bits C0:C11 represent column address. Since every page is 2048 bytes in size, 12 bits are required to address a column of a page. 6 bits, P0:P5 are used to address a page since there are 64 pages in a block. A 2Gb memory has 2048 blocks and hence needs 12 bits (B6:B16) to address a block. It is clear that a block erase will need address cycles III, IV and V only, to access a block. On the other hand a page read will need all five address cycles. Since the memory size is 2Gb the Address cycle V has 7 bits which are low. This addressing scheme can represent a maximum memory of size 256 Gb (32GB).

2.1.4 NAND Flash Command Operation

Figure 5 shows how a program or a write command is executed. At the falling edge of WE# (write enable) and rising edge of CLE (Command Latch Enable), the program command (80h) is sent out over the I/O bus. After this the ALE (Address Latch Enable) goes high and at the falling edge of WE#, the five bytes of address are shifted in. Once the address is shifted in, the ALE goes low and data starts shifting in at each cycle. 2112 bytes of a page take 2112 cycles to be shifted into the page register. At the end of this, a *program confirm* command (10h) is sent

to make sure that write was executed successfully. t_{prog} is the time taken to program the page after the register is loaded. This time is typically around 300 to 700 us. Notice that the read enable is always high throughout the program operation except when the status of the operation is to be verified.

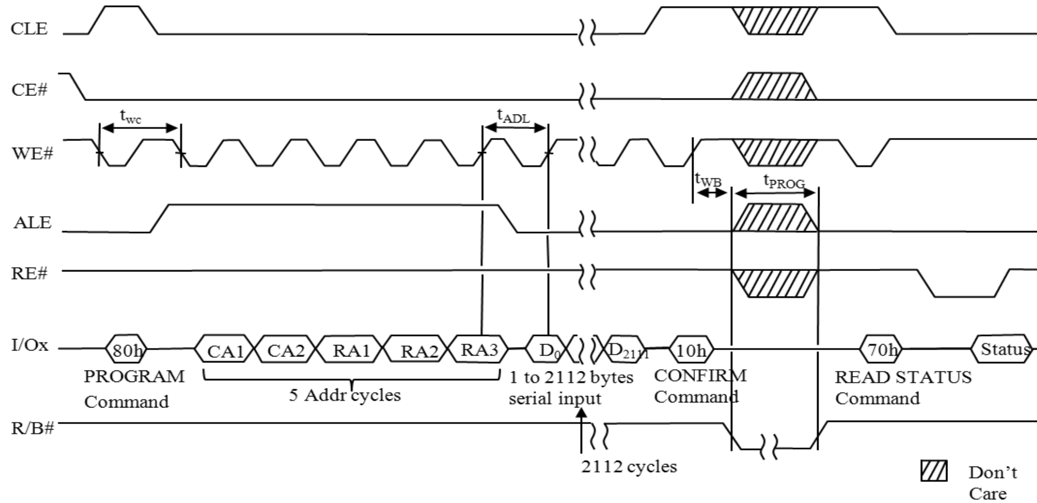

Figure 5: Timing Diagram of the Program (Write) Command Operation

Figure 6 shows the new write command operation. We see that the normal write is executed followed by the random data input command (85h) which writes into the signature block. The figure shows the operation when signature block is written with signatures. If the signature is written into the buffer, Figure 6 would look simpler. The random data input command is followed by the address of the signature block where the signature should be written. After this, the signatures are sent on I/O. At the end of this a confirm command is given followed by the status check.

The search operation of our search scheme is represented by a signal transition diagram shown in Figure 7. The search operation starts with the command 99h given on the I/O bus at the falling edge of WE# and rising edge of CLE. Since there are no address cycles associated with the search operation, the data starts getting shifted on the I/O when the WE# is low. At the end of

2048 cycles, the search data is completely shifted in. Now the search operation will actually begin by calculating the signature of the search data. Then the signature block needs to be read and hence the signature block address is sent on I/O. The signature block is read using the random read command, 30h. At the end of this operation, the addresses of matching data are returned on the I/O bus. 98h represents the search confirm command.

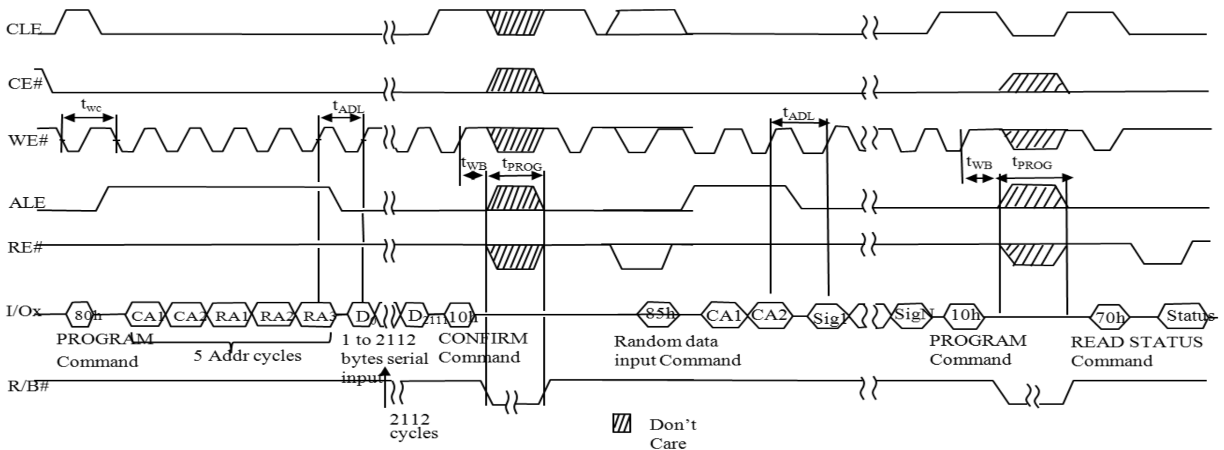

Figure 6: Timing Diagram of the Modified Write Command Operation

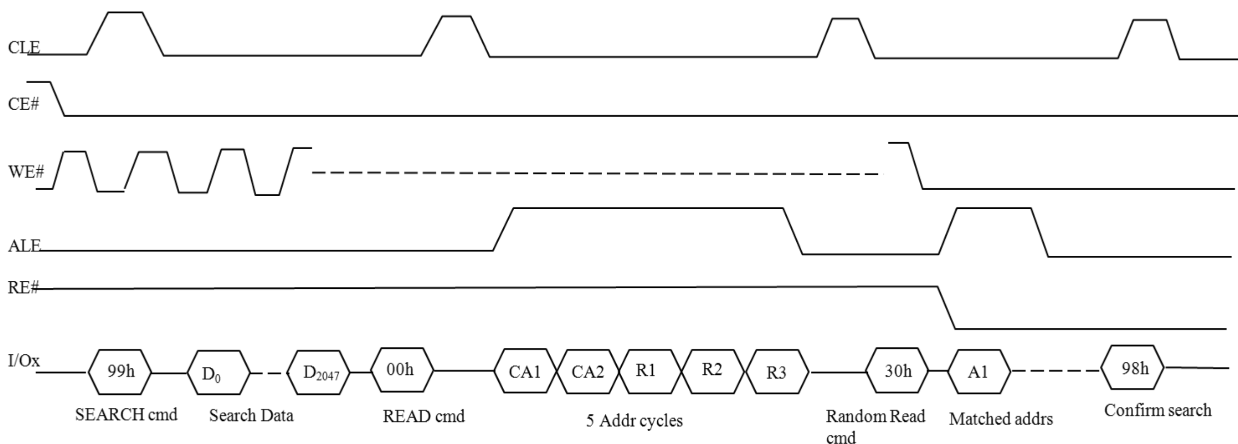

Figure 7: Timing Diagram of the Proposed Search Operation

<u>2.2 Multiple Input Shift Register (MISR)</u>

First part of this work concentrates on generating the signature of data using a Multiple Input Shift Register technique. MISR design is usually used for applications where data is

compressed into a signature. It is rooted in an indivisible polynomial structure. Basic structure of a MISR is shown in Figure 8[5]. The polynomial represented by this design is $x^3 + x + 1$. S0, S1 and S2 represent register bits. The inputs bits *In*[0:2] are XOR-ed with the normal polynomial structure. Such an arrangement of XOR gates and shift registers generates a pseudorandom binary sequence (PRBS) at the output of the shift registers. The polynomial associated with a MISR depends on the register outputs that are tapped to feed them back to XOR gates. Thus if an input *sequence* is applied to this MISR, it compresses the data into a 3 bit *key* or a *signature.* An ideal signature is unique to the input stream. For our purposes, signature need not be unique. MISR signature analysis is used widely in testing VLSI circuits because it is a very simple yet powerful testing design which reduces test time.

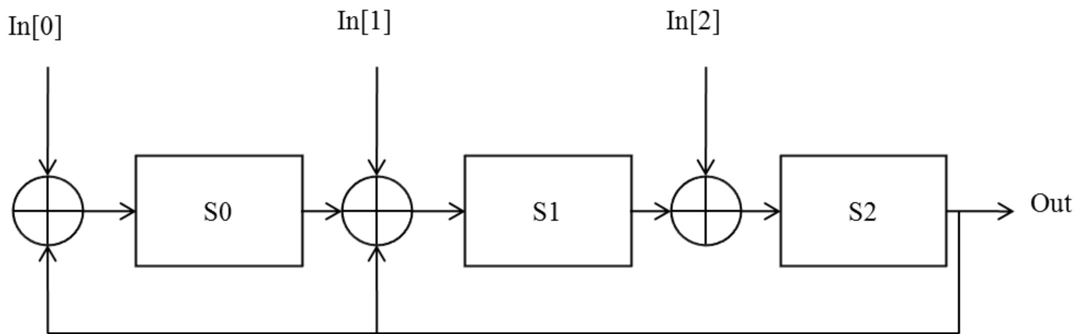

Figure 8: Three Input MISR with P(x) = $x^3 + x + 1$

2.3 Discrete Cosine Transform (DCT)

The second part of the study focuses on implementation of a Discrete Cosine Transform based compression and storage technique. This technique is helpful for partial matching of data and can be used for large multimedia file searches. A **discrete cosine transform** (**DCT**) expresses a sequence of finitely many data points in terms of a sum of cosine functions oscillating at different frequencies. Discrete cosine transforms find their applications in lossy compression of audio and images where small high-frequency components can be discarded. The

cosine functions are used in these applications instead of the sine functions for compression because it turns out that cosine functions are much more efficient (fewer are needed to approximate a typical signal. Discrete cosine transforms are available in different forms according to their type of applications. The most commonly used form of DCT for compression purposes is the DCT-II form which is available in 1 Dimension and 2 Dimensions. We used the 1D form for audio input files and image files. The 2D form is expected to give better results for image and video files. Hence this form will be used in the future work. The 1D DCT-II form looks as follows

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad k = 0, \ldots, N-1.$$

Where x1, x2…xn represent the time domain coefficients and X1, X2…Xk represent the frequency domain coefficients [6]. If N=100, k will range from 0 to 99. Hence we will get 100 frequency domain coefficients. But only few of these will be the dominating ones and hence the other small high frequency components can be discarded. This property of DCT is called energy compaction which makes it highly useful in compression techniques.

2.4 Related Work

Previously, work on design to speed-up searches were concentrated on memory hierarchy or hard disk drives. Bu *et al* proposed a keyword matching architecture capable of performing fast dictionary searches with approximate matching capability [7]. In [8] Singaraju *et al* describe a cache memory based architecture for lookup operations which is used in network processing applications. Papers [9] and [10] describe search techniques using ternary content addressable memories. In short, all these architectures utilize the application specific hardware implementation to achieve fast searches. The main component used in all of these architectures is

a content accessible memory (CAM) which stores and compares data in one clock cycle. The disadvantage of the CAM-based method is the area overhead due to the comparison circuit attached to each CAM cell. This extra circuitry also increases power dissipation since CAMs tend to be power intensive. Our NAND Flash search method however has less area and power overhead compared to traditional CAM-based search methods since it uses a signature generator to compress data. It has a simplified control structure and maximizes use of a buffer circuit to reduce reliance on CAM. Thus the TurboNFS method speeds up data search in a NAND Flash memory with minimal area and power overhead when compared with the techniques mentioned above.

CHAPTER 3

MISR-BASED ARCHITECTURE

3.1 Introduction

We saw in section 2.2 that a Multiple Input Shift Register design generates a pseudo-random number at the output for a certain input data. Thus this technique helps in generating a compact signature for a huge stream of input data and hence results in compression of data with a certain probability of aliasing. In our case, the input stream is 2048 bytes in size since a page is 2048 bytes in size and the output signature is 1 byte in size. Thus we achieve a compression of the factor of 2000X. Figure 9 shows the 8 bit MISR Design used in the architecture described in this section. A page of input data *In[7:0]* is shifted in byte by byte serially into the MISR design. If the page size is 2048 bytes, at the end of 2048 bytes, an 8 bit signature is obtained at the output *Out[7:0]* which is then stored into the spare blocks of memory.

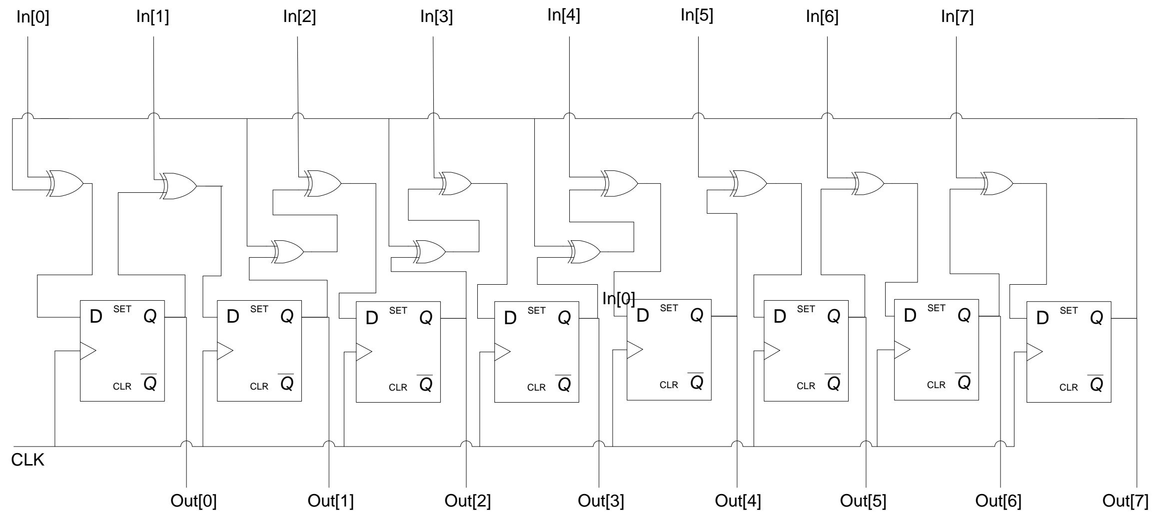

Figure 9: Eight Input MISR Design with P(x) = $x^7+x^4+x^3+x^2+1$

3.2 Scheme

Figure 10 shows the block diagram of the search scheme implemented in this study. The scheme is also described in details in [11]. The proposed scheme involves use of additional

hardware for write and search operations. The idea is to generate compressed signatures of each page when a page write comes in and store these signatures in a spare block. On the other hand when a search is triggered, the spare blocks which store the signatures are read and addresses of the matched data are decoded. Here we calculate the signatures of the data part of the page and ignore the 64 spare bytes that are used for ECC, wear leveling etc.

In order to generate a signature of the 2KB data that is written into the memory, a MISR signature generator is used. An 8 bit MISR signature generator with a polynomial $x^8 + x^6 + x^5 + x^4 + 1$ is used in the proposed design. The page data is fed to the MISR 8 bits at a time for 2048 cycles at the end of which an 8 bit signature is obtained at the output *Out[7:0]*. Thus 2048 bytes of data is compacted into just one byte using the MISR design. An 8-bit MISR shows an aliasing probability of ~ $1/(2^8)$ because it can generate only 255 unique signatures. The MISR design is used to calculate the signature of the write data as well as the search data. To select one of these, a multiplexer is used which has a select input of write/search.

After generating the signature of a written page, it has to be written into the signature block. If a write to signature block is triggered for every data write, the cycle time to program a page would almost double. To avoid this, we need to store data into a temporary buffer. This buffer helps in reducing the latency of writing into the signature block. We call this buffer as Sigbuffer since it stores the signatures of data along with their addresses. The buffer size is not fixed and varies with the memory size to optimize the area, power and performance. To avoid re-ordering during signature write, the buffer has been designed to be set-associative with fixed set of entries per block. For example if a block has 4 entries dedicated to it in the buffer, 4 signatures of the same block can be written into it before a write back to signature block is triggered. The

buffer stores the signature as well as address of each page. This page address is later used to encode the *implicit* address of the signature to be stored in the signature block.

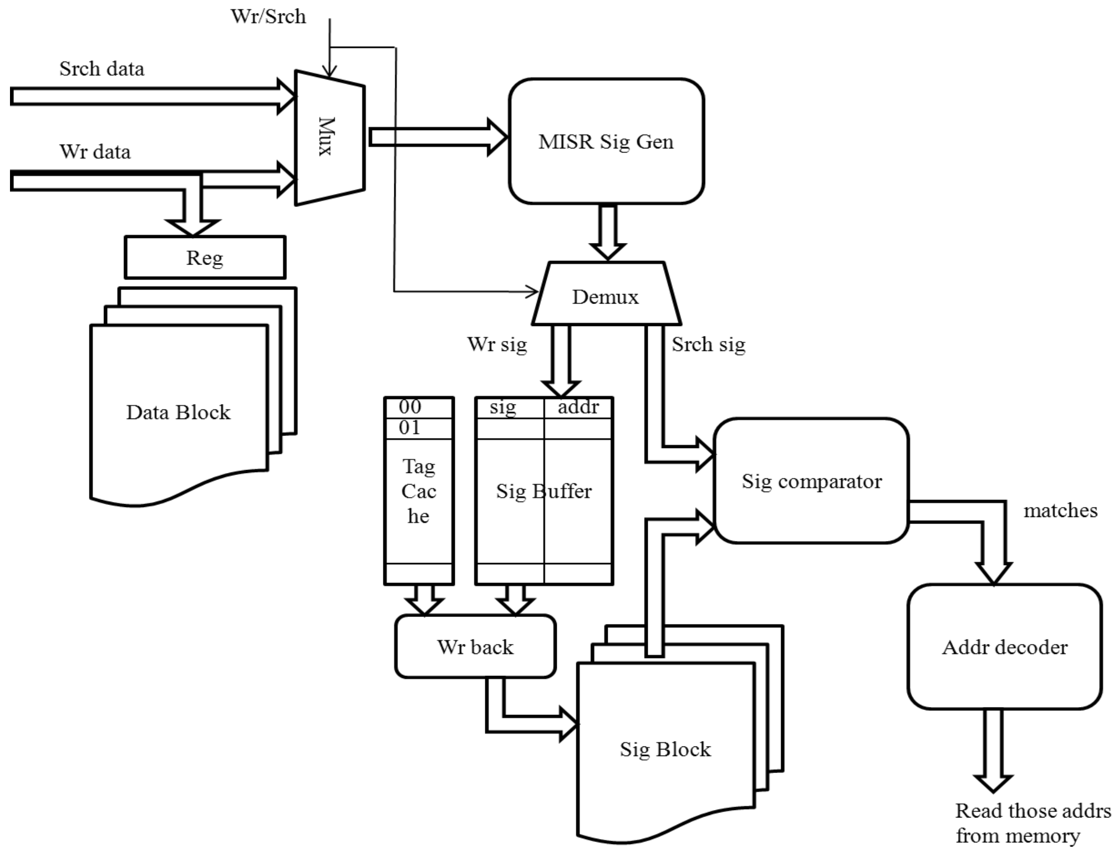

Figure 10: MISR-Based Architecture [11]

To minimize the latency and number of erases on the signature block, we use column addressing to write into the signature block. This means instead of programming a whole page each time, it is possible to write a byte of data in the signature block. This is useful because the signatures that need to be written into it are a byte in size. In spite of doing column addressing and writing signatures into the signature block using random data input command, it is preferable to have the signatures of a particular block sorted according to their page address before writing into the signature block. This helps in reducing the time to write the signatures when a write back mechanism is triggered. Implicit addressing also helps reduce storage for signatures. To sort the

signature entries in the buffer before writing them, we maintain a tag cache which keeps an entry of sort bits associated with the buffer entries. This avoids further need for hardware based sorting.

When a buffer set becomes full, the contents of the buffer are written into the signature block and the buffer is flushed completely. The write back mechanism encodes an implicit address to which each signature has to be written in the signature block, by using the information from the signature buffer and the tag cache. The number of spare blocks required to store signatures depends on the size of the memory and the signature length. A 2Gb memory comprising 2048 blocks with an 8 bit signature requires one spare block to store the signatures of the entire memory.

When a search command comes in, the search signal is given to the multiplexer which selects the search data and the MISR calculates the signature of the search data. The search data is assumed to be a multiple of pages. Once the MISR calculates the signature, a read operation is triggered which starts reading the signature blocks byte by byte. The signature comparator compares the newly generated signature of the search data with the signatures present in the signature blocks. If a match is found, the address decoder block decodes the address of the signature from its position in the signature block. This matching procedure continues till all the signatures have been compared. The decoded addresses of all the matches are stored and returned at the end of the search operation. The same buffer is re-used for storing matches.

3.3 Hardware Overhead

The hardware overhead of this scheme is the signature buffer, tag cache, MISR signature generator, signature comparator and some control circuitry. The signature buffer is 16KB in size for a 2Gb memory with 8 bit signature and increases with the size of memory and signature

length. The size of tag cache is 2Kb for a 2Gb memory with 2 bit comparators used for sorting. The MISR circuit comprises of 8 registers and few XOR gates for an 8 bit signature. The spare blocks used for storing signatures are assumed to be present in the Flash memory. The overall hardware overhead will be estimated by synthesizing this design as part of future work. From approximate calculations we can say that area overhead caused by the signature buffer (16Kb), tag cache (2Kb), MISR design and control circuitry (2Kb) will be 20Kb for a 2Gb memory. The exact numbers of area overhead will be deduced in chapter 7.

3.4 Experimental Framework

3.4.1 Platform

The architecture proposed in this paper has been implemented within a simulator written in C++. This platform is used to do performance analysis by varying various parameters like size of memory, signature length and buffer size. Performance analysis requires a workload. Since there are no publicly available standard workloads for flash performance analysis, we use a random process to create such a workload. A number of scripts have been tested as workload. A script generates an event list such as random read, random write, sequential read, sequential write, search after memory is written etc. These test cases help in estimating the performance of the design under different conditions. A global counter called ‘clock’ is used to keep track of the clock cycles consumed by each operation. The final value of the clock at the end of a simulation decides the number of clock cycles needed to execute the entire run through the test case. To get an idea of how the write and search operations work, we describe a pseudo code of these events as shown in Figure 11. The pseudo code for write starts with a check if the page to be written is empty to begin with. If yes, the incoming data is written into the memory through the page register. If this is not the case, it means the page has some 0s written to it and hence it has to be

erased first. Since a page erase is not possible, the block needs to be erased. Before erasing the block, the contents of the block are copied into another empty block. Once the block is in erased state, the new page is written into it and the old contents are copied back into the other pages.

```
Procedure: turboFSWrite(memAddr, memData)
{
if (emptyPage) write (memAddr, memData);
else {
copyBlock (currBlkAddr, newBlkAddr);
eraseBlock (currBlkAddr);
copyBackNewContents (currBlkAddr, newBlkAddr);}
signature = generateSignature (memData);
if (bufferNotFull) {
sortAndWriteToBuffer (signature, pageAddr);}
else if (sigBlkLocationEmpty) {
sigBlkAddr = decodeAddr(pageAddr);
writeBackToSigBlk (sigBlkAddr, signature);
flushBuffer();
sortAndWriteToBuffer (signature, pageAddr);}
else {
copySigBlkContentsToNewSigBlk(sigBlkAddr, newSigBlkAddr);
writeBackToSigBlk (newSigBlkAddr, signature);
flushBuffer();
sortAndWriteToBuffer (signature, pageAddr);}
}
Procedure: turboFSSearch(searchData)
{
signature = generateSignature(searchData);
do{compareSignatureWithSigBlkSigs(signature, sigBlkAddr);
if (match)
matchedAddr = decodeAddr();
return(matchedAddr);} while (!end of sig blks)
}
```

Figure 11: Pseudo Code of TurboNFS [11]

This is how the normal write command is executed. In the new architecture, this write is augmented by another set of events. First a signature of the memData is calculated. If the buffer location corresponding to this page is empty, the signature and page address are written into it else the buffer contents are written into the signature block, buffer is flushed and then written with the new signature and page address. If on the other hand, the signature block location corresponding to this address is not empty, the signature block contents are copied into a new

signature block along with the new signature written into the appropriate location. Entering into each of these subroutines costs us additional clock cycles.

The search operation starts with a signature generation of the search data. This signature is then compared with every signature in the signature block(s) and if a match is found, the address corresponding to that signature is decoded and returned. Hence at the end of search operation, all the addresses corresponding to matched data are returned.

3.4.2 Experiments

The main objective behind designing the flash memory simulator is to do performance analysis of the proposed architecture. This can be done by varying the control variables and analyzing their impact on the performance.

The first control variable is the size of the memory itself. For bigger memories, we keep the page and block size constant and increase the number of blocks in the memory. It is intuitive that as the memory size increases, it takes longer to test the memory.

The second control variable is the size of the signature buffer used. If the buffer is large and the number of entries available per block is also large, it will take longer for a buffer set to be full. This means that the write back procedure which writes the buffer contents into the signature block and flushes the buffer will be called less often and hence the performance will improve. However this requires a large buffer that adds more area overhead. Hence for every memory size, simulations have to be run to find the optimum buffer size which minimizes the area without large impact to the performance. The third control variable in this architecture is the signature length. If the signature length is really small, probability of aliasing will be more and there might be erroneous matches returned when a search operation is requested. On the other hand if the signature length is made large to minimize the errors, the number of spare blocks

required to store this signature will increase. Hence it is essential to optimize the signature length for a memory of particular size. When experiments are run on the flash memory, the read performance is not measured because the proposed architecture does not affect the read time at all. The write and search performance however is measured each time a control variable is changed because the write and search performance depends on these variables.

3.5 Results and Analysis

In Figure 12 the proposed search method is compared against the normal search method. The plot shows that Turbo NFS is significantly faster than regular NAND flash search and the benefit increases with the memory size. For 16Gb flash, the performance gain is 2 orders of magnitude.

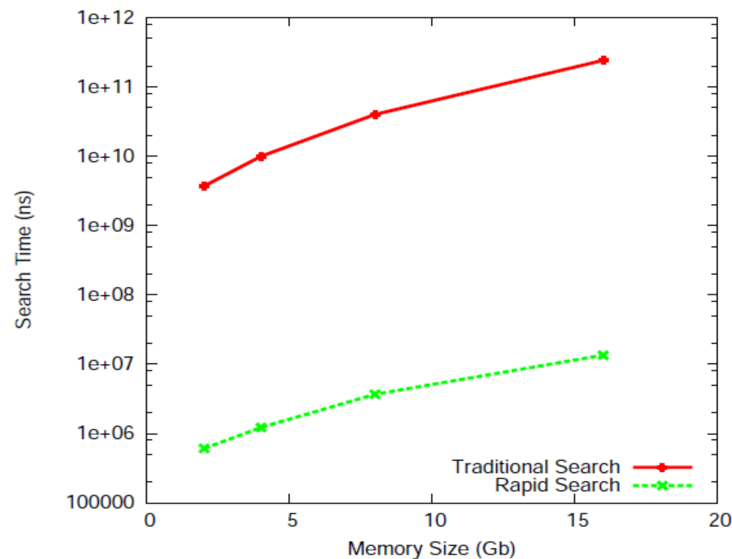

Figure 12: Search Gain with TurboNFS

Next we analyze the effect on write performance latency due to the new method. Figure 13 shows the write performance latency comparison for sequential writes. When a sequential write request comes in, the buffer configuration is changed and it is no more set associative. Sequential incoming entries are directly written into the buffer and hence the *writeBack* routine will be

called only when the buffer is completely full. If the buffer size is 2048 locations, writeBack will be called after every 2048 writes. This does not cause any overhead on write operation. As seen from Figure 13, there is no write penalty for sequential writes.

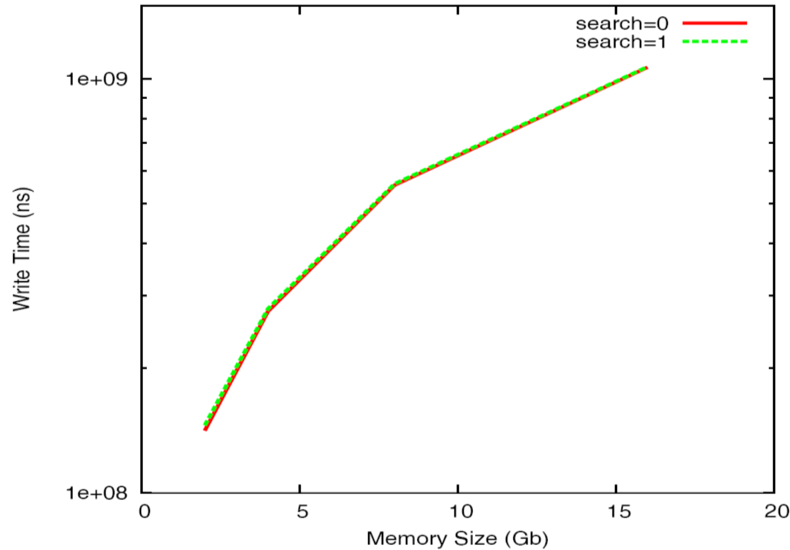

Figure 13: Comparison of Sequential Write with and without TurboNFS

We also analyze the effect on write performance for random writes on the memory. This comparison is done for varying memory and buffer sizes. Figure 14 shows the penalty in latency in write performance due to the TurboNFS method. The plots show performance vs memory size for buffer set sizes of 4, 8 and 16. When compared to sequential writes, the effect on random writes due to the new method is different. This is obvious because random writes write into different blocks of memory and hence the buffer sets get filled up faster than they do in sequential write. Hence the write back procedure is called more frequently and the average write time increases. As seen from Figure 14 the write penalty is approximately 8-9% for random writes.

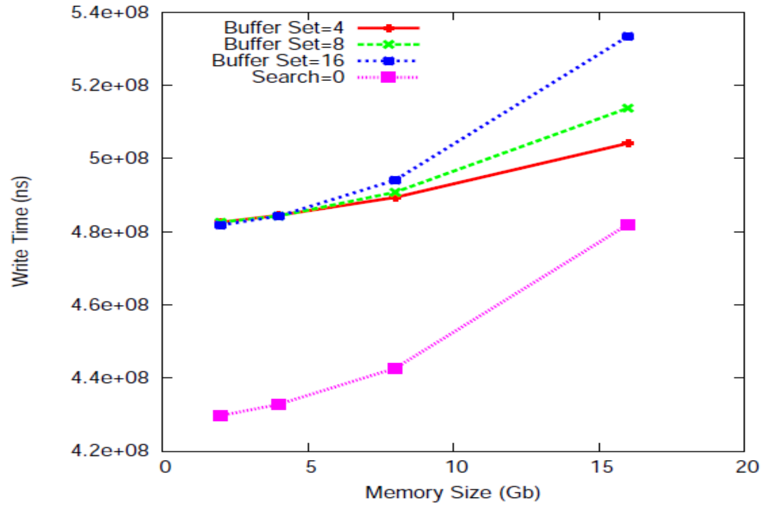

Figure 14: Penalty on Random Write due to TurboNFS

Finally, we analyze the effect of change in signature length on the probability of aliasing, and the number of spare blocks required to store the signatures. The graph shown in Figure 15 helps in deciding the optimum signature length to minimize errors in searches and also to minimize the number of spare blocks used for a memory of particular size. This permits a designer to select a signature length on the basis of required accuracy and target hardware overhead. For example, for a memory of size 16Gb the best signature length is the intersection of the red and light blue line which is approximately equal to 18.

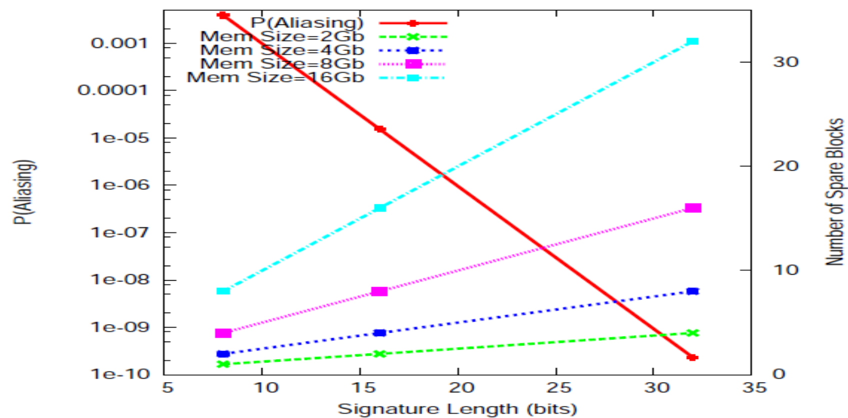

Figure 15: Effect of Signature Length

CHAPTER 4

DCT-BASED ARCHITECTURE

4.1 Introduction

Discrete Cosine Transform method is a well-known compression technique used for images, audio and video files because it has a strong 'energy compaction' property. In a DCT, most of the signal information is concentrated in a few low-frequency components of the DCT. Hence storing these frequency components only is sufficient to retrieve the original data. The need for DCT-based signature generations arises because DCT-based technique can give partial matches for a search request unlike MISR-based method. MISR method gives a match only if every single byte of the search string matches with the memory content. This is not likely to happen in case of huge files such as image and multimedia files. Hence the MISR based technique would give zero matches even if the search data is almost equal to the data present in the memory. This motivated us to develop a DCT-based compression technique which is capable of giving partial data matches.

4.2 Scheme

The block diagram of the DCT-based method looks similar to the MISR-based method except the MISR block and the Sigbuffer block. The new block diagram is shown in Figure 16 and is described in details in [12].

The DCT block performs a discrete cosine transformation on a set of incoming pages which means the frequency coefficients are calculated for these chunks of pages. For experimental purposes this number is set to 4. Hence a DCT is performed on every 4 incoming pages. In MISR technique we stored signature for every page. Here we are trying to increase the

compaction ratio and hence the granularity for performing DCT is 4 pages. There are different types of DCT used for different applications.

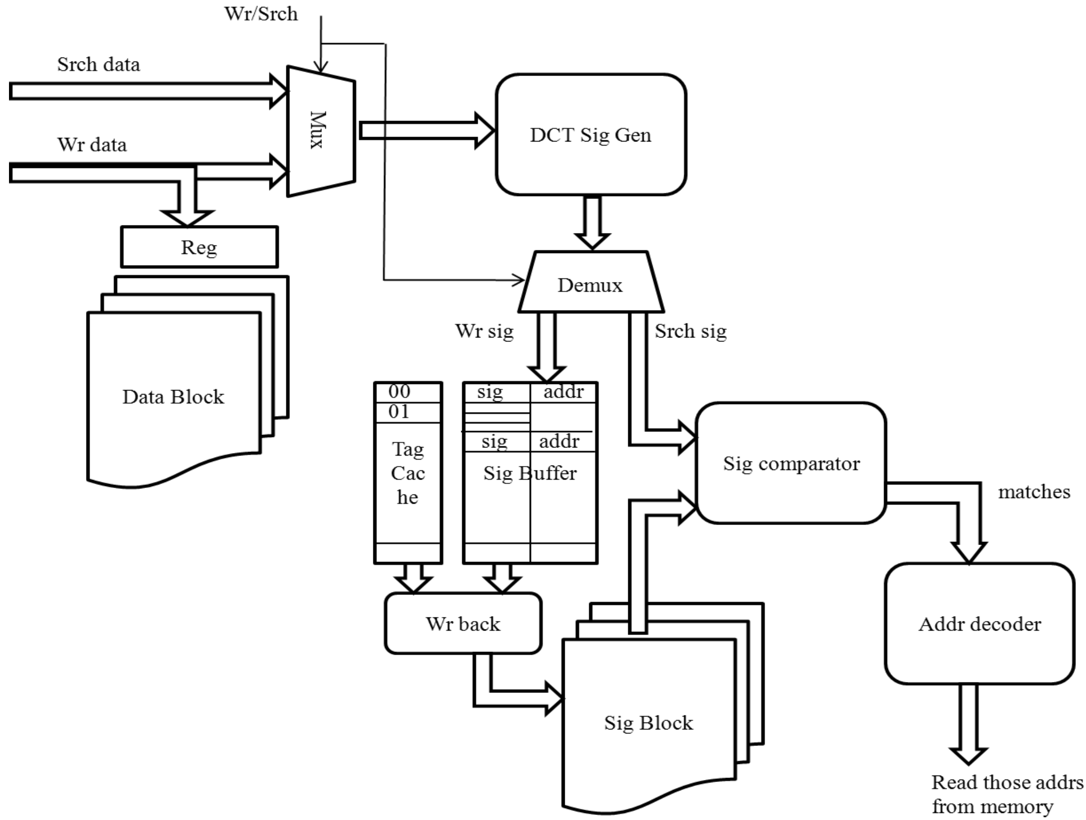

Figure 16: DCT-based Architecture

We use the 1 dimensional DCT-II which is the mostly commonly used form of DCT and which is good for compressing JPEG files. The transform is as follows

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad k = 0, \dots, N-1.$$

Where Xk is the frequency domain coefficient and xn is time domain coefficient of x. In our case we find DCT for 4 pages which is equal to 4*2048 bytes = 8192 bytes. Thus N=8192 and k also goes from 0 to 8191 which means we get 8192 coefficients in the frequency domain out of which only a few are dominant and should be stored. Thus we sort these coefficients based on their

absolute values and store only the top few coefficients along with their positions. The DCT block in Figure 16 does this computation.

In MISR technique the signature generated by the MISR block was stored in the sigbuffer corresponding to its page address. In this case we don't have a one to one mapping of page address and signature in the sigbuffer. Instead we store a 4 byte wide signature consisting of top two coefficients and their positions corresponding to its page address. Also, since we calculate signature for every 4 pages, the sigbuffer block will not have entry for each page but will have page addresses like 0, 3, 7, 11 and so on. Though the length of sigbuffer will reduce due to above reason, the width will increase. This is due to the fact that instead of storing a 1 byte signature, now we are storing 4 bytes of DCT coefficients and their positions. Thus the overall size of the sigbuffer remains same unless we want to store more coefficients to increase the accuracy of matching. The sigbuffer design is also shown in Figure 16.

Other than the DCT and the sigbuffer block, the functionality and flow of the DCT based technique remains same as the MISR based technique.

4.3 Hardware Overhead

The hardware overhead of the DCT based scheme is the signature buffer, tag cache, DCT signature generator, signature comparator and some control circuitry. The signature buffer is 16KB in size for a 2Gb memory if only 2 top coefficients and their positions are stored and if the DCT is performed on every 4 pages of data. If more coefficients are to be stored in order to increase the accuracy of the method, the sigbuffer size will increase. The size of tag cache is 2Kb for a 2Gb memory with 2 bit comparators used for sorting. The DCT block comprises of few adders, multipliers and a cos function generator. The spare blocks used for storing signatures are assumed to be present in the Flash memory. The number of spare blocks required to store the

DCT coefficients also depends on the size of the memory. The bigger the data memory, the more spare blocks required to store the DCT signatures. Also, for storing more coefficients, more spare block data would be required. The overall hardware overhead will be estimated by synthesizing this design as part of future work. Approximate hardware overhead would be 0.02% of the total memory area.

4.4 Experimental Framework

The simulator implemented in C++ for the MISR based technique is used for the DCT based technique as well by doing some modifications in the environment. The input data that is given to this scheme is real data generated from image files. Software called ImajeJ is used to convert an image into a 2D array of numbers which are actually the pixel values. This array of numbers is given as input data to the simulator and stored into the flash memory.

4.5 Results and Analysis

4.5.1 DCT on an Image File

Figure 17 shows an image which is given as input to the DCT-based scheme. A part of the image is converted into pixel values and given as a search request to the simulator. This stream of numbers is then converted into DCT coefficients and searched in the spare blocks. If a match is found, it shows that the search mechanism works correctly. If an exact part of the image is given as search request, a 100% match is found. If instead a part of the search stream is changed, the output shows percentage of data match instead of a 100% match. This shows that partial matching of data is possible with the DCT based method. Figure shows a 2048x1024 size image that was converted into text format and given to our simulator. Since one page is 2048 bytes in size, there were 1024 writes for the following image in order to store the image completely in the memory. Since one block consists of 64 pages, the following image was completely

accommodated in 16 blocks (1024/64=16). The DCT coefficient information is stored in the spare block. Figure 18 shows a part of the image shown in Figure 17 which is converted into text and given as search request to the simulator. Since this part exactly matches the one in the original image, the output of the experiment is "100% Data Match found" and the addresses of the matched locations are stored in an output file and returned as output of the experiment.

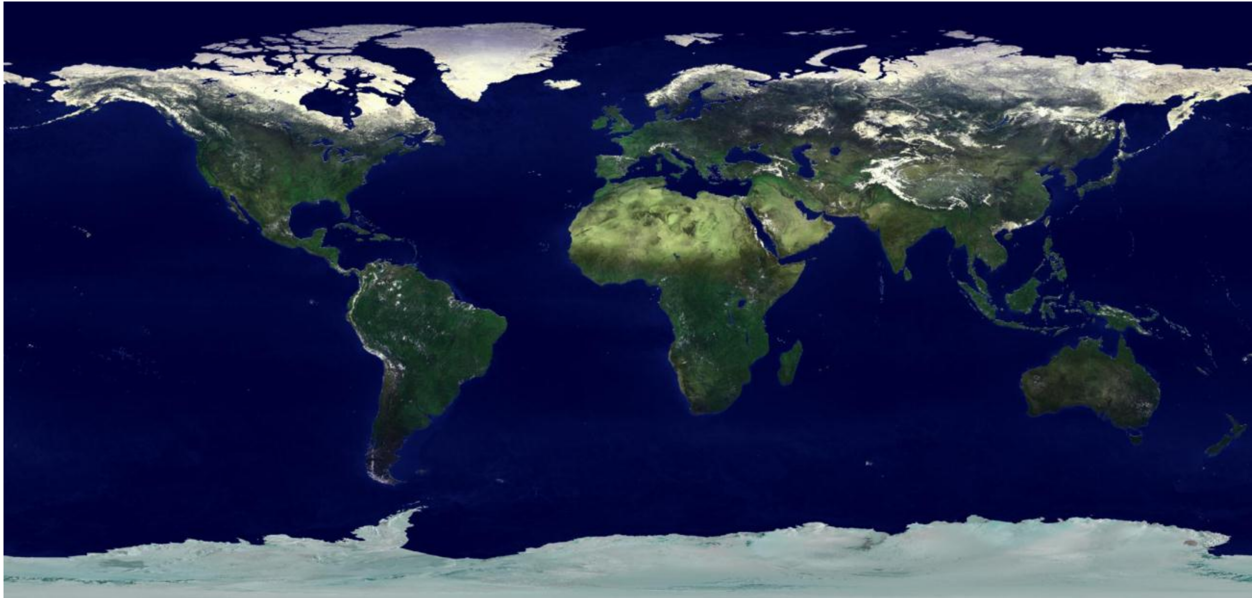

Figure 17: Image written into the Flash memory

If on the other hand, the search request shown in Figure 18 (b) is given to the simulator, it does not give "perfect match". This is because we took a part of the original image and distorted it. Thus the output of this experiment is "71% Data Match Found" and the addresses for which data is matched are written into a file and returned as output.

Figure 18: (a) Search Request which is part of image (b) A distorted search request

4.5.2 Timing Analysis

Different testcases such as sequential read, sequential write, random read, random write, search after write etc. were run on the DCT scheme similar to the experiments run on MISR scheme in order to do the timing analysis. We saw that the only difference between these two schemes is the DCT block and the sigbuffer block. If we consider 2 pairs of coefficients and their positions to be stored in the sigbuffer and the spare block for every 4 pages of data, the sigbuffer configuration also remains same. Hence the only change now is in the DCT block. The MISR block takes 2048*8=16384 cycles to generate one signature. The DCT block however takes 8192 cycles to generate coefficients and 8192^2 cycles to sort these coefficients. Hence the total cycles required for a DCT signature to be calculated is 67117056 cycles which is much more than the cycles required to calculate MISR signature. But this is just an absolute increase in the number of cycles required for signature generation. Since the remaining blocks and the flow remain same, the timing graphs look similar to those shown in section 3.5. These graphs are shifted up because of DCT blocks consuming more cycles but the shape of the graphs remain same. Hence they are not shown here separately.

4.5.3 Spare Block Estimation

Figure 19 shows the number of spare blocks required for different coefficient sizes. It looks similar to one shown in section 3.5. Here we consider that these coefficients are calculated for every 4 pages of data.

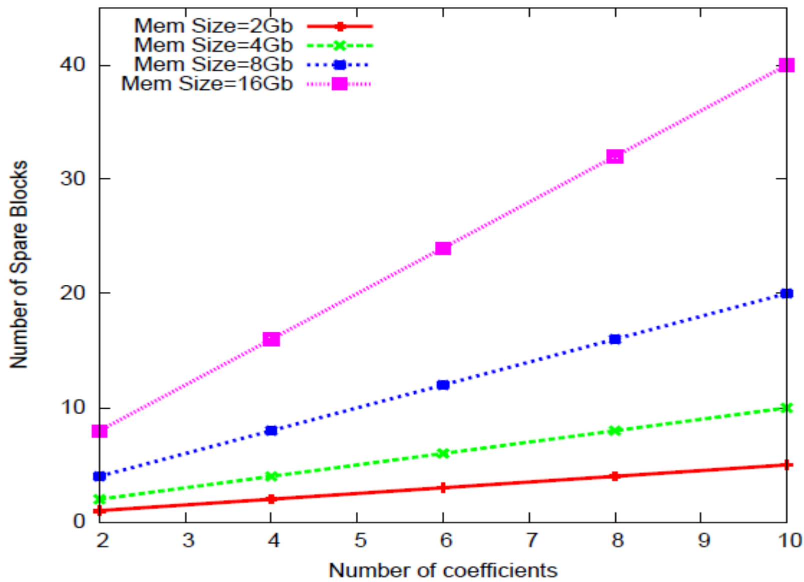

Figure 19: Increase in Spare blocks with number of DCT coefficients

If we want to increase the granularity and freedom of search, this minimum number of pages has to be reduced. This will in turn increase the number of spare blocks required to store the DCT coefficients. Figure 20 shows the number of spare blocks required for a 2 Gb memory if the DCT is performed on different sizes of data and only the top 2 coefficients are stored. Figure shows that one spare block is required to store 2 coefficients if the DCT is performed on 4 pages of data. The number of spare blocks increases exponentially as the number of minimum pages on which DCT is performed reduces. It is clear from the graph that the optimum number of pages on which a DCT should be performed is 1 page (2048 bytes) which would need 4 spare blocks to store the coefficients. But if we want to allow really small search requests, this minimum limit has to be reduced further, for example 512 bytes which would require 16 spare blocks to store 2 coefficients for a 2Gb memory.

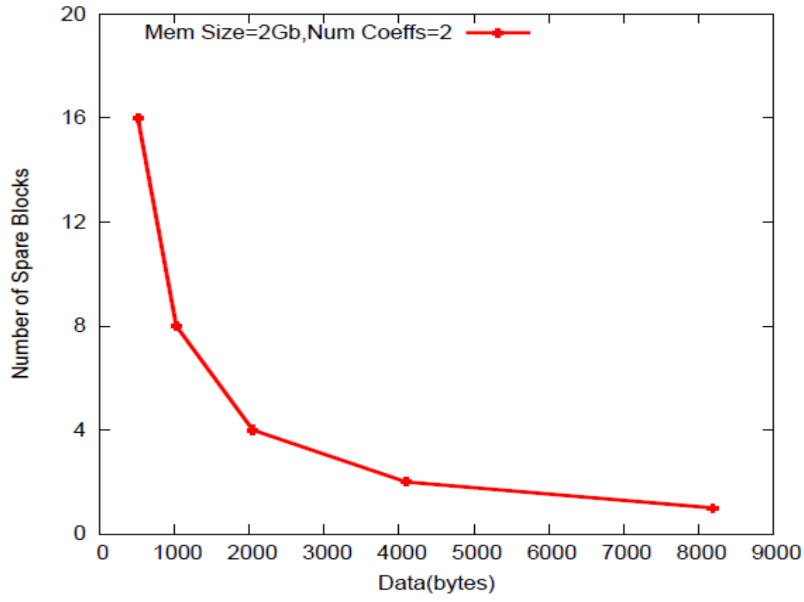

Figure 20: Effect on Spare blocks with the size of data on which DCT is performed

CHAPTER 5

EFFECT ON PRODUCT LIFETIME

Erases on NAND Flash memories can be quite expensive. The total number of erases on a flash memory before it becomes unusable is usually of the order of 100,000 for SLC (Single Level Cell) NAND Flash memories. Hence the erases on flash memories should be done carefully. In a NAND flash memory, if a page is not in erased state and a write request comes in for this page, the new data cannot be written on the page without an erasure. This is because erasing a block means writing all 1's in it and writing data means writing a pattern of 0's always. Thus a write always has to be done on an erased page which contains all 1's. This means that a write cannot be done on a previously written page or a block. Since a page cannot be erased, the whole block has to be erased. Similarly, if a signature for a page is written into the signature block and a new signature has to be written in the same location, the signature block needs to be erased. If the total number of erasures on different pages of the memory are E, the signature block has to be erased E times. This reduces the life of the signature block. For this reason, we use multiple spare blocks as signature blocks to minimize the number of erasures on the same spare block. For a memory greater than 2Gb, multiple signature blocks are required to store the signatures and hence the probability of erasing the same signature block reduces. A threshold is set for the maximum number of erases on a spare block. The moment this threshold crosses, the old spare block is made a data block and a new block is chosen as a spare blocks. Thus the number of erasures on the memory are minimized which helps in increasing the lifetime of the blocks and in turn the whole memory.

CHAPTER 6

EFFICIENT STORAGE USING COMPRESSION

6.1 Introduction

In the above chapters we concentrated on using different compression techniques to expedite the process of data search in a flash memory. The focus till now was on improving search speed using compression. This chapter will use the same compression techniques for a different application namely to improve the data storage capability of a flash drive. One may wonder how the capacity of a flash memory can be increased by keeping its size constant. We are effectively trying to improve the capacity of a flash memory by not increasing its size but by compressing the existing data and making space for new data so that in the same amount of area, more data can be accommodated. This is possible because of the fact that the data written into any memory is not always unique. It is possible to find redundancy in this data and if we manage to find it, we can use it to our advantage and increase the storage efficiency. The common name for this kind of data storage is called data deduplication. It is widely used in companies where lot of similar data is present on servers due to data backups provided by the companies. When a backup of certain data is stored and the new files are changed, most of the times the changes done to the files are minimal and a user ends up storing lot of redundant data. To avoid this unnecessary duplication of data, the data deduplication techniques are very extensively used now-a-days. In this chapter we will see how we use our existing MISR-based compression technique to achieve data deduplication and increase storage gain of a NAND flash memory.

6.2 Data Deduplication

Data Deduplication literally means avoiding any duplication while storing data into memory. It identifies redundant data, eliminates all but one copy, and creates logical pointers to the

information set so that users can access the material as needed. It reduces the volume of data stored [13]. A class of deduplication systems splits the data stream into data blocks (*chunks*) and then finds exact duplicates of these blocks. Most common technique used for data deduplication is called *hashing or fingerprinting*. Let us consider the example shown in Figure 21 to get a better idea of data deduplication.

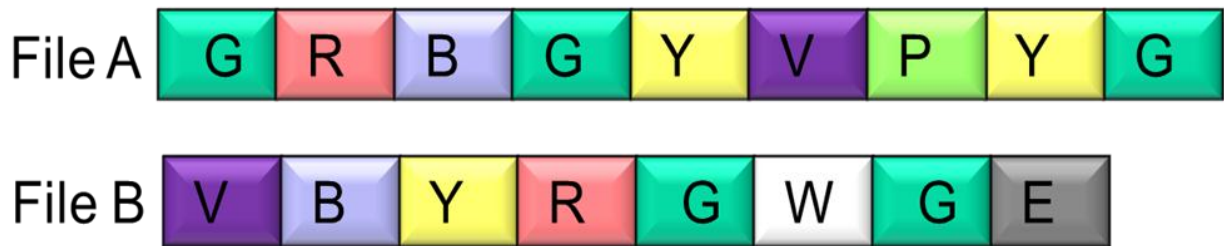

Figure 21: Example of Data Deduplication [13]

File A is first stored into the memory. While storing it, it is divided into different logical blocks called R, G, Y etc. A new file B has to be written into the memory. Now instead of writing this file completely into the memory, it is again divided into blocks and compared with the blocks of data already present in the memory. In the above case all the blocks except A and L are already present in File A. Hence logical pointers are created for these blocks of data and only blocks A and L are actually written into the memory. This technique helps is saving lot of memory and prevents from duplication of data and hence is called Data Deduplication. Now instead of storing the blocks of data as it is, a fingerprint (hash or a signature) is created and stored instead. When a new write request comes in, this hash table is read instead of reading the entire memory. If matches are found, logical pointers are created for that data, otherwise new data is written into the memory along with its corresponding signatures. In case of data deduplication, signatures are stored in a cache for ease of access and in our case signatures will be stored in a header page.

There are two types of data deduplication methods – 1. Identity-based Data Deduplication [14] and 2. Similarity-based Data Deduplication [15]. The MISR-based method corresponds to identity-based data deduplication because both methods look for exact matching signatures or hashes only and the chunk size (or minimum data used to calculate signature) is small. The DCT-based method corresponds to Similarity-based Data Deduplication where partial matches of signatures are considered too and the chunk sizes are huge. The search mechanism developed in this research will be modified in order to add the data deduplication feature for the MISR-based technique alone. Hence we will implement only the identity-based data deduplication mechanism in this research.

6.3 Data Deduplication for Flash Memory

This section describes how the identity-based data deduplication method is implemented using our MISR-based compression technique. For this implementation we disable the search feature of our scheme. Hence the flash memory will not have any signature block or signature buffer to store the signatures of the data. We will generate these signatures, but store them in a different fashion now.

6.3.1 Implementation

In the new scheme of data storage in a flash memory, we assume that there is no search taking place on the flash drive. Thus the signatures of pages are not stored in the signature block as before. Instead we now store the signature local to every block. Every data block of the memory will consist of a header page which will not contain any data that was earlier stored in it. The header page is always the first page of a block and the new flash memory structure with these header pages is shown in Figure 22. The header page will contain information about all the remaining pages in the block which are essentially data pages. The size of a header page is fixed

which means that the information about data deduplication will never overflow to any of the data pages. The duplication is detected only within a block. This is described in the assumptions section.

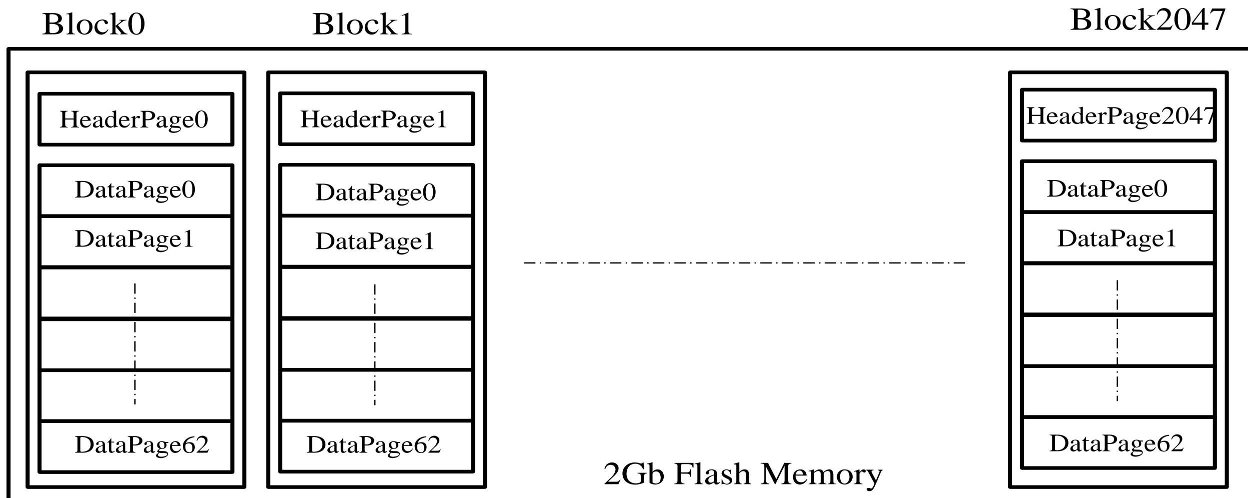

Figure 22: 2Gb Flash memory Structure with Data Deduplication Scheme

The header page inside a block contains the information of pages present in that block alone. Hence deduplication is done local to a block. The structure of a header page present inside a block is shown in Figure 23. The first field of a header page is 8 bit wide and is reserved for the signature of a page. Second field is a counter which determines the maximum number of pages that can share the same signature. This counter is 32 bit wide and hence only 32 pages to the maximum can be shared. The third field is reserved for the addresses of the pages that are shared and have that particular signature. This information is for 1 page. Similar information will be stored for all the pages in the block. Thus the header page will consist of entries for all the data pages which will be (8+32+6*32)*63 bits of data. 8 bits for signature, 32 bits for the counter, 6 bits per page for 32 shared data pages and together (8+32+32*6) bits for all the data pages are assigned inside a header page. Thus it comes out to 1827 total bytes. The size of a header page is 2048 bytes and hence this data will easily accommodate in 1 header page. However if we try to

increase the total possible shared pages more than 32, we would have to store page addresses for the extra pages too and it will exceed 2048 bytes. Hence the maximum shared pages are limited to 32 so that not more than 1 header page is required for this scheme. It should be noted that not all the fields in the header will always be filled. If the block has no duplicate pages, a signature will be written for every page and the counter and page address fields would be all zeros. However for duplicate pages in a block, the counter and page address fields would be written and at the same time not all signature fields would be filled.

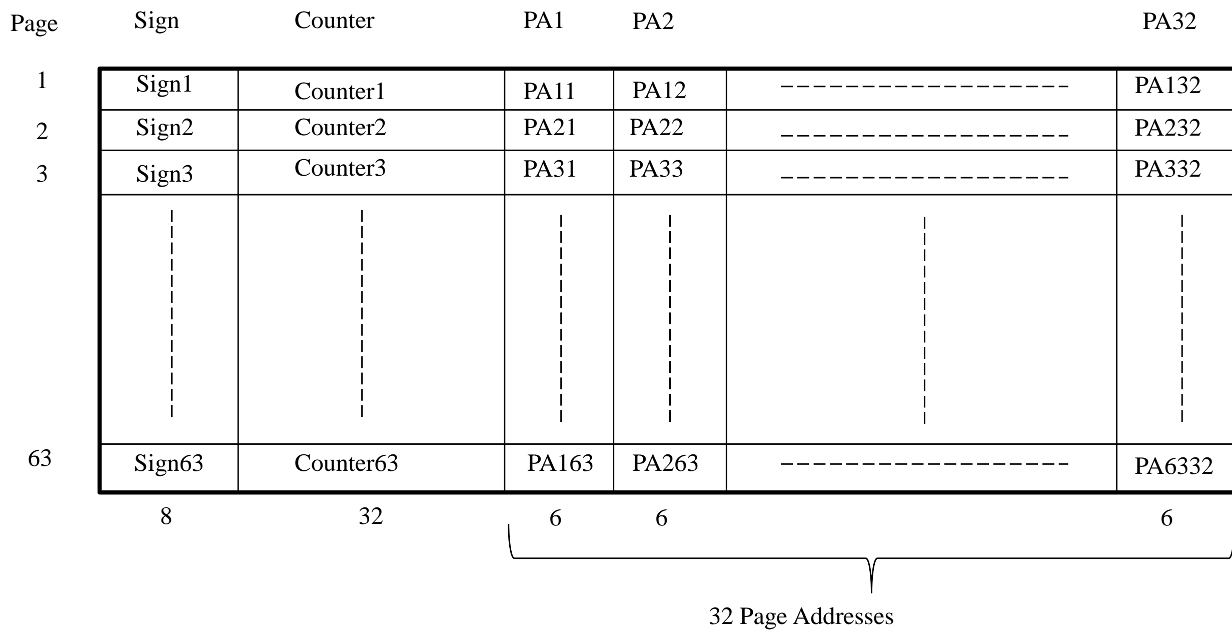

Figure 23: Header Page Format

6.3.2 Assumptions

Following is the summary of some of the assumptions that are considered while designing the header pages in the new flash memory structure.

1. The new scheme is not integrated with the search scheme which was implemented earlier. Thus with the data deduplication scheme active, there cannot be any search performed on the flash memory.

2. Header page is always assumed to be the first page of a block. This is just to give a regular structure to the scheme and for ease of implementation.

3. Data addressing of the flash memory is changed considering that the first page of any block is never the data page. This is due to the fact that now the first page of every block is a header page and hence data cannot be stored in the 1st page of any block.

4. The size and format of a header page is fixed. Bits are allocated for each field inside a header page and if there is no duplication present in the block, these bits will remain empty but still be a part of header page.

5. Duplication is checked within a block of the flash memory. This means if a page in block 0 matches exactly with a page in block 4, they are not considered as duplicates and are still stored separately since they belong to different blocks. This is done because the overhead of implementing deduplication across blocks is a lot and will reduce the storage efficiency intended by the new scheme.

6.4 New Memory Operations

This section shows how the different memory operations namely read, write and erase are changed with the new data deduplication scheme. Figure 24 shows the pseudo code of new read and write operations and Figure 25 shows the flowchart of the read, write and erase operations in the our experimental framework.

6.4.1 New Write

The write operation for the new scheme will change due to the introduction of header pages in the memory. The pseudo code for new write is shown in Figure 24. For better understanding, Figure 25 can be referred to see the flow of memory write operation. The first step for a write is to find the signature of the data to be written into the memory. Once the signature is

ready, it is compared to all the signatures in the header page of the block to which this page is intended to be written.

```
// Pseudo Code for New Write
Procedure DDWrite (sign, blkAddr, pageAddr)
{
  generateSign(sign);
  do{
    if (signMatch) {
      readPageFromMemory();
      if (pageMatch) {
        incrementNumSharedPages();
        writeNewPageAddr(pageAddr);
      }
    }
    else {
      writeNewSign();
      initializeNumSharedPages();
      writeNewPageAddr(pageAddr);
      writePageToMemory();
    }
  } while(!(endOfHeaderPage));
}

// Pseudo Code for New Read
Procedure DDRead (blkAddr, pageAddr)
{
  do {
    if (pageAddrMatch) {
      return firstPageAddrFromEntry;
    }
    readMemory (blkAddr, firstPageAddrFromEntry);
  } while (!(endOfHeaderPage);
}
```

Figure 24: Pseudo Code for New Write and Read

If no match is found, it means that the page contains new data with no duplication with the existing data in the memory. In this case, the signature of this page is written into the header page with the page address where this page is actually being written. The counter value for this entry in the header page will remain zero since it is a unique page and is not shared with any other page. After this the actual page is written into the block to the desired address. If on the

other hand, the signature of the page to be written matches with an existing signature, the counter for that header page entry is incremented and the page address of this page is written in the page address field of the header page. This is done only after confirming that the page data matches exactly with the new data to be written. This can be done by reading the page corresponding to the matched signature. This extra check is needed to avoid errors in writing due to aliasing. After this confirmation we can say that the page already exists in the memory and hence need not be written once again. Thus we save space by not writing duplicate pages in the memory. However we can see that the write time of a page is increasing due to the new scheme. Instead of simply writing the page, we first generate the signature, read the header page and then write the data. The details of overhead in performance due to new write will be discussed in results section.

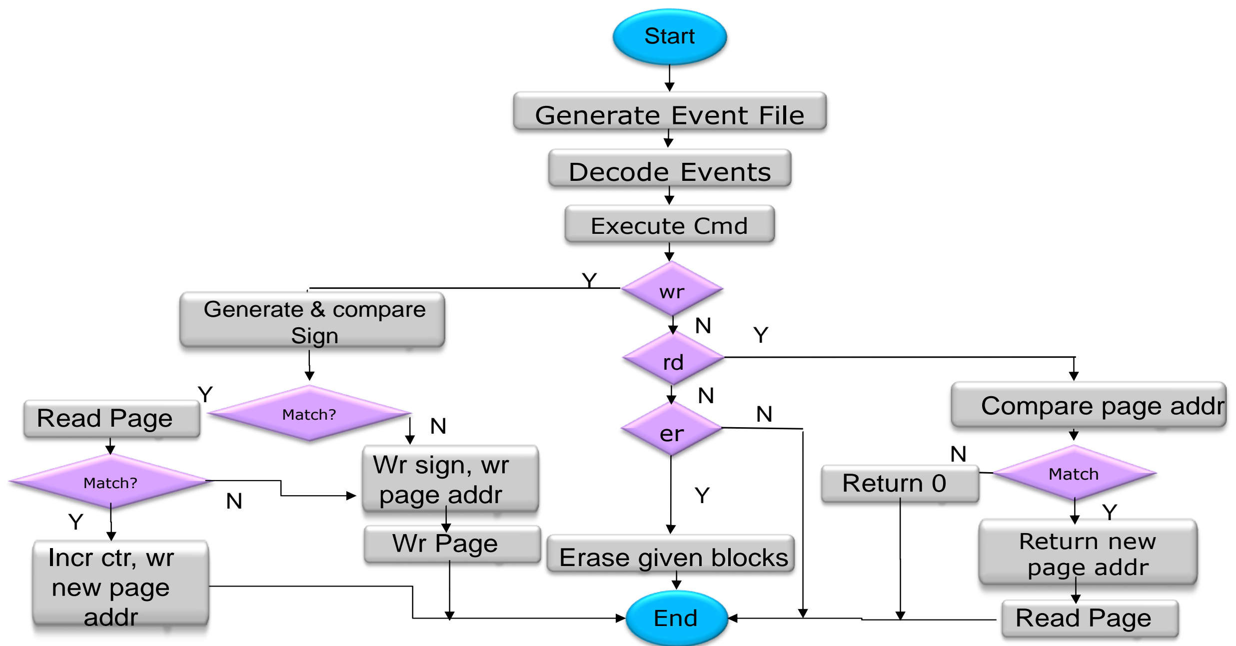

Figure 25: Flowchart for New Scheme with Data Deduplication

<u>6.4.2 New Read</u>

The pseudo code for new read is also shown in Figure 24. The read operation with data deduplication will also change. This is because not every page in the memory now contains the

data. Some locations contain pointers to the address where the data actually is. Hence if we directly try to read a page, we might end up reading just a pointer and not the real data. Thus the first step to do in a read operation is to search for the address of the page to be read in the header page. If a match of the page is found, the first page address in that entry of the header page is noted and corresponding data from that page address is read. This is because the first page address in the header page entry actually contains data and the remaining page addresses are just pointers to this data. Thus the overhead with respect to performance on a read is reading the header page and decoding the address from which the page should be read in order to get the right data.

6.4.3 New Erase

The erase operation will remain the same for the new scheme. Once an erase command is given, the complete block of data is erased and filled with all 1s including the header page. Thus for future writes to this block, the header page programming will start from scratch. Thus there will be no performance penalty due to erase operation on the flash memory.

6.5 Results and Analysis

In this section we will discuss the effect of the new scheme on the basis of different parameters and judge whether this scheme is suitable for the NAND flash memory design. The effect of this scheme on storage gain, performance and aliasing probability is discussed.

6.5.1 Data Deduplication and Storage Gain

The primary objective of introduction of the data deduplication scheme on a NAND flash memory is to achieve compression in data storage and hence store the data effectively in a flash drive. According to the configuration of the header page as discussed in the previous section, only 32 identical pages can be shared in one location of the header page. In a block of 64 pages,

if all the pages are identical to each other, with the new scheme the actual write will occur on 2 pages instead of all 64 pages. Hence the maximum storage gain that can be achieved for 100% duplicate data in a block of memory is 32X. As the percentage of duplication reduces, the storage gain also reduces exponentially. If we try to write unique data to a block of the memory (all 64 pages are unique and different from each other), there will be no gain in terms of storage and we will end up writing all 64 pages including the header page. Thus the minimum gain of this scheme is observed when the data to be written is unique in nature. Figure 26 shows a plot of increase in storage gain with duplication and prove that maximum gain possible with this scheme is 32X.

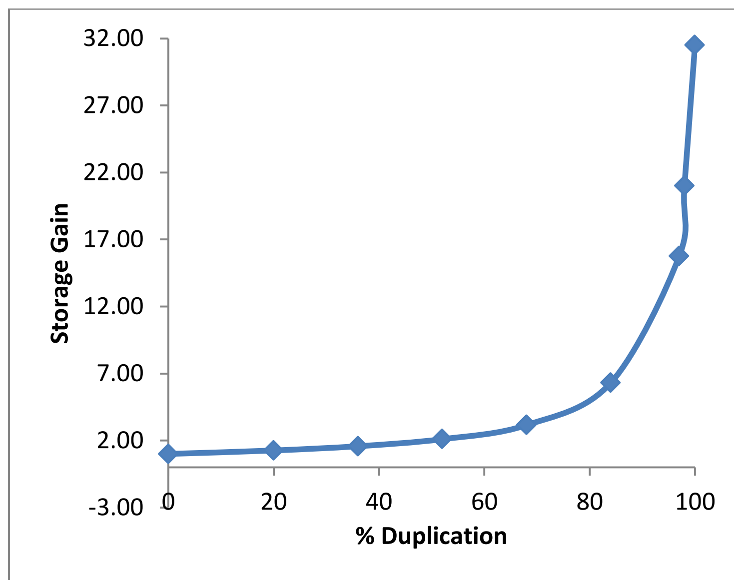

Figure 26: Increase in storage gain with duplication

Figure 26 gives a general picture of the storage gain for any case. Another experiment was performed by considering a specific case and the results observed are shown in Figure 27. A file A of size equal to 30 pages was written into a block of memory. All 30 pages in file A were unique and hence will be written actually into 30 pages in the memory. Then a file B of same size was written in the same block. Initially file B was exactly same as file A and hence all 30

pages of file B were not actually written into the memory but were just updated in the header page of the block. When the contents of file B were changed gradually, the number of pages to actually store the two files increased since the duplication decreased. This behavior is shown in Figure 27. Without the presence of data deduplication, 60 pages would be required to store the two file always. Thus the two curves meet at a point when file A and file B are completely different. In this experiment, if file A and file B were parts of different blocks, no storage gain will be observed. If file A or file B were partially in the same block, there will be some storage gain but the percentage of storage gain will be lesser than that observed in Figure 27. Thus for this case, the blue curve would shift up and more pages will be required to store the files.

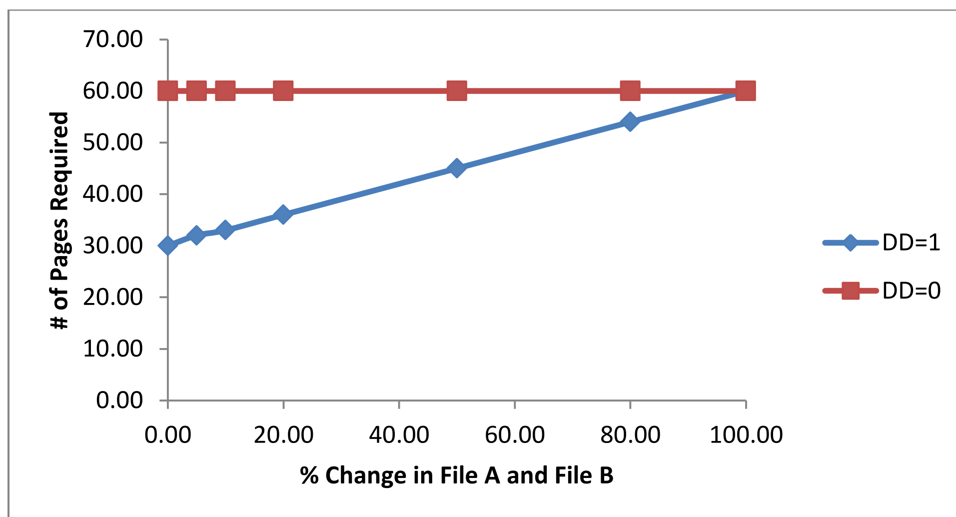

Figure 27: Number of pages required to write File A and File B

<u>6.5.2 Effect of Data Deduplication on Performance</u>

The new data storage scheme adds extra cycles for the read and writes operations. This is due to the fact that instead of reading or writing a page directly, we first read or write into the header page. We also need to generate a signature of the data to be written so as to compare it with the existing signatures in the header page. All these operations result in some extra cycles. Here we discuss how these changes affect the read and write performance of the flash memory.

We saw that the erase operation is unchanged and hence there will be no effect on the time to erase a block using the new scheme.

6.5.2.1 Write Time

The write time due to the new scheme is expected to increase due to the header page reading and writing. However it is not the case always. It depends on the percentage duplication in the data to be written into the memory. When a page to be written is a duplicate, we don't actually write it. We just read the header page and find if it is a duplicate and write the address of this page in the header page entry. Hence instead of writing 2048 bytes of data into the memory we are just writing 6 bits of page address into the header page. This is the reason why we see the write time to be less than the original write time without data deduplication scheme in Figure 28 for higher percentages of duplication. Thus we can say that for high duplication percentages, the new scheme is very suitable since it increases the storage gain and at the same time reduce the write time. This advantage however reduces with the reduction in duplication of data. This is because as the percentage duplication reduces, more and more pages are unique and hence need to be actually written into the memory and not just the header page. In the sample considered for Figure 28, the write time with and without the scheme coincides at ~30% duplication. But this percentage will vary depending on the sample of data considered.

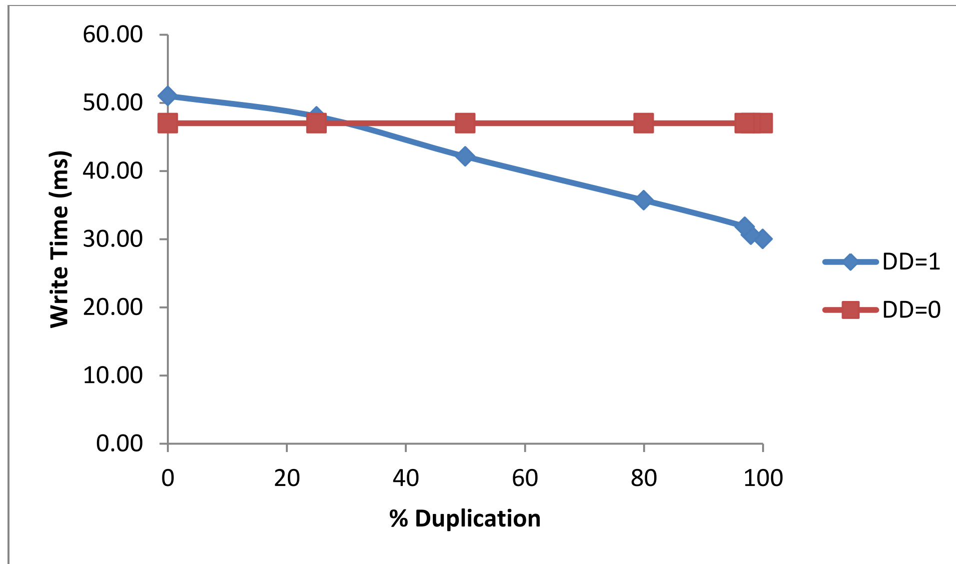

Figure 28: Effect of Data Deduplication on Write Time

6.5.2.2 Read Time

The read time with the new data duplication scheme increases for every read irrespective of the percentage duplication in the data. This is because the new read includes reading the header page first to find a page address and then read the actual page from the memory. Hence there is an overhead on every new read. This is clear from Figure 29 where blue line shows the read time for the new scheme for a sample testcase and red line shows the read time without the scheme for the same testcase. Thus from this curve we can conclude that all reads are penalized by certain amount (9.6% in this case) by introducing the data deduplication scheme.

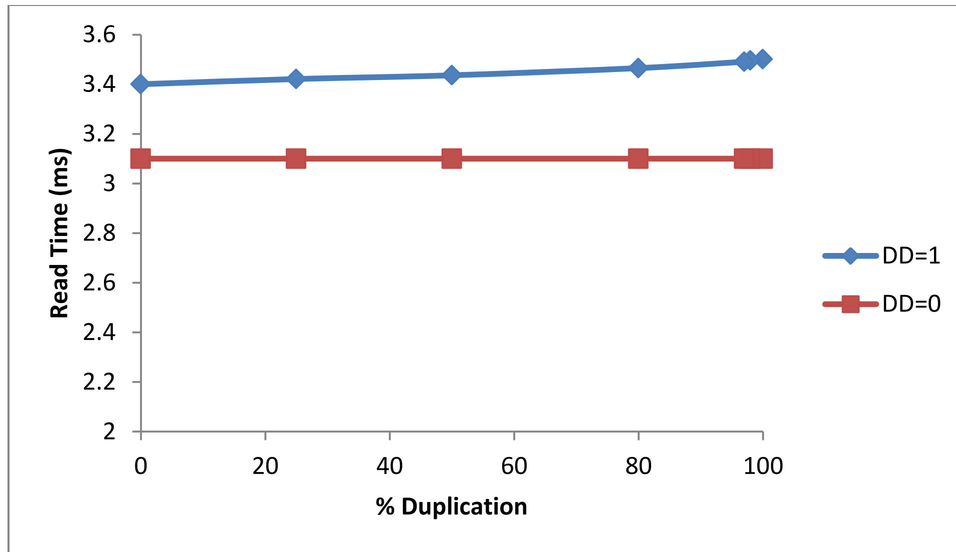

Figure 29: Effect of Data Duplication on Read Time

6.5.3 Data Deduplication with varying memory size

The storage gain is also dependent on the configuration of the memory other than the percentage of duplication. It will also depend on the configuration of the header page and the size of the signature. The storage gain will be dependent on the size of a page. This is because as the page size increases, the header page size will increase accordingly and there will be more bits available to store the shared page addresses. This however will increase the probability of aliasing since the page size is increasing and the signature size is still 1 byte. If the signature size is increased to keep the probability of aliasing minimum, the header page will occupy more signature bits and less shared page addresses. Hence the storage gain depends on the signature length and the increase in size of page.

Increasing block size means more number of pages in a block. If the header page has free bits that can be used to share pages, increasing block size can give extra storage gain. However if the header page is full and has no more capacity for extra shared page addresses, increasing block size will not give any additional storage gain.

CHAPTER 7

HARDWARE IMPLEMENTATION

One of the goals of this research is finding the area, power and timing estimates for the MISR-based design described in the previous chapters. The only way to find out these numbers is to do a Verilog implementation of the whole design and obtain the area and power estimates. This chapter gives a brief description of the Verilog implementation and the results using the Mentor Graphics and Synopsys Tools.

7.1 Verilog Implementation

7.1.1 Structure of Design

From the various techniques implemented in the previous chapters, we choose the MISR-based compression technique to implement in the hardware. A fixed memory configuration is also chosen since the size of memory can no longer be varied in hardware unlike we did in C++ implementation. Thus we select the 2Gb memory which is divided into 2048 blocks each blocks containing 64 pages and each page made up of 2048 bytes. The size of a signature is chosen to be one which requires one signature block to be present in the memory. The signature buffer is considered as 4-way set associative and hence contains 2048*4 locations each containing (8+6) bits. These 8 bits are used to store signature and remaining 6 bits are used to store the page address. Keeping this configuration as constant, a state machine is written for the different operations in a flash memory and different modules are coded.

7.1.2 Finite State Machine Description

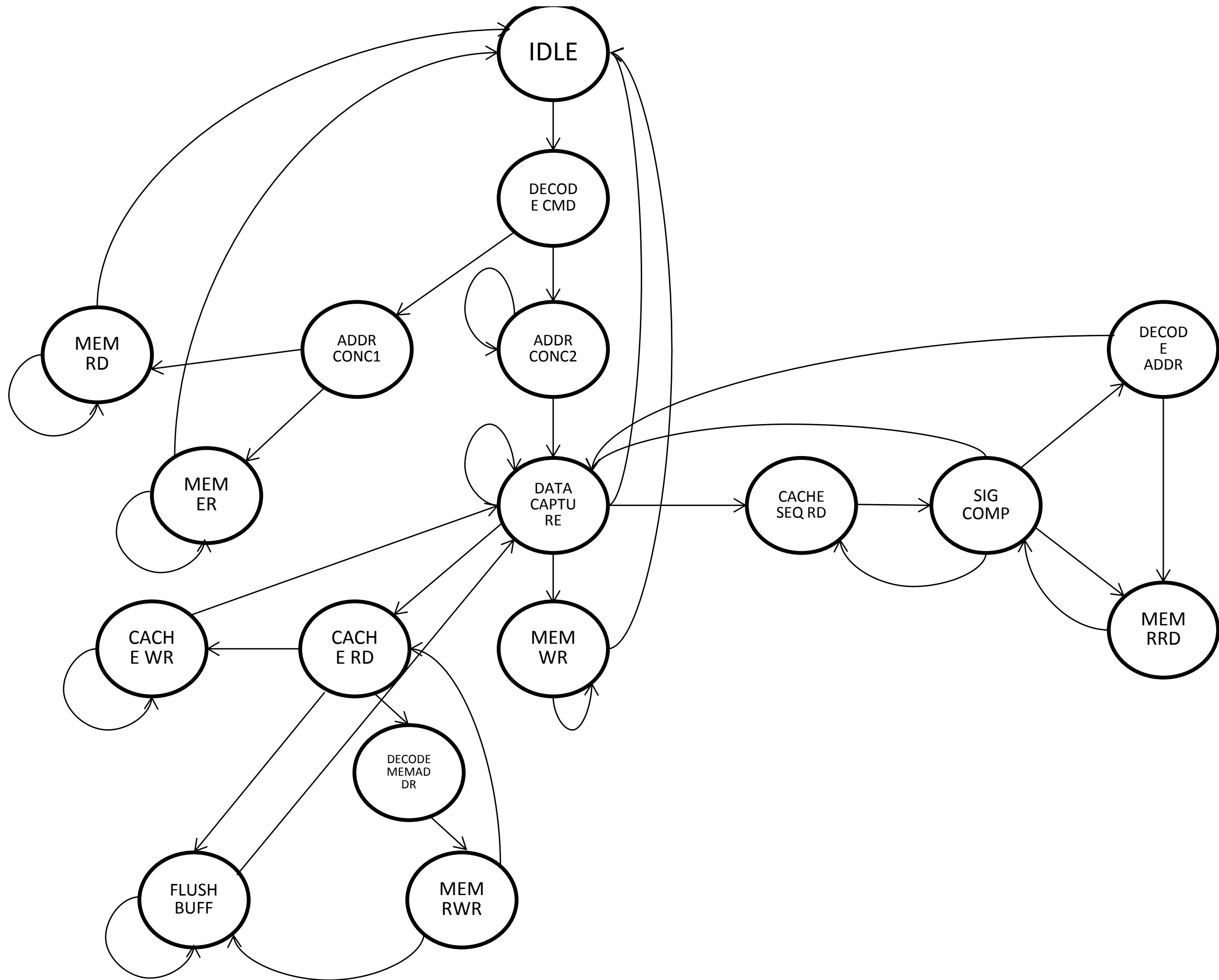

Figure 30: Finite State Machine for Flash Operations

Figure 30 shows the entire state machine for the different operations in a flash memory such as read, write, erase and search. The state machine can be better understood if it is related to the timing diagrams shown in Figure 5, 6 and 7. The signals that trigger the state changes are not shown in the figure to maintain readability of the state machine. The state machine gets triggered when the chip select of the memory goes low (ce_n is active low signal). After this when the command latch enable goes high, the state changes to DECODE CMD and decodes if the command is a read, write, erase or search. For read and erase, it goes to next state which is

ADDR_CONC1. If it is a read, MEM RD state is reached and the read operations start on the memory. This continues till the whole page is read and when it is done, state machine goes to IDLE state. Similarly for erase, it remains in state MEM ER till erase is performed on a block and then goes to IDLE state. Write and search however are more complex operations than these. For write/program or search operation, ADDR_CONC2 state executes and till the address is decoded it remains in this state. In the DATA_CAPTURE state, till write enable remains high, data is written into the register. At the same time this data is also given to the MISR state machine so that the signature of the data is calculated simultaneously. When data capturing is done and MISR state machine asserts a signal called misr_done, DATA_CAPTURE is done. Now if the operation is write, the data stored in the register starts getting written into the memory and at the same time data is written into the cache or signature buffer as we originally called it. The left bottom portion of the state machine shows this part of writing into the cache. If the cache locations are full, the contents of cache are written into the signature block using the random write command. Once the write back operation is done, the cache is flushed and the new signature is written into the appropriate location of the cache. Once this is done, the state machine is sent to the IDLE state. While doing this, it is also checked if the write on the memory was done successfully or not. The bottom right part of Figure 30 shows search operation. This operation also needs misr_done signal in order to trigger. In this operation, the content of the cache is first sequentially read and a match for the signature decoded is looked for. If a match is found, the address of the match is decoded and returned as output. Once the cache is completely read, the signature block is read for signature matches and similarly if matches are found, the address is decoded and returned. This makes sure that multiple matches are returned using the

scheme. Once the complete signature block is read, the state machine goes to IDLE state. In this manner all the flash memory operations are performed in Verilog.

We used Mentor Graphics Modelsim tool (version 6.6c) to compile and simulate the Verilog code. Simulation results were verified by writing testbenches for each memory operation and checking the expected behavior through modelsim waveforms. The results section will show the waveforms for different memory operations in Modelsim.

7.2 Results and Analysis

7.2.1 Simulation Results

The Verilog design was compiled and simulated using Mentor Graphics' simulation tool called Modelsim. The Modelsim Student PE Edition (6.6c) was used to perform the simulation analysis. The design was tested by writing various testbench files for every memory operation such as read, write, erase and search. This helped in verifying the flow of the finite state machine showed in Figure 30. Figure 31 shows the module level diagram of the Verilog design

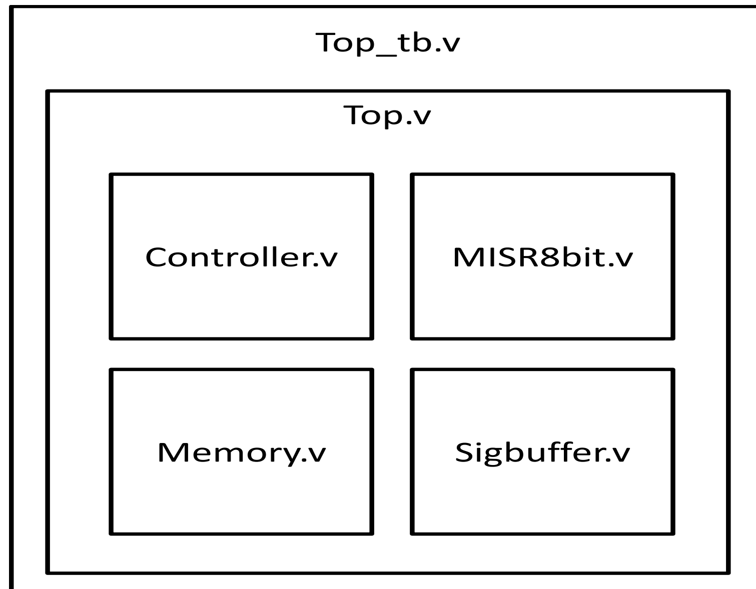

Figure 31: Module Level Diagram of the Flash Hardware Design

The state machine showed in Figure 30 is a partially a part of controller module. The left and right bottom parts of the state machine are implemented in the sigbuffer module. The MISR module takes a page of data as input and returns an 8 bit signature using a small state machine that is not shown in Figure 30. The memory module contains the instantiation of a 2Gb memory with blocks and pages and the result of different operations performed on the memory. Module level testbenches were implemented to verify the functionality of each module individually but are not shown in Figure 31. The top module contains instances of all the sub-modules and connections between these sub-modules. The top_tb.v is the top most testbench file which verifies the functionality of the entire Verilog design of the flash memory with MISR-based search scheme.

Figure 32 shows a snapshot of the *read* waveform showing what exactly happens when a read operation is triggered. At negdge of reset, all signals are reset to 0. After this CLE goes high for a cycle and the command 00h comes on io bus. Thus the read signal goes high in the next

cycle. After command is decoded, CLE goes low and ALE goes high for 5 cycles. In this time, the address appears on the io and is decoded and stored in the address register. Once the command and address are decoded, we goes low and re_n goes low (re_n is active low). This means output data should start appearing on the io bus. Thus the memory is read and output is seen on the io till re_n remains low. After a page is read, re_n goes low and so does mem_rd_done. This marks the end of read operation of a single page.

The new *write* waveform is shown in Figure 33. Similar to the read operation, write starts with resetting all signals, sending 80h (command cycle for write) followed by address on the io. After this the we still remains high since the data starts appearing on the io. This data is written into a data register which is 2048 bytes wide and then written into the memory at the specified page and block address. At the same time a signature of the data is calculated when misr_en goes high and this signature is written into the cache if the cache location is empty. If not, the contents of cache are written back into the signature block and cache is flushed so that new write can be performed. This part is not completely shown in the figure since it takes lot of cycles and is difficult to show in 1 waveform.

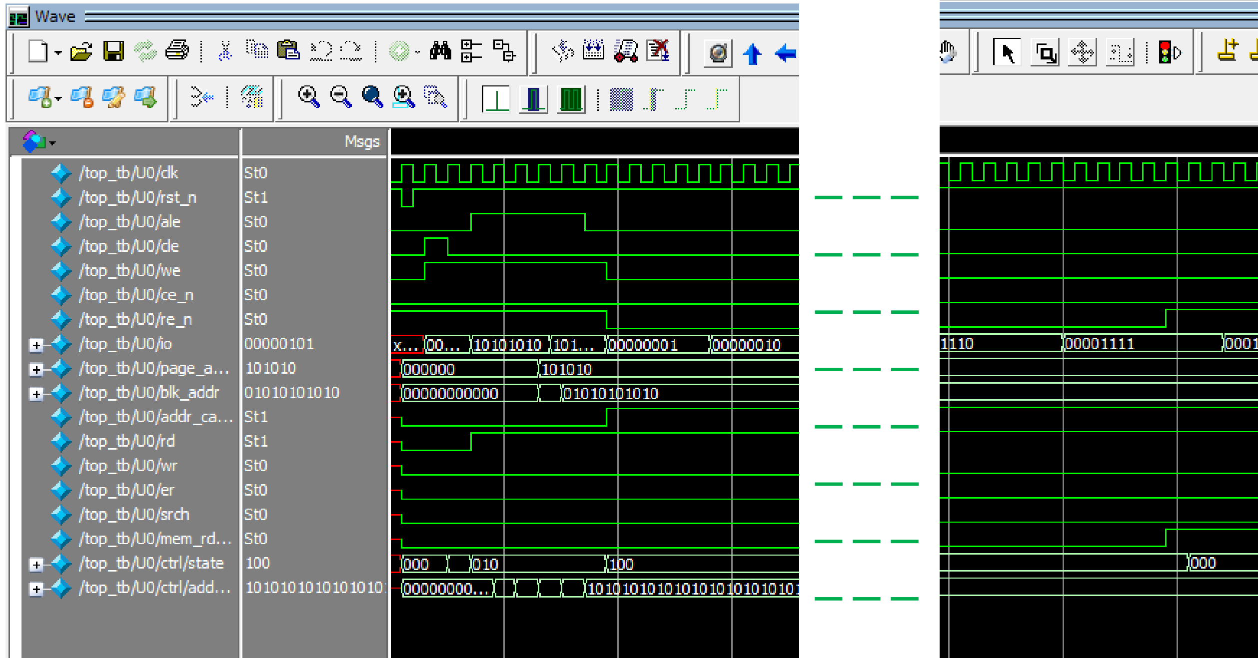

Figure 32: Modelsim Waveform for READ Operation

Figure 34 shows the waveform for search operation. The waveforms are generated in Modelsim by using the commands mentioned in [16]. It begins with resetting all signals and sending a search command (99h) on io bus followed by the address of the signature block. After this the data to be searched appears on the io and is converted into a signature by the MISR state machine. Once the signature is ready, cache_rd goes high and cache is read sequentially to match the signature. If signature matches, match goes high and matched address is decoded and returned on io. If not, the signature block is read after the cache read is finished. It is not possible to show the whole search operation in Figure 34. Waveform till cache read is shown. The search operation finishes when the signature block is completely read and mem_srch_done goes low.

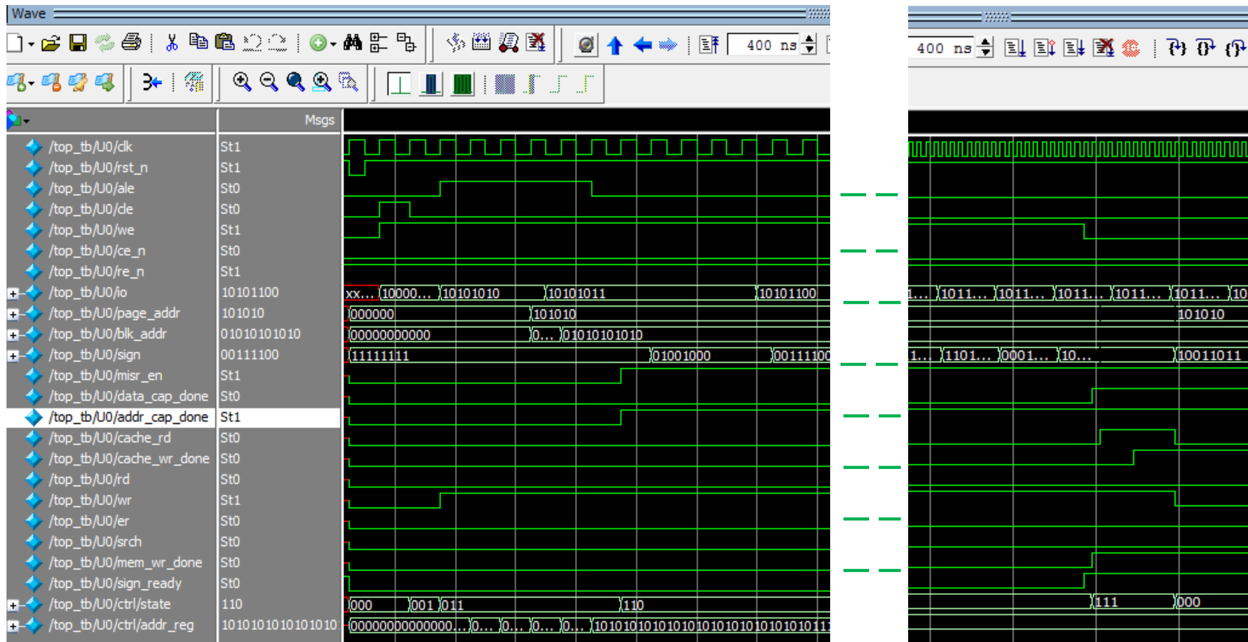

Figure 33: Modelsim Waveform for WRITE Operation

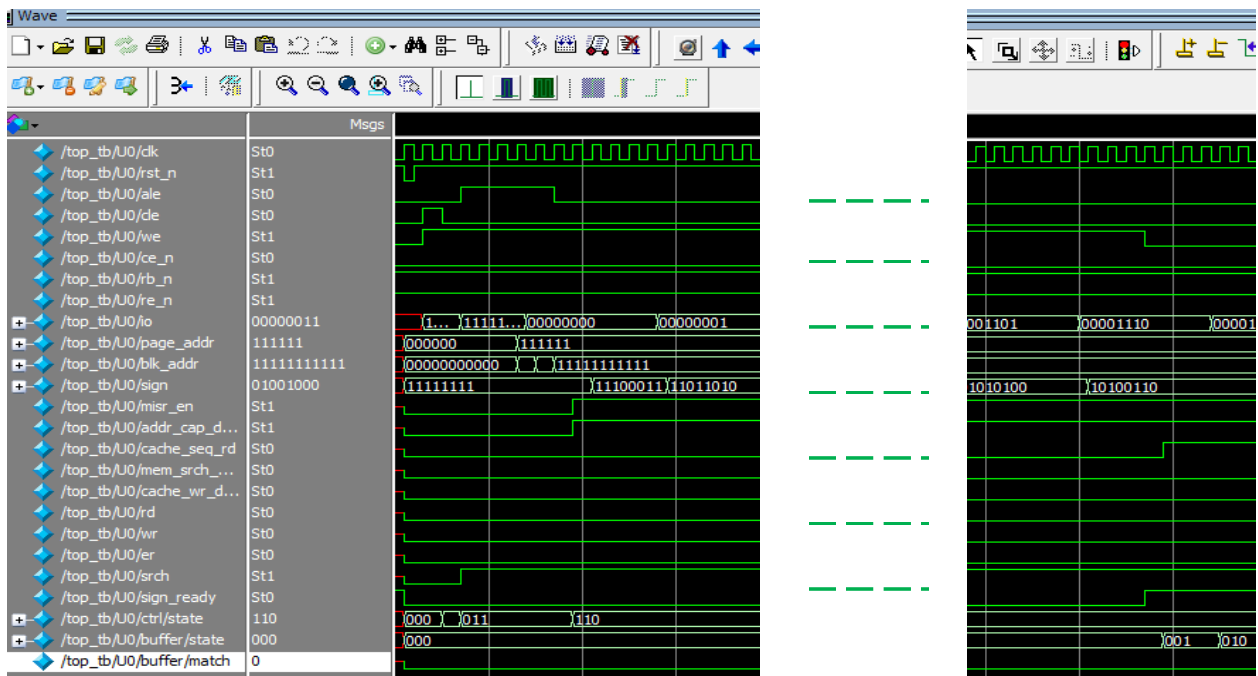

Figure 34: Modelsim Waveform for SEARCH Operation

7.2.2 Area Calculation using Design Compiler and CACTI5.3

In order to do area analysis, we used design compiler and a tool called CACTI. Design compiler is useful to find area of the logic. However it cannot handle a huge memory of 2Gb to calculate the area. Hence a tool called CACTI is used to find the area numbers of memory and the cache [17]. CACTI is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power calculation tool by HP Labs. We use the latest version of cacti called CACTI 5.3. Figure 35 shows the usage of CACTI for area calculation of the memory and Figure 36 shows the area numbers for the 2Gb memory which is 2312 mm^2. Similarly, area is calculated for the cache of size 16384 bytes as shown in Figure 37 and it comes out to be ~0.138 mm^2. Area for the control logic such as the state machine, MISR etc. is calculated using Design compiler. The result for this area compilation is shown in Figure 38.

CACTI 5.3

Normal Interface
Detailed Interface
Pure RAM Interface
FAQ

Parameter	Value
RAM Size (bytes)	268435456
Nr. of Banks	2048
Read/Write Ports	0
Read Ports	1
Write Ports	1
Single Ended Read Ports	0
Nr. of Bits Read Out	8
Technology Node (nm)	45
Temperature (300-400 K, steps of 10)	300
RAM cell/transistor type in data array (choose ITRS transistor for SRAM cell)	ITRS-HP / ITRS-LSTP / ITRS-LOP / LP-DRAM / COMM-DRAM (selected)
Peripheral and global circuitry transistor type in data array	ITRS-HP / ITRS-LSTP / ITRS-LOP (selected)
RAM cell/transistor type in tag array (choose ITRS transistor for SRAM cell)	ITRS-HP / ITRS-LSTP / ITRS-LOP / LP-DRAM / COMM-DRAM (selected)
Peripheral and global circuitry transistor type in tag array	ITRS-HP / ITRS-LSTP / ITRS-LOP (selected)
Interconnect projection type	Aggressive / Conservative (selected)
Type of wire outside mat	Semi-global / Global (selected)

Submit

Figure 35: Using CACTI 5.3 for Memory Area Calculation

CACTI is available for only SRAM and DRAM memories. To estimate area of a NAND flash memory, we choose the DRAM option in the above figure because a DRAM memory consists of

1 transistor per memory cell which is similar to a NAND flash memory. SRAM cell however consists of 6 transistors typically. Hence the DRAM approximation will be closest to NAND flash memory with respect to area.

RAM Parameters:

Number of banks:2048
Total RAM Size (bytes):268435456
Read/Write Ports per bank:1
Read Ports per bank:0
Write Ports per bank:0
Technology Size (nm):45
Vdd:0.7

Access time (ns): 8.25579105767
Random cycle time (ns):2.81797808064
Multisubbank interleave cycle time (of data array) (ns):3.59150446194
Total read dynamic energy per read port(nJ): 0.0596645917162
Total read dynamic power per read port at max freq (W): 0.0211728374064
Total standby leakage power per bank (W): 0.000236685878842
Refresh power (percentage of standby leakage power): 0.19808688971
Total area (mm^2): 2312.35097808
DRAM array refresh interval (microseconds):64000.0
DRAM array availability (percentage):99.9988728088
Best number of wordline segments: 4
Best number of bitline segments: 8
Best number of sets per wordline: 64.0
Best degree of bitline muxing: 1
Best degree of sense-amp level 1 muxing: 4
Best degree of sense-amp level 2 muxing: 16

Figure 36: Area, Power and Timing Numbers for 2Gb Memory

RAM Parameters:

Number of banks:1
Total RAM Size (bytes):16384
Read/Write Ports per bank:0
Read Ports per bank:1
Write Ports per bank:1
Technology Size (nm):45
Vdd:1.0

Access time (ns): 0.411490075713
Random cycle time (ns):0.212827136518
Multisubbank interleave cycle time (of data array) (ns):0.160147747399
Total read dynamic energy per read port(nJ): 0.00612224529665
Total read dynamic power per read port at max freq (W): 0.0287662813906
Total standby leakage power per bank (W): 0.0108133797834
Refresh power (percentage of standby leakage power): 0.0
Total area (mm^2): 0.137241696066
DRAM array refresh interval (microseconds):0.0
DRAM array availability (percentage):0.0
Best number of wordline segments: 2
Best number of bitline segments: 8
Best number of sets per wordline: 32.0
Best degree of bitline muxing: 1
Best degree of sense-amp level 1 muxing: 2
Best degree of sense-amp level 2 muxing: 32

Figure 37: Area, Power and Timing Numbers for Cache

```
shruti@vlsitest:~/flashVerilog/finalVfiles
Information: Updating design information... (UID-85)

****************************************
Report : area
Design : top_1
Version: B-2008.09
Date   : Mon Oct 18 16:56:22 2010
****************************************

Library(s) Used:

    NangateOpenCellLibrary (File: /home/shruti/flashVerilog/finalVfiles/NangateO
penCellLibrary_typical_conditional_ccs.db)

Number of ports:              14
Number of nets:            88592
Number of cells:           87676
Number of references:         45

Combinational area:      79703.707311
Noncombinational area:   100430.963732
Net Interconnect area:   33252.846670

Total cell area:         180134.671043
Total area:              213387.517714
1
~
```

Figure 38: Area Calculation for Logic using Design Compiler [18]

The aim of this experiment is to find the increase in area due to the introduction of the new hardware for the search scheme. If we consider that without this scheme the original area is just the area of memory, the original area would be 2312 mm^2. The new area would be as follows

New area = Area of memory + Area of Cache + Control Logic

= 2312mm2 + 0.138mm2 + 0.214mm2

= 2312.352

The percentage increase in area due to the search scheme can be written as

% Increase in Area = (New Area – Original Area)*100 / Original Area

= (2312.352-2312)*100 / 2312

% Increase in Area ~ 0.02%

This number will increase if the cache size is increased. The control logic size will mostly remain the same. Thus it shows that the % increase in area due to the new scheme is very less

and hence it is totally affordable to put this extra piece of hardware which gives such a big speed-up in the search operation of a NAND flash memory.

7.2.3 Power Estimation using Design Compiler

Power analysis can also be done similar to area analysis. We will find power numbers for memory and cache using CACTI and the control logic using design compiler. Figure 36 and 37 also show the power numbers for memory and cache. Figure 39 shows power analysis for control logic. The percentage increase in power can be calculated as follows

% Increase in Power = (New power – Original Power)*100 / Original Power

Original Power = Power (Memory) = Dynamic power + leakage Power per bank

= 0.02117W + 0.000237*2048

= 0.50655W

New Power = Power (Memory) + Power (Cache) + Power (Control Logic)

Power (Cache) = Dynamic Power + Leakage Power per bank

= 0.00725W + 0.00191*4W

= 0.01489W

Power (Control Logic) = Dynamic Power + Leakage Power

= 642.28 uW + 2.393 mW

= 0.0030353W

New Power = 0.50655 + 0.01489 + 0.0030353

= 0.524475W

% Increase in Power = (0.524475-0.50655)*100 / 0.50655

% Increase in Power = 3.53%

shruti@vlsitest:~/flashVerilog/finalVfiles

```
Report : power
        -analysis_effort low
Design : top_1
Version: B-2008.09
Date   : Mon Oct 18 16:56:42 2010
****************************************


Library(s) Used:

    NangateOpenCellLibrary (File: /home/shruti/flashVerilog/finalVfiles/NangateO
penCellLibrary_typical_conditional_ccs.db)


Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

Design        Wire Load Model           Library
------------------------------------------------
top_1                  1K_hvratio_1_4    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 624.3049 uW   (97%)
  Net Switching Power  =  17.9750 uW    (3%)
                         ---------
Total Dynamic Power    = 642.2798 uW  (100%)

Cell Leakage Power     =   2.3931 mW
```

Figure 39: Power Analysis for Logic using Design Compiler

7.2.4 Timing Analysis using Design Compiler

The timing analysis was done on the entire circuit including memory and cache unlike power and area analysis which were done partially in design compiler and partially using CACTI. The initial clock was selected as 10ns with 50% duty cycle. This resulted in a slack of 7.89ns shown in the DC timing report. Hence the clock was subsequently reduced till the slack was minimized and the timing was still met without any setup or hold time violations. This was done iteratively to arrive at the minimum clock for which the design can be safely run and this number was found to be **2.13ns**. Thus the maximum frequency of the design is 1/2.13ns = **0.47 GHz**. A snapshot of the timing report is shown in Figure 40.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : top_1
Version: B-2008.09
Date   : Mon Oct 18 20:57:03 2010
****************************************

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: ctrl/addr_reg_reg[22]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: buffer/next_match_reg
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  top_1              1K_hvratio_1_4        NangateOpenCellLibrary

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.00       0.00
  ctrl/addr_reg_reg[22]/CK (DFFR_X2)       0.00       0.00 r
  ctrl/addr_reg_reg[22]/Q (DFFR_X2)        0.11       0.11 r
  U23824/ZN (INV_X4)                       0.01       0.12 f

  buffer/next_match_reg/CK (DFFRS_X2)      0.00       2.13 r
  library setup time                      -0.05       2.08
  data required time                                  2.08
  -----------------------------------------------------------
  data required time                                  2.08
  data arrival time                                  -1.90
  -----------------------------------------------------------
  slack (MET)                                         0.18
```

Figure 40: Timing Report showing Clock and Slack Numbers

CHAPTER 8

CONCLUSIONS

NAND Flash memories are increasingly used in most consumer electronic goods these days to store any kind of data. As the size of a flash memory increases, searching any kind of data in the memory becomes slower. A novel compression-based search technique is implemented as a part of this research to minimize the search time in a NAND flash memory. The compression is achieved by two different techniques. The first technique described is a Multiple Input Shift Register based technique which compresses a page of data into a small signature which represents the entire page. The second compression technique is based on finding the discrete cosine transform of the data and storing the top few coefficients of the DCT as a signature. Each technique has its own advantages and disadvantages that are discussed and analyzed in this study. The DCT-based technique is mainly implemented for multimedia searches. Implementing these search techniques results in some penalty in the time taken to write data into the memory. Thus reduction in search time is achieved at the cost of increasing write time.

The compression techniques developed in this research have another application. These techniques can be used to store data effectively in a flash memory. This technique of compressing data, storing only the new data and storing pointers for existing data is called Data Deduplication and is widely used in the industry. It results in efficient data storage by storing more data in the memory without actually increasing the size of the memory. This technique also comes at the cost of increasing the read and write time of the memory.

The hardware-based techniques for enhanced search and efficient storage mentioned above are implemented in a C++ based simulator. The simulator is useful for varying different

parameters and analyzing the results. However it does not give a complete idea about the area, power and timing of these techniques which will be eventually implemented in hardware. Hence the hardware implementation of one of these techniques namely MISR-based search techniques is done in Verilog. Simulations are done using Modelsim tool by Mentor Graphics. Area, power and timing analysis is done using Synopsys Design Compiler and CACTI tool by HP Labs. This analysis shows that the new hardware will increase area by 0.02%, power by 3.53% and can operate at a maximum frequency of 0.47GHz.

Looking ahead, the DCT based search technique and the Data Deduplication methods can be implemented in hardware and area, power and timing analysis can be done. The data deduplication technique implemented here was the identity-based technique. Future work can include implementation of the similarity-based data deduplication technique for efficient data storage.

BIBLIOGRAPHY

[1] Jim Handy, “The Future of Flash”, in Proceedings of Flash Memory Summit, 2009.

[2] Micheloni, R.; Picca, M.; Amato, S.; Schwalm, H.; Scheppler, M.; Commodaro, S., “Non-Volatile Memories for Removable Media,” Proceedings of the IEEE, vol. 97, no: 1, 2009

[3] Zimmer, V.; Rothman M., “System, Method and Apparatus to Accelerate RAID Operations”, United States Patent US7594077, Sept 2009

[4] Jim Cooke, “NAND Flash 101: An Introduction to NAND Flash and How to Design It Into Your Next Product”, in Proceedings of Embedded Systems Conference, 2006

[5] S. Xiaoling, and W. Tutak, “Error identification and data retrieval in signature analysis based data compaction”; in Proceedings of IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, 1996

[6] Jianqin Zhou and Ping Chen, “Generalized Discrete Cosine Transform”, Pacific-Asia Conference on Circuits, Communications and Systems, 2009, PACCS’09

[7] Bu, L.; Chandy J., “A Keyword Match Processor Architecture using Content Addressable Memory”, in Proceedings of the 14th ACM Great Lakes symposium on VLSI, 2004

[8] J. Singaraju, and J. A. Chandy, “A Generic Lookup Cache Architecture for Network Processing Applications,” in Proceedings of International Symposium on FPGA, 2006

[9] R. Panigrahy and S. Sharma, Sorting and searching using ternary CAMs. IEEE Micro, 23(1):44–53, Jan/Feb 2003

[10] J. P. Wade and C. G. Sodini. A ternary content addressable search engine. IEEE Journal of Solid-State Circuits, 24(4):1003–1013, Aug. 1989

[11] V. Shruti, S. Aswin, K. Sandip **, “**TurboNFS: Turbo NAND Flash Search”, GLSVLSI (Great Lake Symposium On VLSI), May 2010

[12] V. Shruti, S. Aswin, K. Sandip, “DCT Based Scheme To Accelerate Multimedia Search in NAND Flash Memories”, ISOCC (International SoC Design Conference) November 2010

[13] Geer, D., “Reducing the Storage Burden via Data Deduplication”, IEEE Computer Society, 2008

[14] Dirk Meister, André Brinkmann, “Multi-Level Comparison of Data Deduplication in a Backup Scenario”, SYSTOR'09, May 4-6, Haifa, Israel

[15] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, Shmuel T. Klein, “The Design of a Similarity Based Deduplication System”, SYSTOR'09, May 4-6, Haifa, Israel

[16] Modelsim User's Manual by Mentor Graphics, Software Version 6.6c

[17] Thoziyoor S, Muralimanohar N., Ahn J.H. and Jouppi N.P., "CACTI 5.1", HP Laboratories, Palo Alto, 2008.

[18] Synopsys Design Compiler User Guide, Version 2002.05, June 2002