Enhancing Flash Lifetime in Secondary Storage

by

Chengjun Wang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 6, 2011

Keywords: Flash, Lifetime, Storage Systems

Copyright 2011 by Chengjun Wang

Approved by

Sanjeev Baskiyar, Chair
Associate Professor of Computer Science and Software Engineering
Hari Narayanan, Professor of Computer Science and Software Engineering
Cheryl Seals, Associate Professor of Computer Science and Software Engineering

Abstract

This research addresses the limited usable life of NAND flash-based storage systems. Unlike a magnetic disk drive, a NAND flash suffers from limited number of write cycles ranging from 10-100K depending on the specific type of flash. As flash memory densities increase and cell sizes shrink, further decrease in write endurance is expected. Although substantial research has been conducted to solve or mitigate the problem by wear leveling, write endurance remains a concern for write intensive applications.

We have proposed to use a DRAM cache to filter write traffic to flash memory. Intuition tells us that DRAM cache can filter writes to flash by coalescing and merging overwrites. However, the effectiveness of such a mechanism is not obvious considering that a large file system cache already exists which also merges overwrites. Since DRAM is volatile, to handle integrity of data upon power failure, we propose to use a supercapacitor backup which can provide short duration power during which the DRAM data can be retired to flash memory. The use of supercapacitors is superior to traditional use of battery-backed DRAMs as batteries, unlike supercapacitors, suffer from limited number of charge/discharge cycles as well as slow charge times.

We studied the effectiveness of DRAM cache in reducing write traffic to flash and the resulting response time and throughput changes. We investigated the use of a DRAM cache under two settings: a) when flash is used as a disk cache within a magnetic disk controller and b) when flash is used as a full secondary storage. Under the first setting, we used two levels of disk cache: DRAM disk cache and flash disk cache. Under the second setting, we used a single DRAM cache to filter the traffic to the full flash secondary storage. For both settings, we compared two policies to retire data from DRAM to flash memory: early vs. lazy retirement. In early retirement policy, flash is updated at the same time DRAM is

updated. In lazy retirement policy, flash is updated only upon data eviction from DRAM cache. Conventionally, early update policy has been used to avoid data loss upon power failure. With early update, write traffic to flash is not reduced via DRAM cache. In contrast, lazy update policy substantially reduces write traffic thereby extending the flash lifetime. Our simulation results show that using a medium-sized DRAM cache, flash lifetime doubles with lazy update policy compared to early update policy. Moreover, miss ratio and average response time decrease as well. With little effort, our technique can be extended to improve the usable life of other emerging non volatile memory systems, such as PCM and MRAM.

Acknowledgments

It is a pleasure to thank those who made this dissertation possible. I would never have been able to finish my dissertation without the guidance of my committee members, help from friends, and support from my family and my wife.

I would like to express my deepest gratitude to my advisor, Dr. Sanjeev Baskiyar, for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research during my study and research at Auburn University. His perpetual energy and enthusiasm in research had motivated all his advisees, including me. In addition, he was always accessible and willing to help his students with their research. As a result, research life became smooth and rewarding for me.

Dr. Hari Narayanan, Dr. Cheryl Seals, and Dr. Victor P. Nelson deserve special thanks as my advisory committee members and advisors for guiding my research and lending me encouragement. I attended Dr. Narayanan HCI course. I was moved by his rigor and passion on research. I asked Dr. Seals some questions when I took my Ph.D. qualifying exams. She was (still is and will be) very nice and willing to help her students. All of my classmates had the same impression. I would like to thank Dr. Nelson for willing to serve as a outside reader to assure the quality and validity. He spent so much time on reading and correcting my dissertation. His advice made this dissertation much better.

I would like to thank Dr. Prathima Agrawal for granting me Vodafone Fellowship for two years, which is a special honor and an encouragement for me.

I am grateful to Dr. David Umphress. He has been hiring me as Graduate Teaching Assistant since Fall of 2008, during which I have worked with Dr. Sanjeev Baskiyar, Dr. Kai Chang, Dr. Saad Biaz, Dr. Alvin Lim, Dr.Richard Chapman, and Dr. Daniela Marghitu. They are all helpful to my understanding of teaching.

I would like to extend my thanks and appreciation to Ms. Jacqueline Hundley, who served as my mentor for the summer course COMP1200 (MATLAB for Engineers) that I taught at Auburn University. It was her help that made my teaching easier and successful.

I would like to thank the faculty and staff of the CSSE department as they have guided me through this process. Special thanks to Ms. Barbara McCormack, Ms. JoAnn Lauraitis, Ms. Michelle Brown, and Ms. Kelly Price. They are all quick in response.

I would like to thank Dr. Xiao Qin for offering a course on "storage system", which helps me a lot in my research of storage systems. Furthermore, he sets a example for me on how productive a researcher can be.

I would like to thank my Master advisor Professor Yantai Shu and Professor Jianqing Pang when I was in Chinese Academy of Sciences. They inspired me to do research.

I would like to thank my fellow students Shaoen Wu, Qing Yang, Cong Liu, and Xiaojun Ruan. Shaoen was my office mate from whom I learned Latex, Cygwin, Microsoft Visio, and other tools. Qing was my roommate of the first year at Auburn. We spent the first year together. Cong was my classmate, with whom I took many courses together at Auburn and we spent some time to discuss research. Xiaojun has provided me with recently published papers related to my research and we had good time together to talk about DiskSim 4.0, the well regarded storage simulator.

I would like to thank Professor Emeritus Dr. Donald Street and his wife Dr. Mary Street. They kept on inviting my family to their family, which gave me opportunities to know more about American culture. I have learned a lot from them.

I would like to thank my colleagues Jianlin Nie, Tianshu Zheng, and Xin Yang. We co-founded our company in 1992 in China. I still received financial support after I left the company. It would not have been possible for me to pursue my Ph.D. in the US without them.

I would like to thank my wife, Jinhua Li. She has been encouraging me and standing by me through the good times and bad.

I would regret my doctoral years at Auburn University if I did not join Auburn Chinese Bible Study Group and Lakeview Baptist Church. It was not only a turning point in my life, but also a wonderful experience. I cherished the prayers and support between me and them, and the friendships with my Christian brothers and sisters. Special thanks to Dr. Kai Chang, Prof. Tin-man Lau, Mr. Paul Froede, Dr. Rob Martin, and Associate Paster Grady Smith. They have helped me grow spiritually. They have been a constant source of encouragement during my graduate study.

Above all, thanks be to God for giving me wisdom and guidance throughout my life. You were with me through all the tests in the past several years. You have made my life more bountiful. May your name be honored, may your kingdom come, may your will be done on earth as it is in heaven.

This dissertation is dedicated to my beloved wife Jinhua, my lovely daughter Helen and all my family.

Table of Contents

Abstract . . . ii

Acknowledgments . . . iv

List of Figures . . . xi

List of Tables . . . xv

1 Introduction . . . 1

1.1 Problem Statement . . . 2

1.1.1 Flash Memory . . . 2

1.1.2 Flash Trends . . . 3

1.1.3 Limitations of Existing Solutions . . . 5

1.2 Scope of the Research . . . 8

1.3 Contributions . . . 10

1.4 Dissertation Organization . . . 10

2 Literature Review . . . 11

2.1 Flash Basics . . . 11

2.2 Flash Disk Cache . . . 15

2.3 System Memory Model . . . 16

2.4 Solid State Drives and Reliability . . . 17

2.5 Reducing Write Traffic . . . 19

3 System Architecture . . . 20

4 Comparison of Traffic Mitigation Techniques . . . 23

4.1 Battery-backed DRAM as a write cache . . . 23

4.2 SLC as a write cache for MLC . . . 24

4.3 PCM as a write cache . . . 24

4.4 HDDs as a write cache 25
4.5 Summary 26
5 Methodology 28
5.1 Simulator 28
5.2 System Setup 28
5.3 Workloads and Traces 30
5.4 Performance Metrics 31
5.5 DiskSim 4.0 modifications 34
5.5.1 Trace formats: SRT1.6 and SPC 34
5.5.2 Secondary flash cache 36
5.5.3 Smart controller and MS SSD add-on 36
5.6 Validation 36
6 Traffic Savings for Flash as a Victim Disk Cache 39
6.1 Early Update vs. Lazy Update 39
6.2 Lazy Update Policy for Reads 42
6.3 Lazy Update Policy for Writes 42
6.4 The Benefits of Lazy Update Policy 42
6.5 Simulation Parameters 45
6.6 Performance Metrics 48
6.7 Experimental Results 48
6.7.1 Write Traffic Savings 48
6.7.2 DRAM Size Need Not to be Very Large to be Effective 55
6.7.3 Flash size has little effect on Write Traffic Savings 55
6.7.4 Miss Ratio Is improved Slightly 55
6.7.5 Response Time Is Improved Slightly 55
6.7.6 Summary 55
7 Traffic Savings for Flash as a Major Storage Medium 75

7.1 Early Update for reads 75
7.2 Early Update for writes 76
7.3 Lazy Update for reads 76
7.4 Lazy Update writes 79
7.5 Simulation Parameters 80
7.6 Performance Metrics 82
7.7 Experimental Results 82
7.7.1 Write Traffic Savings 82
7.7.2 DRAM Size Needs not to be Large to Be Effective 85
7.7.3 Response Time 85
7.7.4 Summary 85
8 Fault Tolerance Issue 93
8.1 What are Supercapacitors? 93
8.2 Why Supercapacitors not Batteries? 95
8.3 How to calculate the Capacitance of Supercapacitors? 95
9 Conclusions and Future Work 98
9.1 Main Contributions 98
9.2 Future Work 99
9.3 Conclusions 100
Bibliography 101
Appendices 107
A DiskSim 4.0 Flowcharts, Modifications, and Post Processing Scripts 108
A.1 Overview 108
A.2 System Diagram 108
A.3 DiskSim 4.0 flowcharts 110
A.3.1 Main loop 110
A.3.2 Message routing 110

A.3.3 HDD reads with dumb controller (type:1) 110
A.3.4 HDD writes with dumb controller (type:1) 110
A.3.5 HDD reads with smart controller (type:3) 116
A.3.6 HDD writes with smart controller (type:3) 116
A.3.7 SSD reads with dumb controller (type:1) 116
A.3.8 SSD writes with dumb controller (type:1) 121
A.3.9 SSD reads with smart controller (type:3) 121
A.3.10 SSD writes with smart controller (type:3) 121
A.4 Post Processing Scripts 121
A.4.1 fig_hplajw.sh source code 125
A.4.2 hplajw.sh source code 128
A.4.3 process.sh source code 129
A.4.4 miss_ratio.p source code 130
A.4.5 res_time.p source code 130

List of Figures

1.1 Lifetime of Flash 4

1.2 Memory taxonomy 6

1.3 Non volatile memory roadmap 7

1.4 Memory hierarchy of computers 9

2.1 Flash cell structure 12

2.2 Flash programmed state 12

2.3 Flash erased state 13

2.4 SLC vs. MLC 14

3.1 System architecture of Hybrid Hard Drives 21

3.2 System architecture of Solid State Drives 22

5.1 System architecture of modified DiskSim 4.0 37

6.1 Early update for reads 40

6.2 Early update for writes 41

6.3 Lazy update for reads 43

6.4 Lazy update for writes 44

6.5 Relative traffic for OpenMail 49

6.6 Relative traffic for OpenMail 50

6.7 Relative traffic for Synthetic workload 51

6.8 Relative traffic for Synthetic workload 52

6.9 Relative traffic for Websearch3 53

6.10 Relative traffic for Websearch3 54

6.11 Miss Ratio for OpenMail 56

6.12 Miss Ratio for OpenMail 57

6.13 Miss Ratio for Synthetic workload 58

6.14 Miss Ratio for Synthetic workload 59

6.15 Miss Ratio for Websearch3 60

6.16 Miss Ratio for Websearch3 61

6.17 Response Time for OpenMail 62

6.18 Response Time for OpenMail 63

6.19 Response Time for Synthetic workload 64

6.20 Response Time for Synthetic workload 65

6.21 Response Time for Websearch3 66

6.22 Response Time for Websearch3 67

6.23 Improvement of lazy update policy over early update policy for OpenMail . . . 68

6.24 Improvement of lazy update policy over early update policy for OpenMail . . . 69

6.25 Improvement of lazy update policy over early update policy for UMTR(websearch3) 70

6.26 Improvement of lazy update policy over early update policy for UMTR(websearch3) 71

6.27 Improvement of lazy update policy over early update policy for synthetic workload 72

6.28 Improvement of lazy update policy over early update policy for synthetic workload 73

7.1 Early update for reads 76

7.2 Early update for writes 77

7.3 Lazy update for reads 78

7.4 Lazy update for writes 79

7.5 Relative traffic for OpenMail 83

7.6 Relative traffic for Synthetic workloads 84

7.7 Relative traffic for Websearch3 86

7.8 Response time for OpenMail 87

7.9 Response time for Synthetic workloads 88

7.10 Response time for Websearch3 89

7.11 Improvement of lazy update policy over early update policy for OpenMail . . . 90

7.12 Improvement of lazy update policy over early update policy for UMTR(websearch3) 91

7.13 Improvement of lazy update policy over early update policy for synthetic workload 92

8.1 Flash as major store with a supercapacitor backup power 94

8.2 Flash as disk cache with a supercapacitor backup power 94

8.3 Supercapacitor and rechargeable batteries 96

A.1 System architecture of DiskSim 4.0 109

A.2 The main loop of DiskSim 4.0 111

A.3 Run simulation 112

A.4 io-internal-event 113

A.5 Message routing 114

A.6 Flowchart for HDD reads (controller type:1) 115

A.7 Flowchart for HDD writes (controller type:1) 117

A.8 Flowchart for HDD reads (controller type:3) 118

A.9 Flowchart for HDD writes (controller type:3) 119

A.10 Flowchart for SSD reads (controller type:1) 120

A.11 Flowchart for SSD writes (controller type:1) 122

A.12 Flowchart for SSD reads (controller type:3) 123

A.13 Flowchart for SSD writes (controller type:3) 124

A.14 Miss Ratio for Hplajw workload 126

A.15 Response Time for Hplajw workload 127

List of Tables

4.1 Comparison of traffic mitigation techniques 27

5.1 System environment 29

5.2 Workload characteristics 30

5.3 Validation of modified DiskSim 4.0 38

6.1 Disk parameters (QUANTUM QM39100TD-SW) 45

Chapter 1

Introduction

The performance gap between the processor and storage of computers is widening with approximately 60% and 10% annual improvement in the processor and hard disk drives (HDDs) respectively [1]. The trend is becoming more marked with the advent of multi-socket, multi-core processors architectures. The I/O performance, especially I/O operations per second (IOPS), has not caught up with the corresponding improvements in processor performance. The processor utilization stays low due to the need to wait for the data being fed from the storage system [2]. Therefore, storage performance is becoming the bottleneck of a computer system.

Fortunately, there are emerging memory technologies that try to bridge the growing performance gap, such as flash memory, Phase Change RAM (PCM), and Magnetic RAM (MRAM). Noticeable among them is flash memory, which has been the most widely used nonvolatile memory. There are two types of flash: NOR flash and NAND Flash. NOR flash is byte addressable while NAND flash is page addressable. In this paper, we are only concerned about NAND flash, which is referred to as flash for short hereafter. Not only has flash been widely used on portable devices as storage media, but also flash-based Solid State Drives (SSDs) are being installed into data centers.

SSDs can provide as much as 3,300 write IOPS and 35,000 read IOPS consuming 2.5 watts of power, whereas even the best HDDs (15K RPM drives) can only offer 300-400 IOPS while consuming 15-20 watts of power[2]. In comparison, the processor can offer 1,000,000 IOPS. Performance in term of IOPS is critical for enterprise applications serving a large number of users, like web servers, email servers, cloud computing, and cloud storage. The common method used to close the gap is to deploy multiple HDDs working in parallel to

support peak workloads. In order to meet the IOPS requirement, more HDDs are added, which results in environments that have underutilized storage (well below 50% of their useful storage capacity). The extra storage in turn incurs power and cooling waste.

However, flash can only be written a limited number of times, ranging from 10K to 100K depending on the type of flash used. Traditional solutions to limited lifetime of flash focused on the algorithms used within Flash Translation Layer (FTL), which spread the writes evenly across the medium. It is referred to as wear leveling, in which no single cell fails ahead of others. Wear leveling does nothing but uses up as much as the flash potential lifetime. Although wear leveling increase the lifetime to some extent, it remains a concern for write intensive applications.

The reminder of the chapter is organized as follows: Section 1.1 presents the problem statement. In Section 1.2, we talk about the scope of the research. In Section 1.3, we show our contributions briefly. Finally, Section 1.4 gives the organization of this dissertation.

1.1 Problem Statement

1.1.1 Flash Memory

As mentioned above, one of flash drawbacks is write endurance. The endurance issue stems from cell degradation caused by each burst of high voltage (10V) across the cell. This problem shows no imminent sign of vanishing for Multi-Level Cell (MLC) write endurance is worse compared to Single Level Cell (SLC). For example, the write cycles of 2X MLCs drop to 10,000 from 100,000 of that of SLC. Furthermore, write endurance becomes worse as cells become smaller.

In order to mitigate the such drawbacks, a Flash Translation Layer (FTL) has been proposed to manage how the flash resources are used [3]. FTL has mapping tables between logical and physical address spaces. When an update to a file is issued, FTL writes the file to a blank page and marks the old page as invalid. FTL updates the mapping tables accordingly. A technique called wear leveling attempts to evenly use all of the memory cells.

In spite of all these efforts, endurance is still a concern for flash based storage systems. The flash lifetime over I/Os per second is shown in Figure 1.1 [4]. The lifetime drops as the number of I/Os increases. As Kim et al. [4] put it, “Although MTTFs for HDDs tend to be of the order of several decades, recent analysis has established that other factors (such as replacement with next, faster generation) implies a much shorter actual lifetime and hence we assume a nominal lifetime of 5 years in the enterprise.” As we can see from the figure, when the number of I/Os exceeds approximately 50 IOPS, the lifetime of flash is less than 5 years. For example, the OpenMail workloads we used have 98 I/Os per second, which corresponds to a lifetime less than 2 years. Therefore, endurance is one of the factors that would hinder further applications of flash-based storage systems to a heavy write-intensive workload environment like some embedded or enterprise systems, where the storage system needs to work 24/7 with heavy write traffic. Therefore, enhancing the flash endurance is demanded.

1.1.2 Flash Trends

According to [5], flash trends can be expressed as: bigger, faster, and cheaper, but not better.

Bigger: Flash component densities are doubling at a rate greater than Moore’s Law. With a new type of 3D stacking technique, called TSOP stacking, the density can be doubled again. In addition, more dies can be stacked in some types of BGA packages. Currently, MLC technology makes it possible for SSD to have capacities of 512 GB to 1TB in 2.5-inch form factors.

Faster: A single operation is needed for programming an entire page or erasing an entire block. Depending on component density, currently page size varies from 2KB, 4KB to 8KB in 2009 while block size can range from 16KB to 512KB. The amount of time is independent of page size and block size. Therefore, as page size and block size are becoming bigger, the effective program/erasure speed becomes faster.

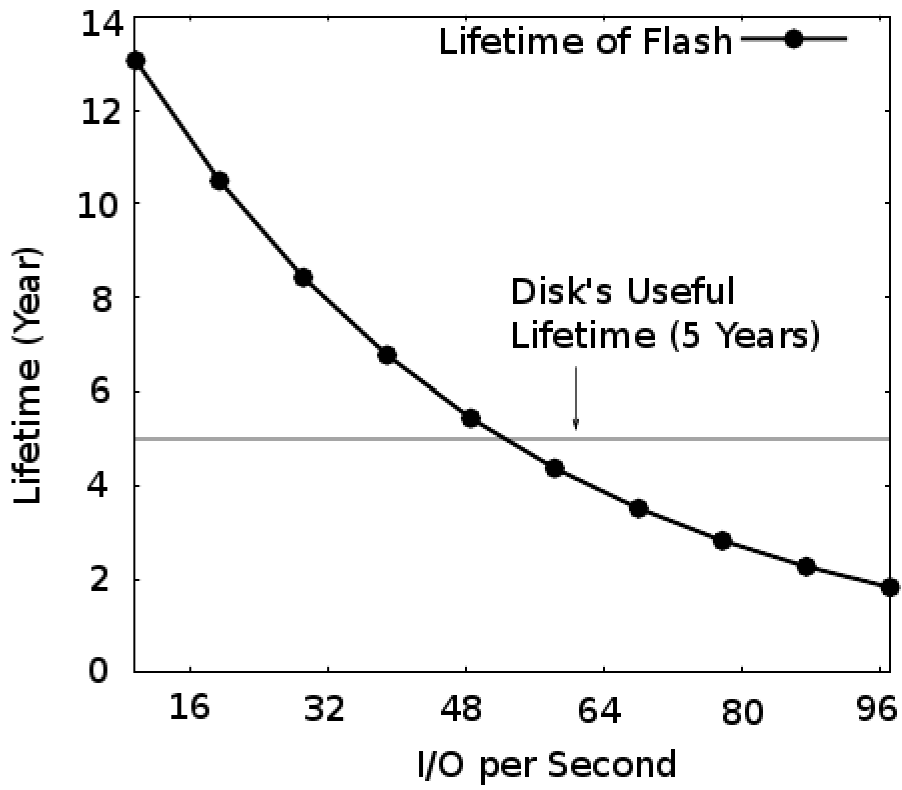


Figure 1.1: Lifetime of Flash

Cheaper: As the densities are going up, price is going down. For example, 2X MLC is roughly half the cost of the equivalent density SLC component. In 2004, 8GB SLC-based SSDs sold for about $4,000 while 40GB Intel X25-V sells for $129.99 in January of 2010– 5 times the capacity at about 1/30 price in about 6 years.

Not better: Better means more reliable in terms of endurance and data retention. Data retention is defined as the length of time a charge remains on the floating gate after the last program. The cells are worn out over program/erasure and make it more difficult to keep the electrons in place. Therefore, there is an inverse relationship between endurance and data retention – the more program/erasure, the shorter the data retention. As NAND manufactures are struggling for lower cost per bit, they keep sacrificing endurance and data retention. The most renowned trade-off is in MLC vs. SLC, in which 2:1 or 3:1 cost benefit is obtained through 10:1 reduction in rated endurance.

A classification of memory technologies is given in Figure 1.2 [6]. Flash has been classified as "mature nonvolatile memory". Most importantly, flash is likely not to disappear in the near future, which can be seen in Figure 1.3 [7]. Although there are emerging memories, such as PCM (PCRAM) and MRAM, they have not reached the maturity level to replace flash. Flash will coexist with the emerging memories for 5-10 years. Therefore, extending the flash lifetime needs to be explored.

1.1.3 Limitations of Existing Solutions

There are two basic methods to solve or mitigate the short flash lifetime 1) efficiently use cells, 2) reduce the write traffic to flash. Most research uses the first method, e.g., wear leveling. Since flash lifetime is directly related to cycles of program/erase, reducing write traffic to flash will extend the flash lifetime. However, little research has employed the second method. We did find Soundararajan et al.'s paper [8] using the second method. However, they use disk-based write cache instead of DRAM cache to save write traffic. RAM buffer has been proposed to improve the flash performance. Park et al. [9] reduce

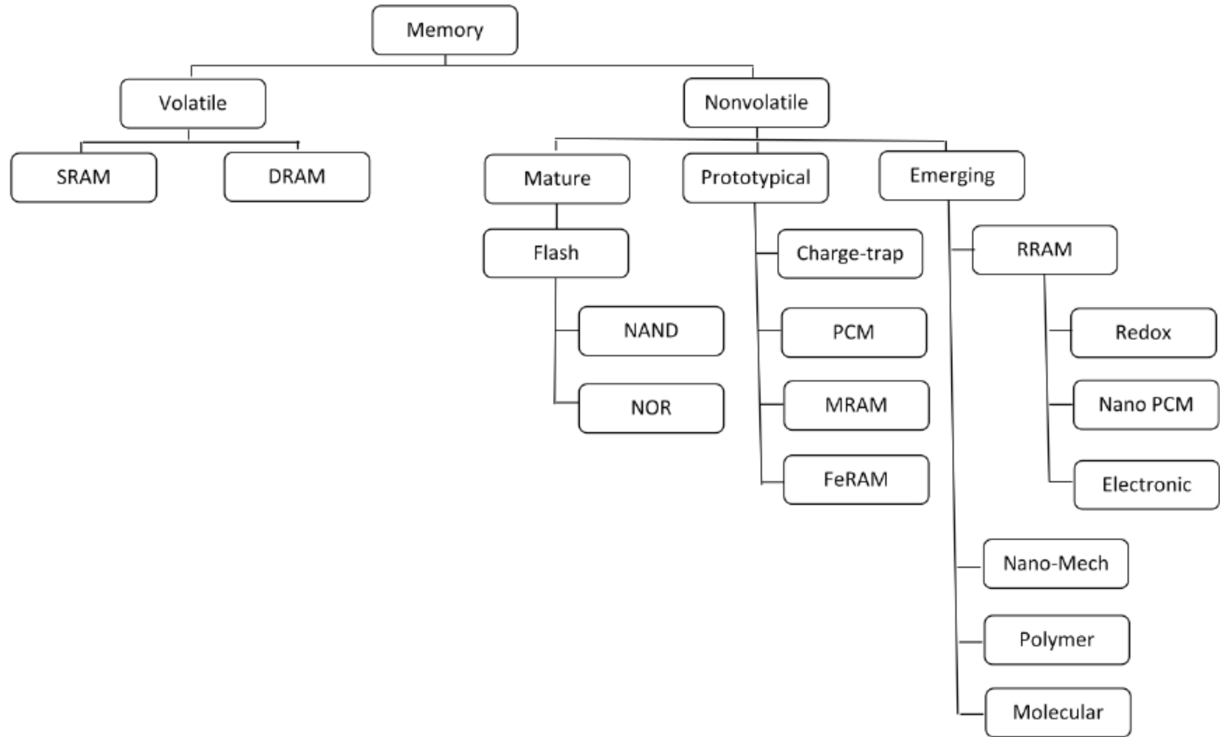


Figure 1.2: Memory taxonomy

writes by evicting clean data first. Jo et al. [10] propose a flash aware buffer management scheme to decrease the number of erase operations by choosing victims based upon page utilization. Kim et al. [11] use three key techniques, block-level LRU, page padding, and LRU compensation to improve random write performance. Gupta et al. [12] enhance random write performance by selectively caching page-level address mappings. However, the above studies except Soundararajan's work did not concentrate on the lifetime issue although Kim et al. mentioned the erase counts in their papers. Next, a common problem of using DRAM is the fault tolerance issue inherent in the volatile memory. Although Kim et al. [11] suggested using a small battery or capacitor to delay shutdown until the RAM content is backed up to flash, both batteries and capacitors have their limitations. Batteries have short lifetime, which require maintenance and replacements during the lifetime of devices while capacitors of a reasonable size do not have enough energy sustaining enough time, during which the

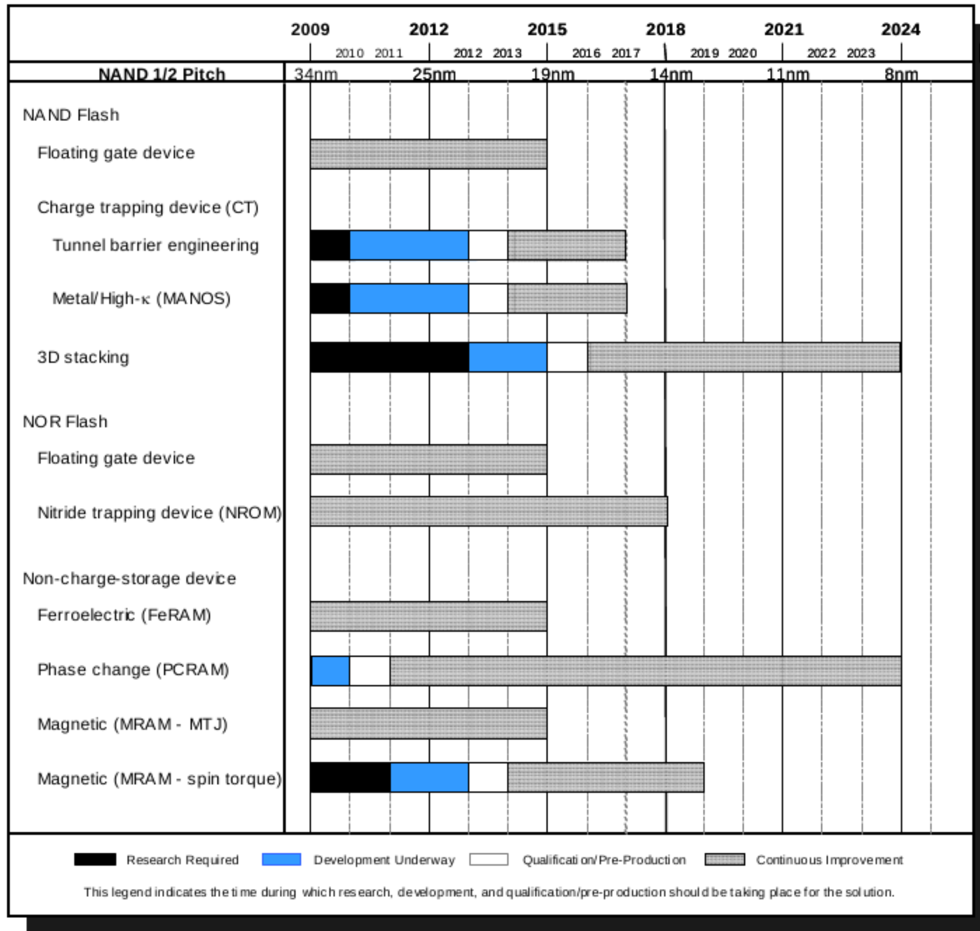


Figure 1.3: Non volatile memory roadmap

data is transferred to flash. Finally, flash as a disk cache has not been studied in their research.

1.2 Scope of the Research

In this dissertation, we focus on extending the flash lifetime, where flash is used at the device level shown in 1.4.

We propose to use flash as a victim device of a DRAM disk cache with a supercapacitor backup power to prolong its lifetime. Unlike traditional solutions, our method uses the second method listed above to decrease the traffic to flash.

Intuition tells us that DRAM disk cache can reduce writes by coalescing and merging multiple writes into a single write. **However, on second thought, we doubt that it would be effective given that there is a larger file system cache in main memory above the device level which already merges writes.** For example, typical DRAM disk cache size in HDDs is 16MB/32MB whereas file system cache can be 512MB or all free memory for a PC with 4GB main memory. Thus, it is far from clear whether such a small DRAM disk cache at device level could indeed reduce traffic to persistent storage.

Moreover, researchers are reluctant to use DRAM in the disk-cache for this purpose because [8] DRAM is volatile memory and thus could cause loss of data upon power failure. Yet, whether DRAM is suitable for reducing traffic depends on the following factors:

- How much write traffic would be reduced by using DRAM cache?
- What is the minimal size of DRAM cache to be effective as a traffic saver?
- How would flash size impact the traffic savings?
- What sort of update policy should be used?

In this dissertation, we attempt to answer these questions.

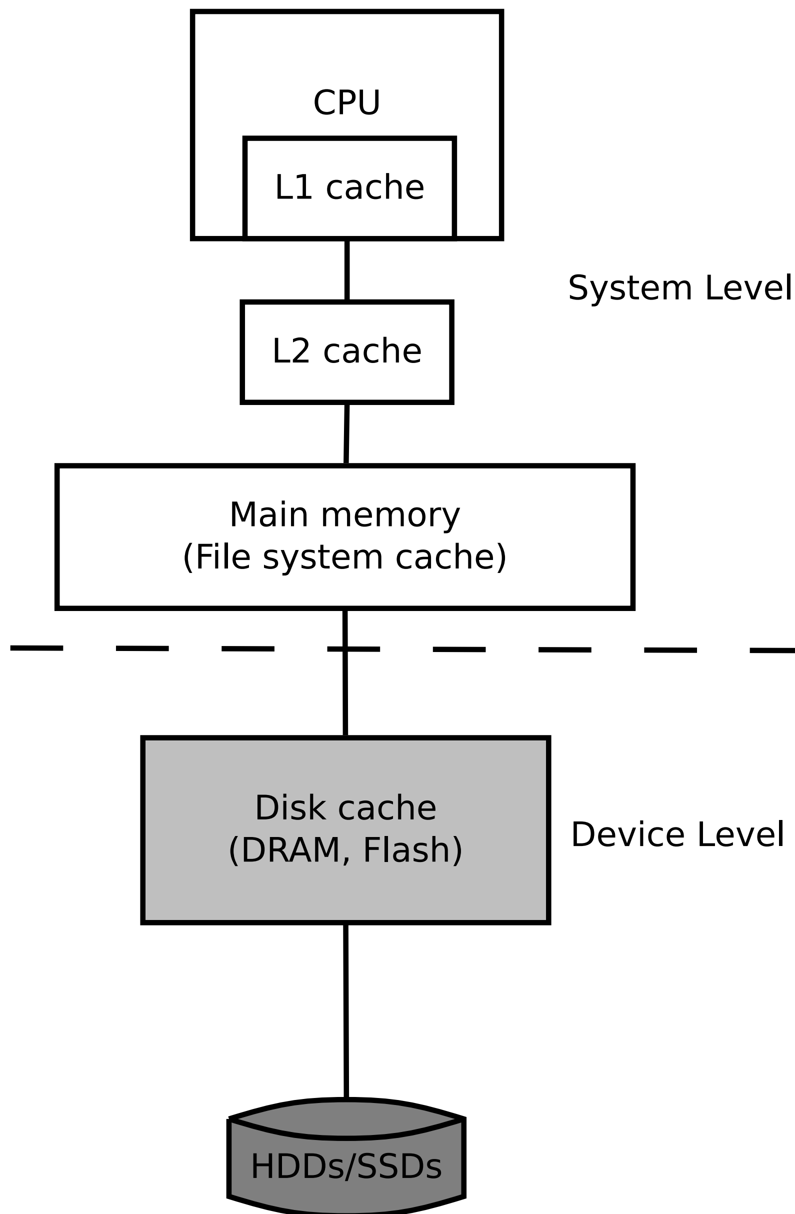


Figure 1.4: Memory hierarchy of computers

1.3 Contributions

A storage system is an indispensable part of a computer system. The success of this dissertation will widen the horizon of using flash in the storage systems, where the endurance issue hinders its wider applications. It will narrow the widening performance gap between the processor and the storage systems, which has plagued the computer system for several decades. It also prepares the storage systems for transition to new kinds of memory technologies. Surely, it will have a big impact on almost all areas, which use computers as a tool since the storage system is a fundamental part of a computer system. More detailed contributions are presented in Chapter 9.1.

1.4 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents literature review. Chapter 3 shows the system architecture. In Chapter 4, we compare three traffic saving methods. In Chapter 5, our research methodology is discussed. Chapter 6 presents our experimental results when flash acts as a victim disk cache. In Chapter 7, we show our experimental results when flash is used as a primary store. In Chapter 8, the fault tolerance issue is discussed. In Chapter 9, we conclude this dissertation and talk about the future work.

Chapter 2

Literature Review

In this chapter, we briefly present previous literature related to flash fundamentals and research on flash endurance that is most relevant to our research.

The remainder of the chapter is organized as follows: Section 2.1 presents the fundamentals of flash physics. In Section 2.2, we show the work related to flash acting as disk cache. In Section 2.3, we talk about the work related to flash being used as part of system memory. In Section 2.4, we show the work related to SSDs and reliability. Finally, Section 2.5 shows the work related to using write traffic savings to extend the lifetime of flash.

2.1 Flash Basics

The fundamentals of flash physics help us understand the properties of flash. The flash cell structure is shown in Figure 2.1 [5]. Each cell is made up of one of these transistors. In a SLC device, one of these transistors can hold 1-bit of data while holding multiple bits for a MLC. Data is written to the cell by electron tunneling; a high enough voltage is applied to the gate, creating a powerful electric field such that electrons will tunnel through the oxide and into the floating gate as shown in Figure 2.2. The electrons remain in the floating gate after the voltage is removed. Apply the voltage across the channel instead of the gate, reversing the bias, and the electrons will go in the other direction as shown in Figure 2.3. We have two states, 0 and 1, and the state is unchanged even if the cell has no power, making it ideal for a storage device.

There are two types of flash memory: NOR and NAND [13]. They were invented to replace EPROMs and EEPROMs. Both types can be read and written. Like their predecessors, a non-blank flash cell must be erased before further writing. A single cell

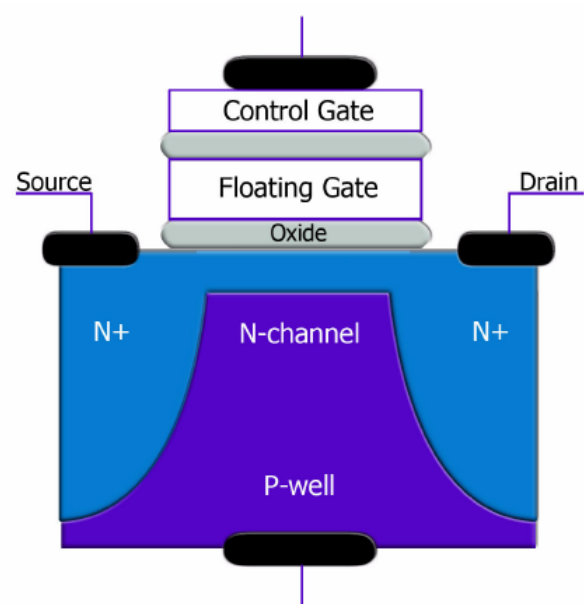


Figure 2.1: Flash cell structure

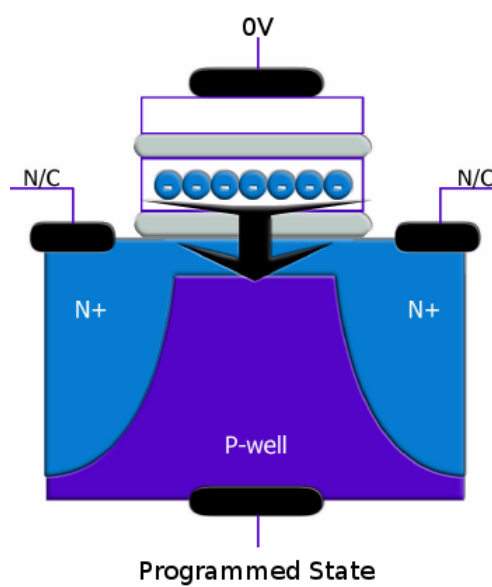


Figure 2.2: Flash programmed state

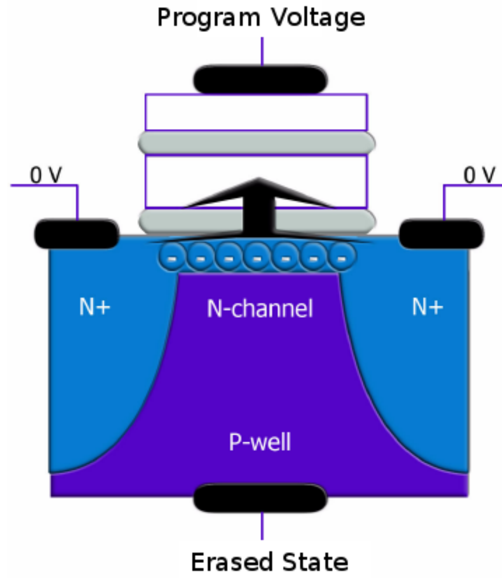
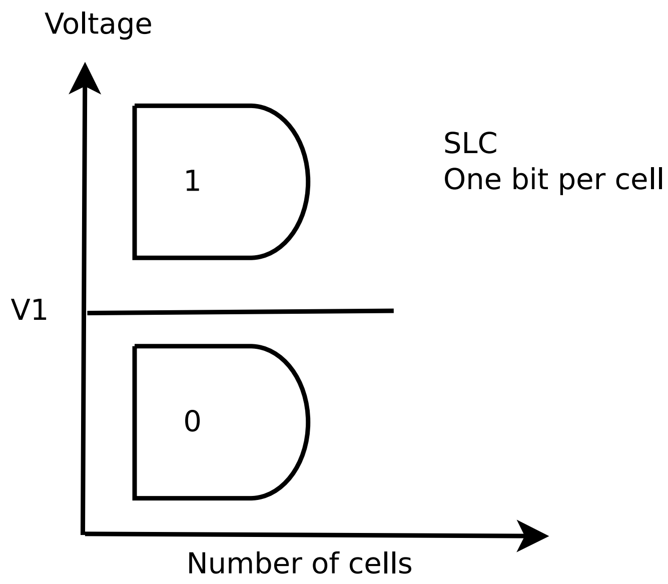


Figure 2.3: Flash erased state

cannot be erased; only a block of cells can be erased together. NOR flash is byte accessible, meaning that an individual byte can be read or written. This characteristic is suitable for holding code. Thus, NOR flash was quickly adopted as a replacement for EPROMs and EEPROMs. Unlike NOR flash, NAND flash cannot be byte-accessible but is page-accessible, which means that only pages of data can be streamed in or out of the NAND flash. This change makes NAND flash denser than NOR flash. The standard NAND cell is 2.5 times smaller. Therefore, NAND flash is much cheaper than NOR flash and is more suitable for mass storage. In this proposal, we are concerned about NAND flash. The words flash and NAND flash will be used interchangeably to denote NAND flash.

Flash has been developed into two forms: single-level cell (SLC) and multilevel cell (MLC) [14] as shown in Figure 2.4. In SLC technology, each cell represents one bit while in MLC, each cell can represent more than one bit. 2-bit MLC has been commercially used. 3-bit and 4-bit MLC are on the manufacturers' roadmaps. SLC needs one reference voltage (V1) to distinguish 0 from 1. In a 2-bit MLC, three threshold voltages (V1, V2, and V3) are needed. MLCs with more bits tend to be more error prone because the more the bits per cell the smaller the voltage margins.



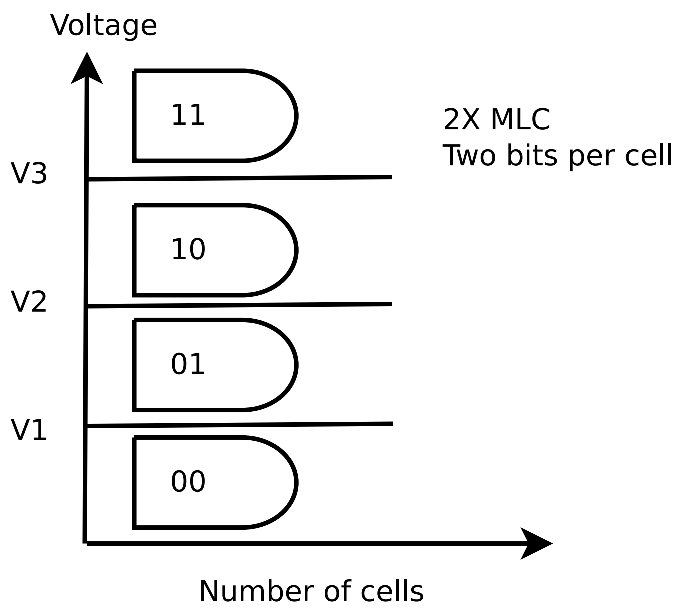


Figure 2.4: SLC vs. MLC

There are three basic operations: reading, erasing, and programming [15]. The read operation is performed by applying to the cell a gate voltage that is between the values of the thresholds of the erased and programmed cells and sensing the current flowing through the device. The write operation involves three main operations: read, erase, and program. Erasing sets all of the bits to 1. Programming only occurs on pages that are erased since program operation can only change a bit from 1 to 0. The erasure operation requires a high voltage pulse to be applied to the source when the control gate is grounded and the drain floating. Programming is an iterative process. The controller will apply voltage to the gate (or the channel),allow some electrons to tunnel and check the threshold voltage of the cell. When the threshold voltage has reached some predetermined value, the data is stored.

NAND flash consists of blocks, which are composed of pages. A block is the smallest erasable unit of storage. Erasing a block is an expensive operation, taking about 1.5-2 ms in a SLC device, and about 3 ms in a MLC device. Each block consists of a set of addressable pages, where SLC usually has 64 pages per block and MLC has 128 pages per block. A page is the smallest programmable unit. Each SLC page is usually 2 KB in size, and a 2X MLC page is 4 KB in size.

2.2 Flash Disk Cache

One usage model of flash is for flash to act as disk cache.

Hsieh et al. [16] proposed to use flash as disk cache for saving energy and improving performance. They did not study the relationship between the flash disk cache and the upper layer DRAM disk cache. They focused on a lookup mechanism. They compared several lookup mechanisms in terms of energy efficiency and product lifetime. However, they cited 1M as endurance cycles (10^6) to estimate the product lifetime. The current high density chip has fewer endurance cycles (10^5). The MLC NAND Flash has even fewer endurance cycles, typically being rated at about 5K-10K cycles. Therefore, the estimated lifetime by Hsieh et al. is no longer accurate with regard to the current technology. In addition, they

assumed that all the reads and writes are inserted into the flash disk cache. However, some data, such as stream data, are unlikely to be reused frequently. Allowing such data to go through the cache will waste the lifetime of flash.

Bisson et al. [17] presented four techniques to improve system performance using the characteristics of hybrid disks. They made use of the NVCache provided in hybrid disks to keep the spin down time longer. They divided the NVCache into two: one for writes and one for reads.

Another work that Bisson et al. [18] [19] have done was to use flash as a backup write buffer, in which the data stored in flash is used only in case of power failure. Flash acts as mirror storage to a DRAM write buffer. The limitation of this method is that endurance would suffer for a small size of flash. It will not be suitable for heavy write workloads.

Microsoft ReadyDrive [20] makes use of hybrid hard disk drives. Microsoft ReadyDrive is used to control the pinned set. The flash is mainly used to store boot or program launch related data. It is also used as a non-volatile write buffer while the hard disk drives are in sleep mode. However, the details of how ReadyDrive controls flash remain a commercial secret.

Kim et al. [21] proposed using a PCM and NAND flash hybrid architecture for embedded storage systems. They used PCM to overcome the random and small write issue of flash. Their findings are beneficial for our research in the general mass storage system with PCM and flash.

Joo et al. [22] take advantage of pinned-set flash in a hybrid drive to hold application code. They demonstrate how to decrease the application launch times with only a small part of applications stored into pinned set of flash memory. By pinning 5% and 10% of the application launch sequences, launch times are improved by 15% and 24% respectively.

2.3 System Memory Model

Another usage model of flash is to present flash as part of the system memory.

Microsoft ReadyBoost belongs to this category. It can use nonvolatile flash memory devices, such as universal serial bus (USB) flash drives, to improve performance. The flash memory device serves as an additional memory cache. Windows ReadyBoost relies on the intelligent memory management of Windows SuperFetch. Intel TurboMemory [23] can be used as ReadyDrive and ReadyBoost.

Kgil et al. [24] [25] [26] also proposed flash cache at the main memory level. They also split flash cache into a write region and read region. But such splitting would decrease the lifetime of flash memory because small portions of flash are overused while a large unified flash would mitigate the flash write endurance issues by wear-leveling across a large range.

2.4 Solid State Drives and Reliability

The third usage model of flash is to use flash as the main storage medium in solid state drives.

A solid state drive (SSD) is a storage device that uses solid-state memory. A SSD is commonly composed of flash memory along with DRAM [27].

Several techniques, such as wear leveling, error detection and correction, and block management algorithms, have been employed to extend the life of a SSD at the drive or system level.

Wear Leveling: it is a technique for extending the service life of some kinds of erasable computer storage media, such as EEPROM and flash [28]. Wear leveling attempts to work around these limitations by arranging data so that erasures and re-writes are distributed evenly across the medium. Therefore, it prevents some portions of flash cells from being overused so no single cell fails ahead of others. Wear leveling is implemented in the Flash Translation Layer (FTL). It can also be implemented in software by special-purpose file systems such as JFFS2 [29], YAFFS [30], and ZFS [31].

FTL falls into three categories based on the mapping granularity [32].

- Page-level FTL

- Block-level FTL

- Hybrid FTL with mixture of page and block mapping

In page-level FTL, page is the mapping unit. The logical page number is used to map the requests from upper layers such as the file system to any page of the flash. The advantage of this scheme is the high utilization efficiency. However, the size of the mapping table is big. For example, 32MB of space is needed to store the page-level mapping table for a 16GB flash memory.

On the other hand, block-level FTL uses the logical block number to translate requests between flash and the file system. Block-level FTL is better than page-level FTL in terms of mapping table size but worse in terms of utilization efficiency. The size of the mapping table is reduced by a factor of block size/page size (e.g., 128KB/2KB=64) as compared to page-level FTL.

Hybrid FTL has been proposed to take advantage of both page-level FTL and block-level FTL. In Hybrid FTL, flash blocks are divided into two groups: data blocks and log blocks. Data blocks use block-level FTL while log blocks use page-level FTL.

According to Chang et al. [33] [34] [35], there are two kinds of wear leveling techniques: dynamic wear leveling (DWL) and static wear leveling (SWL) [36] [33]. In DWL, only updated data (hot data) is involved in wear leveling while unchanged data remain in place. In contrast, the SWL technique proactively moves the static data so that all data has a role in wear leveling. According to the authors, SWL achieves much better write endurance than DWL.

Besides, there is much research [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] dealing with write endurance through wear leveling. Although the detailed techniques they use may differ from each other, their goals remain the same: evenly use the memory cells.

Error Detection and Correction: Like DRAM, Flash is not error free memory. Even worse, flash wears over writes and exhibits more errors as the number of program/erasure increases. Therefore, error detection and correction should be applied to flash memory to

guarantee correct data. An error-correcting code (ECC) or parity is used to detect and correct errors. There are trade-offs between the number of bits corrected, cost and complexity.

Storage Management Algorithms: To enhance endurance, the SSD controller manages bad blocks and spare blocks. Bad blocks are recorded in a table. Spare blocks are used to replace bad blocks. In general, a SSD is equipped with spare blocks that are 1 to 2 % of the capacity. More (e.g., 50%) can be expected if more reliability is needed.

2.5 Reducing Write Traffic

Usually, a non-volatile write cache or log region is employed in front of the flash medium to catch the write traffic so that less write traffic would reach the flash medium. Some works [21] [47] use the emerging memory technologies (e.g., PCM) as the non-volatile log region. Unlike flash, PCM supports in-place updating. In addition, PCM is faster than flash. Due to the front end PCM log region, traffic to the flash is decreased.

Soundararajan et al. [8] even use disk-based write caches to extend SSD lifetimes. Their design is based on the following observations. First, there are many overwrites of a small set of popular blocks at the block device level. Second, thanks to the file system cache, there will be not many immediate reads following a write. Finally, HDDs are excellent at sequential writes and reads. Therefore, HDDs fit log-structured write cache perfectly.

Qureshi et al. [48] have proposed using a DRAM buffer to lazy-write to PCM in order to improve write performance as well as lifetime of PCM. However, their target is at the system level rather than at the device level. Additionally, they did not deal with the power failure issue. Therefore, our research is complementary to theirs.

Using DRAM cache with supercapacitor backup power to save write traffic does not need many extra efforts since DRAM is an indispensable part in almost all flash controllers. A little larger DRAM might be needed to be used as a traffic saver.

Chapter 3

System Architecture

There are three major usage models of flash memory:

1. System memory model

2. Disk caches

3. Storage devices (SSDs)

Since our focus is on secondary storage, the system memory model is beyond the scope of this dissertation. Therefore, we deal with two usage models: disk caches and storage devices. For both categories, we propose to use flash as a victim device of the DRAM cache. The first category is shown in Figure 3.1. There are two levels of disk caches: the primary DRAM cache and the secondary flash cache. The flash cache is only updated upon data being evicted from the DRAM cache. The second category is shown in Figure 3.2. There is only one level of disk cache. Like the first category, flash is updated only when the data is evicted from the DRAM cache. The main purpose for doing that is to reduce write traffic to flash thereby extending the lifetime of flash.

There is a main issue here though: fault tolerance. The data in the DRAM cache will be lost in the event of power failure and system crash. This issue must be solved before it is considered as a practical choice. Unlike main memory, secondary storage is supposed to be data safe by operating systems. We will provide our solution to this issue in Chapter 8.

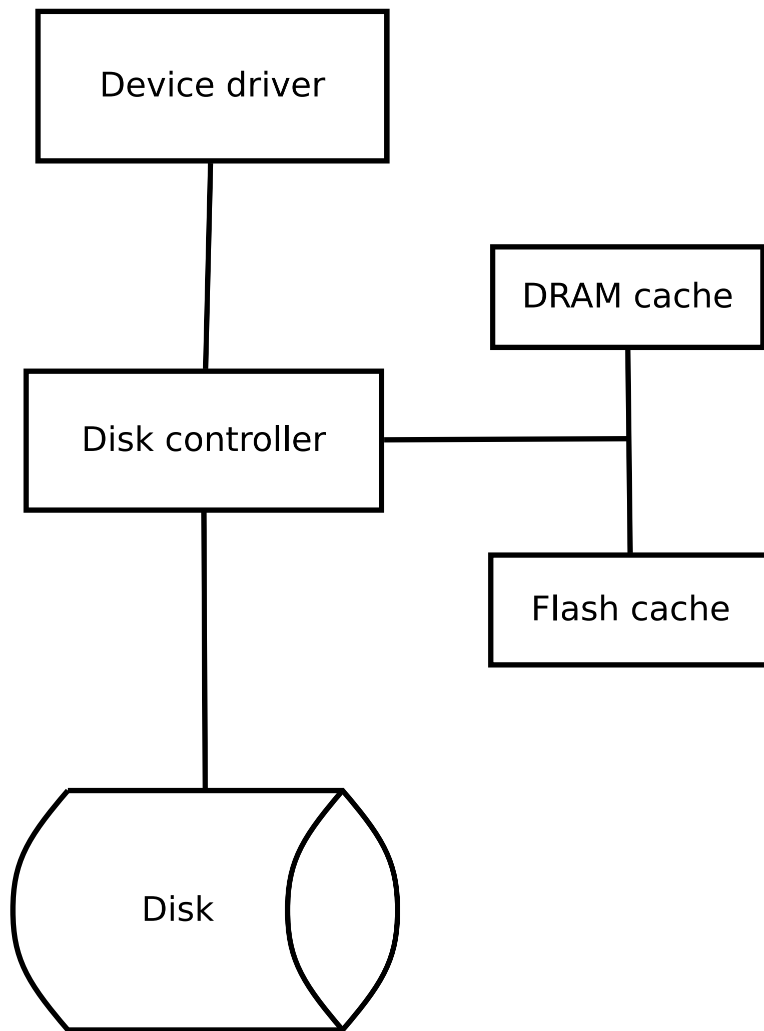


Figure 3.1: System architecture of Hybrid Hard Drives

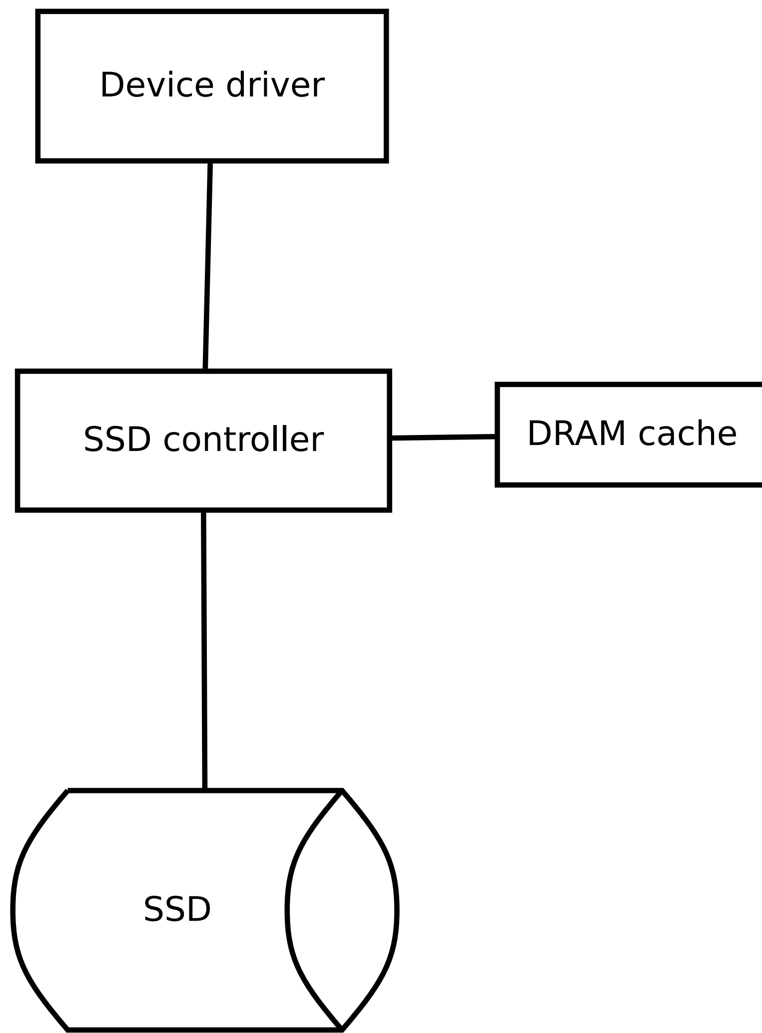


Figure 3.2: System architecture of Solid State Drives

Chapter 4

Comparison of Traffic Mitigation Techniques

The idea of traffic mitigation is to have a non-volatile cache in front of the flash medium. Apart from our solution (DRAM with a supercapacitor backup power), such kind of cache can be:

- Battery-backed DRAM
- SLC as a write cache for MLC
- PCM as a write cache
- HDDs

Since a thorough comparative analysis of all the options is beyond the scope of this disseration, we briefly describe a few other designs and compare them qualitatively with our solution.

The rest of the chapter is organized as follows: Section 4.1 compares our solution with battery-backed DRAM. In Section 4.2, we discuss SLC as a write cache. In Section 4.3, we talk about PCM. Section 4.4 shows HDDs as a write cache. Finally, in Section 4.5, we present a summary of several traffic mitigation techniques.

4.1 Battery-backed DRAM as a write cache

The difference between battery-backed DRAM and our solution lies only in the way the backup power is supplied. As calculated in Chapter 8, a period of 10 seconds is long enough for the contents in DRAM to be transferred into flash. Therefore, we do not need a long-lasting (hours) power supply in the presence of flash, a supercapacitor backup power is

a perfect fit. It does not suffer many drawbacks of batteries, such as regular maintenance or replacement, limited charge/discharge cycles, slow charge, degrade with shallow discharge, etc. More discussions are presented in Chapter 8.

4.2 SLC as a write cache for MLC

SLC can be a write cache for MLC due to the fact that each SLC block has more write cycles. However, SLC is expensive. To be feasible, the size of SLC must be small. The problem with SLC is that SLC also suffers from limited write endurance although write endurance is 10 times better than MLC. Therefore, SLC endurance should be taken into account as a design constraint along with MLC endurance. Since SLC has 10 times the endurance of MLC, to reach the same lifetime, SLC can be a tenth as large as MLC. According to Soundararajan et al. [8], if SLC receives twice as many writes as MLC (where MLC receives 50% write savings), SLC should be a fifth as large as MLC. Hence, the larger the backing MLC SSD, the larger the SLC cache. For example, a 80GB MLC SSD, 16GB SLC is needed. It is believed that 16GB SLC will continue to be expensive enough for a 80GB MLC SSD to afford. In contrast, in our solution, the size of primary store has little impact on the size of cache, i.e., the size of cache alone determines the write savings no matter how large the primary store is.

4.3 PCM as a write cache

Phase-change memory (also known as PCM, PRAM, PCRAM, etc.) is a type of non-volatile computer memory. According to ITRS [7], it is in the status of development underway and at the border of qualification/pre-production. PCM is one of a number of new memory technologies competing in the non-volatile role with the almost universal flash memory. PCM is a potential choice to be a write cache for SLC/MLC due to the following characteristics [49]:

- Nonvolatile

- Fast read speeds: Access times comparable to DRAM.
- Fast write speeds: Significant write speed improvement over NOR and NAND and no erase needed.

However, The write speed of PCM is far less than that of DRAM. Additionally, PCM still suffers write endurance issue (10^8) although it improves a lot over SLC/MLC. It would still be a concern for write-intensive applications. It is not yet clear that how the future of PCM will be in terms of price and the production readiness. Only time will tell. However, our study does not exclude the use of PCM.

4.4 HDDs as a write cache

HDDs have been proposed to be a write cache for flash to take advantage of the fact that a SATA disk drive can deliver over 80 MB/s of sequential write bandwidth. Two observed characteristics support the use of HDDs as a write cache. First, at block level, there are many overwrites of a small set of popular blocks. Second, reads do not follow writes immediately, which gives the write cache a grace period to flush the contents into flash before reads. There are two competing imperatives: On one hand, data should stay longer in write cache in order to catch more overwrites. On the other hand, data should be flushed into flash in advance to avoid expensive read penalty from HDDs. Therefore, many triggers to flush must be designed to achieve the goal of more overwrites with less read penalty, which makes it quite complicated. In contrast, data in our solution can stay as long as the capacity of the DRAM cache allows since there is no read penalty.

Flash-based SSDs have many advantages over HDDs in terms of:

- Acoustic levels: SSDs have no moving parts and make no sound unlike HDDs.
- Mechanical reliability: SSDs contain no moving parts thereby virtually eliminating mechanical breakdowns.

- Susceptibility to environmental factors: Compared to traditional HDDs, SSDs are typically less susceptible to physical shock. Additionally, they have wider temperature ranges.

- Weight and size: The weight of flash memory and the circuit board material are very light compared to HDDs.

- Power consumption: High performance flash-based SSDs generally require 1/2 to 1/3 the power of HDDs.

- Magnetic susceptibility: SSDs are not concerned about magnets or magnetic surges, which can alter data on the HDD media.

By introducing HDDs into SSDs, the device as a whole will lose its advantages in the above aspects. Therefore, using HDDs as write cache of SSDs may hinder their wider applications.

Our experimental results show no less write savings using DRAM cache than using HDD cache although the comparison may not be precise since we use different traces from Soundararajan et al. [8].

4.5 Summary

The comparison between our research and several others is presented in Table 4.1. As we can see from the table, log-structure has been employed in [8] [21] [47]. As mentioned in [8], log-structure is good at writing but it incurs read-penalty.

Technique→	**Kim et al.** [21]	**Sun et al.** [47]	**Qureshi et al.** [48]	**Soundararajan et al.** [8]	**Ours**
Media to reduce traffic	PCM	PCM as a log region of flash data region	DRAM	Disk-based write caches	DRAM (supercapacitor)
Traffic handled	Metadata	Metadata	All data	All data	All data
Update speed	Slow (PCM)	Slow (PCM)	Fast (DRAM)	Slow (HDD)	Fast (DRAM)
Role of flash	Main storage	Main storage	N/A	Main storage	Disk cache/main storage
Usage	Embedded devices	Secondary storage	Main memory	Secondary storage	Secondary storage
Workload characteristics?	Block device	Block device	Main memory	Block device	Block device
Read penalty due to log structured writes?	Yes	Yes	No	Yes	No
Data loss upon power failure?	No	No	Yes	No	No
Comparative added cost?	High	High	Low	High	Low
Matured technology?	No (PCM)	No (PCM)	No(PCM)	Yes (HDD)	Yes (DRAM)

Table 4.1: Comparison of traffic mitigation techniques

Chapter 5

Methodology

Extensive experiments were conducted with a disk simulator. Our simulator was based on DiskSim 4.0 and Microsoft SSD add-on, which were implemented in C. In this chapter, we describe the methodology we used in this dissertation.

The rest of the chapter is organized as follows: Section 5.1 presents the DiskSim 4.0 simulator. In Section 5.2, we discuss system setup. In Section 5.3, we talk about workloads we used. In Section 5.4, metrics are presented. Section 5.5 shows the modifications we did on DiskSim 4.0. Finally, Section 5.6 presents validation of the modified DiskSim 4.0.

5.1 Simulator

We evaluated the proposed architecture using DiskSim 4.0 [50] with flash extension. DiskSim is a widely used disk drive simulator both in academia and industry alike. DiskSim is an event-driven simulator. It emulates a hierarchy of storage components such as buses and controllers as well as disks. In 2008, Microsoft research implemented an SSD module [51] on top of DiskSim. Based on that, we made some changes to the code to reflect our architecture.

5.2 System Setup

The system environment in which we run our simulator is presented in Table 5.1.

Hardware	Parameters	Descriptions
	CPU	Intel (R) Core (TM) 2 Quad Q6600 2.4 GHz
	L2 cache	4096KB
	Memory	2GB
	Storage	Intel SATA X25-M 80GB SSD
Software	Ubuntu	10.10 (Linux 2.6.35-22 generic)
	gcc/g++	v4.4.5
	bisson	2.4.1
	flex	2.5.35
	GNU bash	4.1.5
	perl	v5.10.1
	python	2.6.6
	gnuplot	4.4
	dia	0.97.1
	pdfTeX	3.1415926-1.40.10-2.2 (TeX Live 2009/Debian)
	pdfcrop	1.2
	latex2rtf	2.1.0
	latex2rtf-gui	0.5.3
	texmaker	2.0
	emacs	23.1.1 (GTK+ version 2.20.0)

Table 5.1: System environment

	Dev[a]	TIOR[b]	Reads	IOPS[c]	ARS[d](KB)
OpenMail	080	51295	36373 (70%)		5.0/10.0/6.5
	096	363789	119400 (32%)	98.29	7.8/7.3/7.5
	097	366223	117530 (32%)		7.9/7.5/7.6
	098	421270	219553 (52%)		3.6/5.6/4.6
	099	424887	227744 (53%)		3.5/5.6/4.5
	100	295536	152372 (51%)		3.5/5.4/4.4
UMTR	0	1439434	1439434 (100%)	5.27	15.2/-/15.2
Web3	1	1410370	1409113 (99%)		15/26/15
	2	1410493	1410492 (99%)		15/8.0/15.5
	3	489	487 (99%)		15/8.0/15
	4	486	486 (100%)		15.1/-/15.1
	5	434	434 (100%)		16.3/-/16.3
Synthetic workloads	0	10000	6600 (66%)	40.05	6.4/6.2/6.3

[a] Device No
[b] Total I/O Requests
[c] I/O Per Second
[d] Average Request Size in KB

Table 5.2: Workload characteristics

5.3 Workloads and Traces

Many studies of storage performance analysis use traces of real file system requests to produce more realistic results. The traces we used are from HP [52] [53] [54]: OpenMail. In addition, we used disk traces from University of Massachusetts Trace Repository (UMTR) [55] to test the impact of different update policies on the disk behavior of enterprise level applications like web servers, database servers, and web search. We also generate synthetic workloads that represent typical access distributions and approximate real disk usage. The characteristics of workloads used in this paper are shown in Table 5.2. We selected the first device (which can represent the workload characteristics in class) within each trace to report simulation results.

OpenMail [52]: It was a one-hour trace from five servers running HP's OpenMail, collected during the servers' busy periods. The trace was collected in 1999.

UMTR [55]: There are two kinds of traces: OLTP application I/O and search engine I/O. The former includes two I/O traces (Financial1.spc and Financial2.spc) from OLTP applications running at two large financial institutions. The later includes three traces (Websearch1.spc, Websearch2.spc, and Websearch3.spc) from a popular search engine. These traces are made available courtesy of Ken Bates from HP, Bruce McNutt from IBM and the Storage Performance Council.

Synthetic workloads [50]: The synthetic workloads were generated using DiskSim 4.0 workload generator. The generator can be configured to generate a wide range of synthetic workloads. In addition, probability-distribution parameters can be set to uniform, normal, exponential, Poisson, or twovalue.

5.4 Performance Metrics

Response time and *throughput* are generally two important metrics in measuring I/O performance. Response time starts from a request being issued until the request is served. Throughput measures the ability of a system in a form of the number of I/Os per second. The difference between *Response time* and *Throughput* is whether we measure one task (*Response Time*) or many tasks (*Throughput*). According to Hsu and Smith [1], throughput is hard to be quantified for trace-driven simulation in that the workloads are constant. However, they maintain that throughput can be estimated by taking the reciprocal of the average service time, which tends to be optimistic estimate of the maximum throughput. Additionally, we examine the *miss ratio* of the read cache and the write cache. The miss ratio is the fraction of I/Os that need to access physical devices (HDDs). Finally, flash endurance is measured using the write traffic (Bytes) to the flash. To compare different policies (early update policy and lazy update policy), we introduce *Relative Traffic* γ, which is defined as:

$$\gamma = \frac{\beta}{\alpha} \tag{5.1}$$

where

- β is the write traffic to flash.

- α is the write traffic to device.

The ideal flash lifetime, which does not take write amplification into consideration (write amplification will be discussed later), can be expressed as:

$$\eta = \frac{\delta\varepsilon}{\lambda} \tag{5.2}$$

where

- η is the flash lifetime in days.

- δ is the maximum of the flash write cycles (10K-100K) depending on the type of flash.

- ε is the capacity of the device.

- λ is the write traffic per day.

As we can see from 5.2, flash lifetime is related to rated write cycles δ and the capacity of the device ε. Since we will use the same kinds of flash and same size of flash to compare early update policy to lazy update policy, we would like to eliminate these two factors from the equation. In order to do so, we introduce *relative lifetime* φ:

$$\varphi = \frac{\eta 1}{\eta 2} = \frac{\frac{\delta\varepsilon}{\lambda 1}}{\frac{\delta\varepsilon}{\lambda 2}} = \frac{\lambda 2\mu}{\lambda 1\mu} = \frac{\beta 2}{\beta 1} = \frac{\frac{\beta 2}{\alpha}}{\frac{\beta 1}{\alpha}} = \frac{\gamma 2}{\gamma 1} \tag{5.3}$$

where

- $\eta 1$ is the flash lifetime for early update policy.

- $\eta 2$ is the flash lifetime for lazy update policy.

- μ is the time in days the flash is used.

- $\lambda 1$ is the write traffic to flash per day for early update policy.

- $\lambda 2$ is the write traffic to flash per day for lazy update policy.

- $\beta 1$ is the write traffic to flash for early update policy.

- $\beta 2$ is the write traffic to flash for lazy update policy.

- α is the total traffic to the device.

- $\gamma 1$ is relative traffic for early update policy.

- $\gamma 2$ is relative traffic for lazy update policy.

From 5.3, we see that relative lifetime φ can be expressed using relative traffic γ. The advantage of using relative traffic to compare relative lifetime is that the rated write cycles δ and the capacity of the device ε are taken out of the equation.

The flash lifetime in practice is smaller than the ideal flash lifetime due to a factor, called write amplification. Write amplification is referred to as the phenomenon that n bytes of write traffic from a file system will be translated into nm ($m > 1$) bytes of write traffic to flash. There are several factors [56] that contribute to write amplification. First, write amplification is caused by wear leveling, where cold/hot data swapping consumes extra write cycles. Second, garbage collection contributes to write amplification. In garbage collection, in order to reclaim invalid pages, the valid pages within the reclaiming blocks need to be relocated, which consumes extra write cycles. Write amplification varies for different FTL. Usually, simpler FTLs own larger write amplification while more complex FTLs have smaller write amplification. Many efforts have been made to minimize write amplification to extend flash lifetime. According to Soundararajan et al. [8], flash lifetime can be an order of

magnitude worse than the optimum even for advanced FTLs, such as Intel X25-M MLC SSD.

Due to the write amplification, it is not straightforward to map between reduced write traffic and increased lifetime. However, decreasing the write traffic in half will at least double its lifetime because of reduced write amplification [8].

5.5 DiskSim 4.0 modifications

In doing our research, we made some modifications on DiskSim4.0. First, we added two trace formats: SRT 1.6 and SPC. In addition, we added a trace filter to control what types of requests will be fed into our simulator. Second, we added secondary flash cache on top of the primary disk cache. Last, we fixed bugs with regard to the incompatibility issue of Type 3 Smart Controller for Microsoft SSD add-on.

5.5.1 Trace formats: SRT1.6 and SPC

DiskSim 4.0 supports many trace formats including SRT 1.4, but it does not support HP SRT1.6 and SPC. Many traces, like HP OpenMail and Cello, use SRT1.6. SPC is a trace format designed by the Storage Performance Council. University of Massachusetts Trace uses SPC format.

Our implementation of SRT 1.6 is based on HP WinTraceto SRT. The source code for SRT 1.6 is in disksim_srt.c. Our implementation of SPC complies with SPC specification 1.01. The source code for SPC is in disksim_spc.c. Both are included in disksim4.0-ms-ssd-t2-c3 version. We are planning to make it publicly available.

In addition to adding the above two trace formats, a trace filter is added. With the filter, you can select:

- Device number
- Read only

- Write only
- Set dev=0
- Set maxmum bcount

The filter can be configured via Global parameter in .parv:

```
disksim_global Global {
    Init Seed = 42,
    Real Seed = 42,

# uncomment the following line if you just want requests for dev=80
# Device to Trace = 80,

# uncomment the following line if you only trace reads
# Trace Read Only = 1,

# uncomment the following line if you only trace writes.
# Trace Write Only = 1,

# uncomment the following line if you want to set dev=0
# Map Devices to devno-0 = 1,

# uncomment the following line if you only want bcount <= 512
# Max Bcount = 512

    Stat definition file = statdefs
}
```

5.5.2 Secondary flash cache

DiskSim 4.0 does not support a secondary flash cache that our research needs to have. The original system architecture of DiskSim 4.0 is shown in Figure A.1. Therefore, we implemented a secondary flash cache layer on DiskSim 4.0 as shown in Figure 5.1. Since we concentrate on the traffic savings, we only implemented a simplified model of a flash device that has a constant access time, which can be adjusted by parameter configurations.

5.5.3 Smart controller and MS SSD add-on

Microsoft research implemented an SSD module on top of DiskSim 4.0. Unlike the disk implementation, the SSD implementation does not have a read or write cache. One way to have a read or write cache is to use Smart Controller (type:3) (see Figure 5.1). However, the SSD add-on is not compatible with Smart Controller. The problem is rooted in the slightly different ways that dumb controller and smart controller handle data transfer. We fixed the bugs and added the support of Smart Controller to the SSD add-on. Our version disksim4.0-ms-ssd-t2-c3 contains the update.

5.6 Validation

DiskSim 4.0 comes with a set of validation tests (runvalid), which are used to test the simulator. Runvalid uses a series of validation data to validate the simulator. The validation data were from a logic analyzer attached to the SCSI bus. Validation was achieved by comparing measured and simulated response time distributions. Our modified version of DiskSim 4.0 produces the same values as DiskSim 4.0 for all the validation tests as shown in Table 5.3.

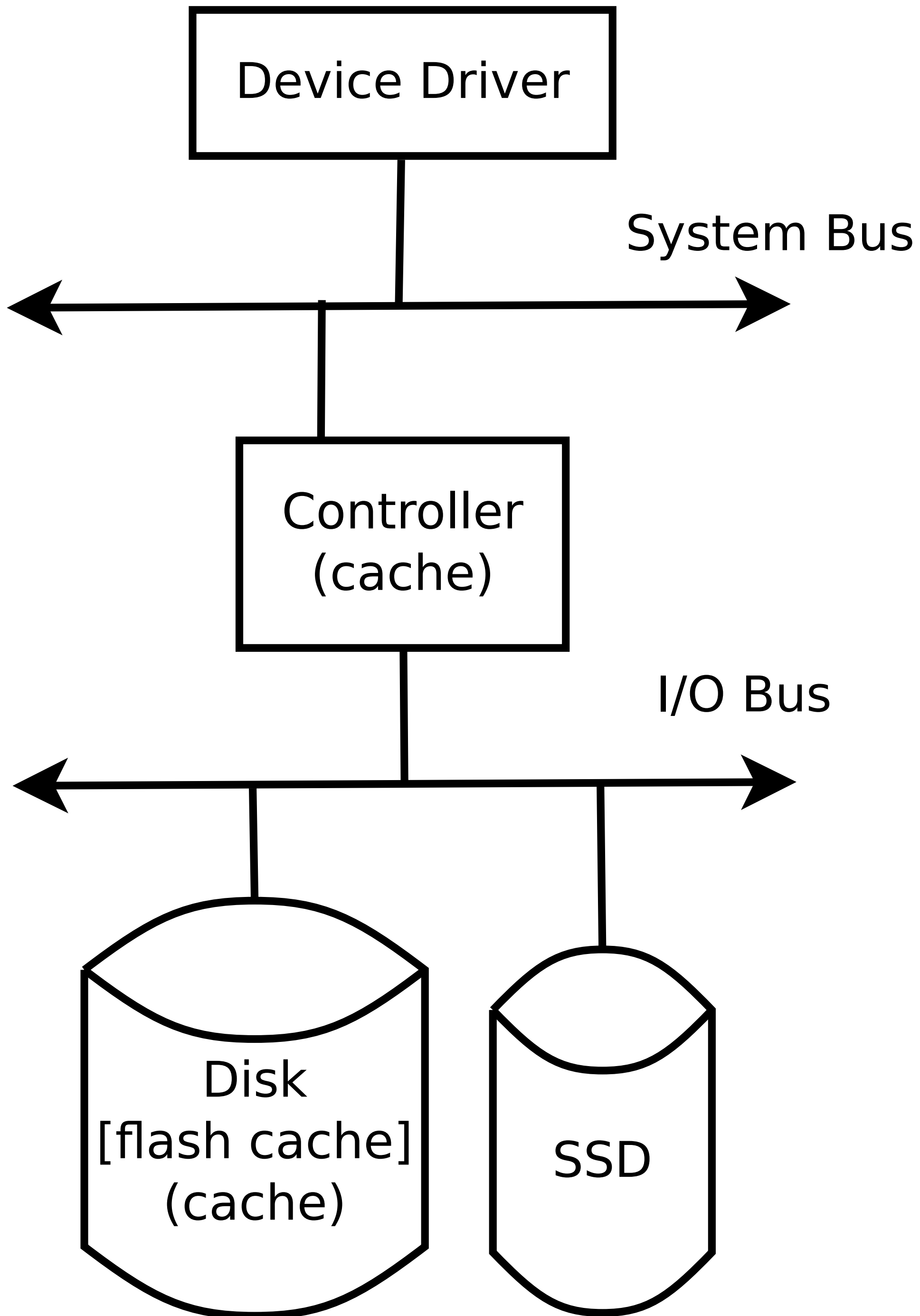


Figure 5.1: System architecture of modified DiskSim 4.0

Tests	DiskSim 4.0	Modified DiskSim 4.0
QUANTUM_QM39100TD-SW	rms[a]= 0.377952	rms = 0.377952
SEAGATE_ST32171W	rms = 0.347570	rms = 0.347570
SEAGATE_ST34501N	rms = 0.317972	rms = 0.317972
SEAGATE_ST39102LW	rms = 0.106906	rms = 0.106906
IBM_DNES-309170W	rms = 0.135884	rms = 0.135884
QUANTUM_TORNADO	rms = 0.267721	rms = 0.267721
HP_C2247_validate	rms = 0.089931	rms = 0.089931
HP_C3323_validate	rms = 0.305653	rms = 0.305653
HP_C2490_validate	rms = 0.253762	rms = 0.253762
Open synthetic workload	10.937386ms[b]	10.937386ms
Closed synthetic workload	87.819135ms	87.819135ms
Mixed synthetic workload	22.086628ms	22.086628ms
RAID 5 at device driver	22.861326ms	22.861326ms
Disk arrays at device driver	34.272035ms	34.272035ms
Memory cache at controller	24.651367ms	24.651367ms
Cache device/controller	28.939379ms	28.939379ms
Simpledisk	13.711596ms	13.711596ms
3 different disks	10.937386ms	10.937386ms
Separate controllers	10.937386ms	10.937386ms
HP srt trace input	48.786646ms	48.786646ms
ASCII input	13.766948ms	13.766948ms
Syssim system integration	8.894719ms	8.894719ms

[a] Root Mean Square error of differences
[b] Average Response Time in millisecond

Table 5.3: Validation of modified DiskSim 4.0

Chapter 6

Traffic Savings for Flash as a Victim Disk Cache

In this chapter, we focus research on flash used as a victim disk cache, whose architecture was shown in 3.1. There are two levels of disk caches: primary DRAM disk cache and secondary flash disk cache. We concentrated on the relationship between the two levels of disk caches: when is the right time to update the flash disk cache? Compared with the early update policy, our study shows that using flash as a victim cache (which corresponds to lazy update policy) can save a lot of lifetime. At the same time, the performance in terms of response time and miss ratio is improving as well.

The rest of the chapter is organized as follows: Section 6.1 introduces the concept of early update and lazy update policy. In Section 6.2, we discuss lazy update policy for reads. In Section 6.3, we talk about lazy update policy for writes. In Section 6.4, the benefits of lazy update policy are presented. Section 6.5 shows the simulation parameters. Section 6.6 talks about performance metrics used in this chapter. Finally, Section 6.7 presents the experimental results.

6.1 Early Update vs. Lazy Update

Early update means the flash is updated as soon as the DRAM cache is updated. Lazy update is referred to as the policy that the flash is updated only when the data is evicted from DRAM cache. In other words, flash acts as a victim device of the DRAM cache. Traditionally, early update policy is employed. For examples, Bisson et al. [18] and Kgil et al. [24] [25] [26] use early update policy. The flow chart of early update policy is shown in Figure 6.1 and Figure 6.2.

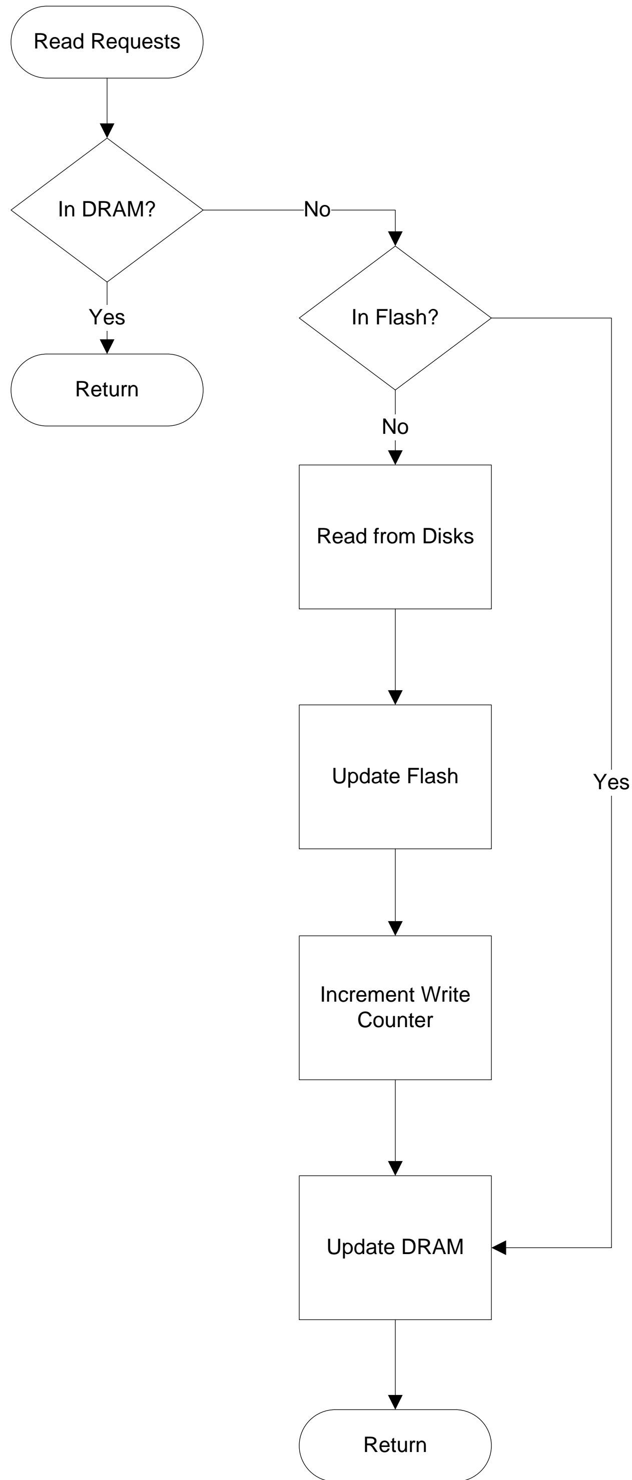


Figure 6.1: Early update for reads

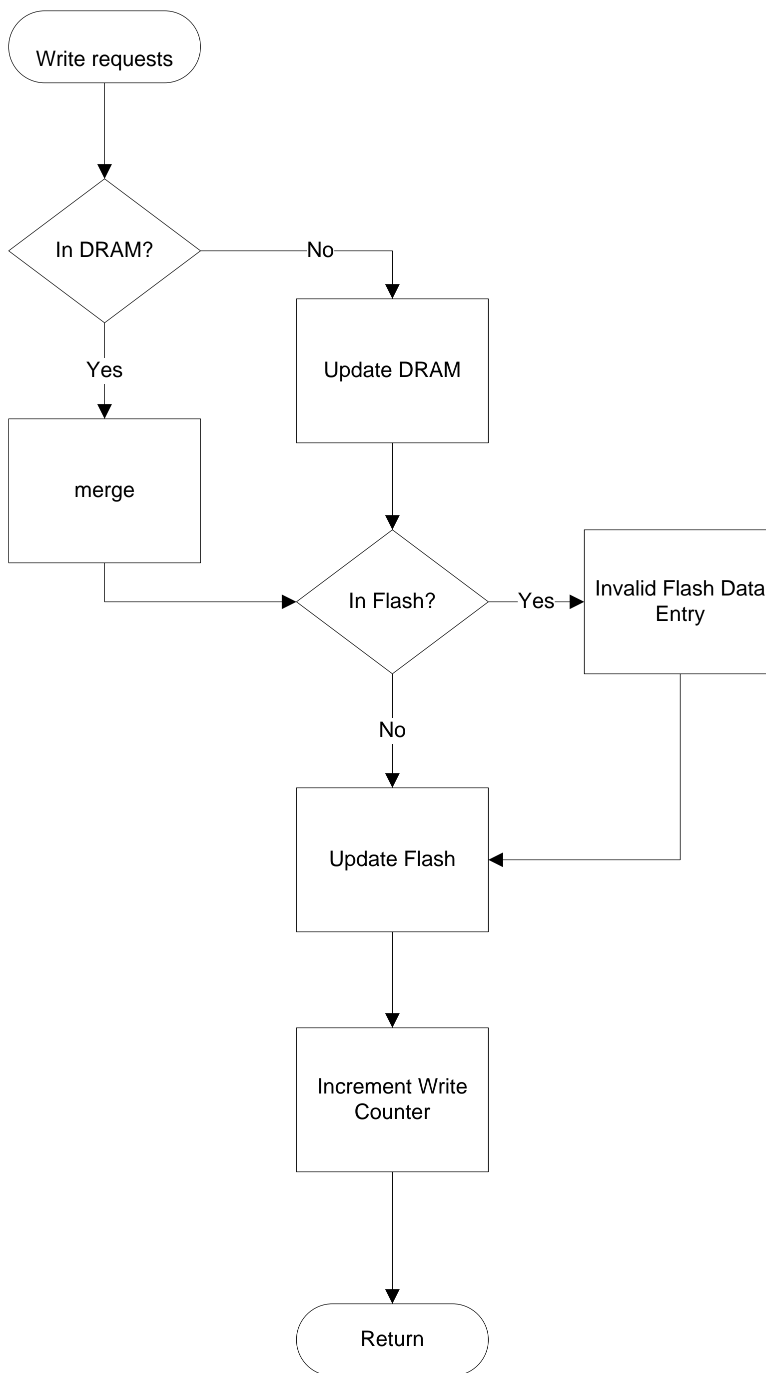


Figure 6.2: Early update for writes

6.2 Lazy Update Policy for Reads

When a read request arrives, DRAM cache is first checked as shown in Figure 6.3. If found in DRAM cache, then the request can be served through DRAM cache. If not, then check with flash. If it has an entry in it, the request can be satisfied by flash. The data will also be copied into DRAM cache. If the data is not in flash either, then the data is fetched from HDDs and it is cached in DRAM cache. But flash is not updated until the data is evicted.

6.3 Lazy Update Policy for Writes

When a write request arrives, DRAM cache is first checked as shown in Figure 6.4. If found in DRAM cache, the request will be merged. If not, allocate an entry in DRAM cache. If there is no space, the least used clean data will be evicted. The evicted data will be copied into flash. If all data in DRAM cache are dirty, the write request has to wait for the dirty data to be retired into HDDs.

6.4 The Benefits of Lazy Update Policy

Lazy update policy benefits from the fact that block devices receive many overwrites of a small set of popular blocks. One cause for overwrites is that many file systems enforce a 30-second rule [54], which flushes buffered writes to disks every 30 seconds for the sake of data integrity. Soundararajan et al. [8] have found that, on average, 54% of the total overwrites occur within the first 30 seconds, which confirms the role that 30-second rule plays. Some file systems, such as ext2, flush metadata more frequently (e.g., every 5 seconds) [57]. With lazy update policy, overwrites can be coalesced, thereby reducing the write traffic to flash significantly.

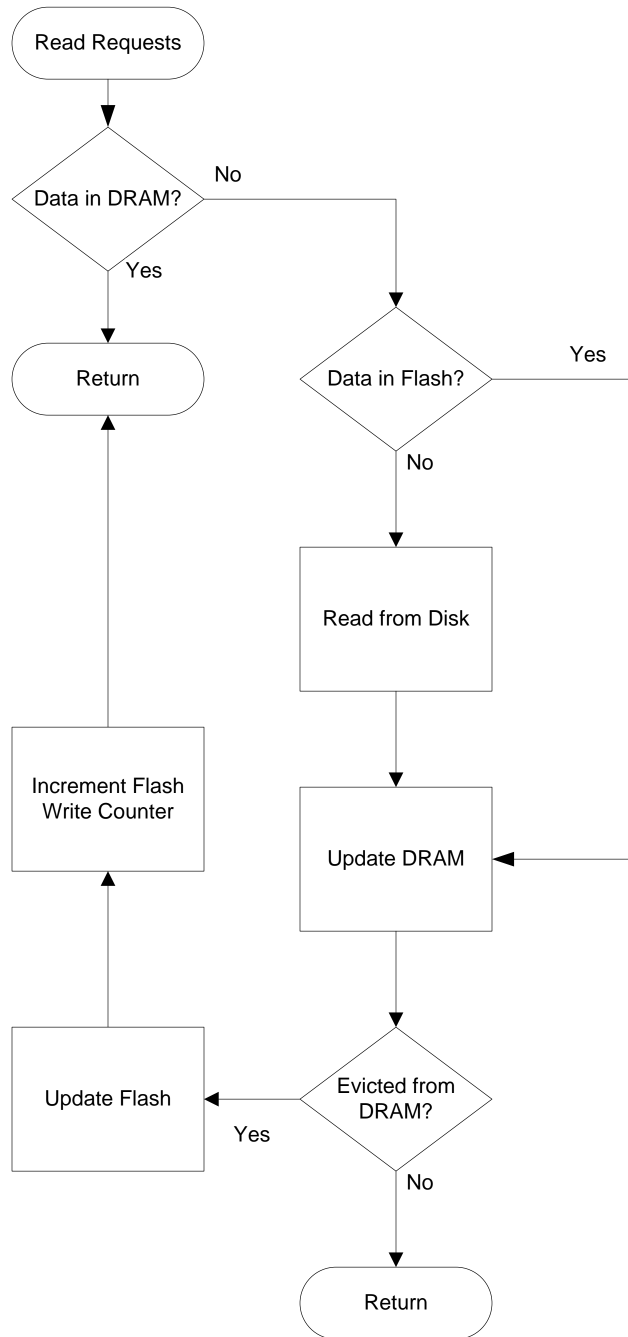


Figure 6.3: Lazy update for reads

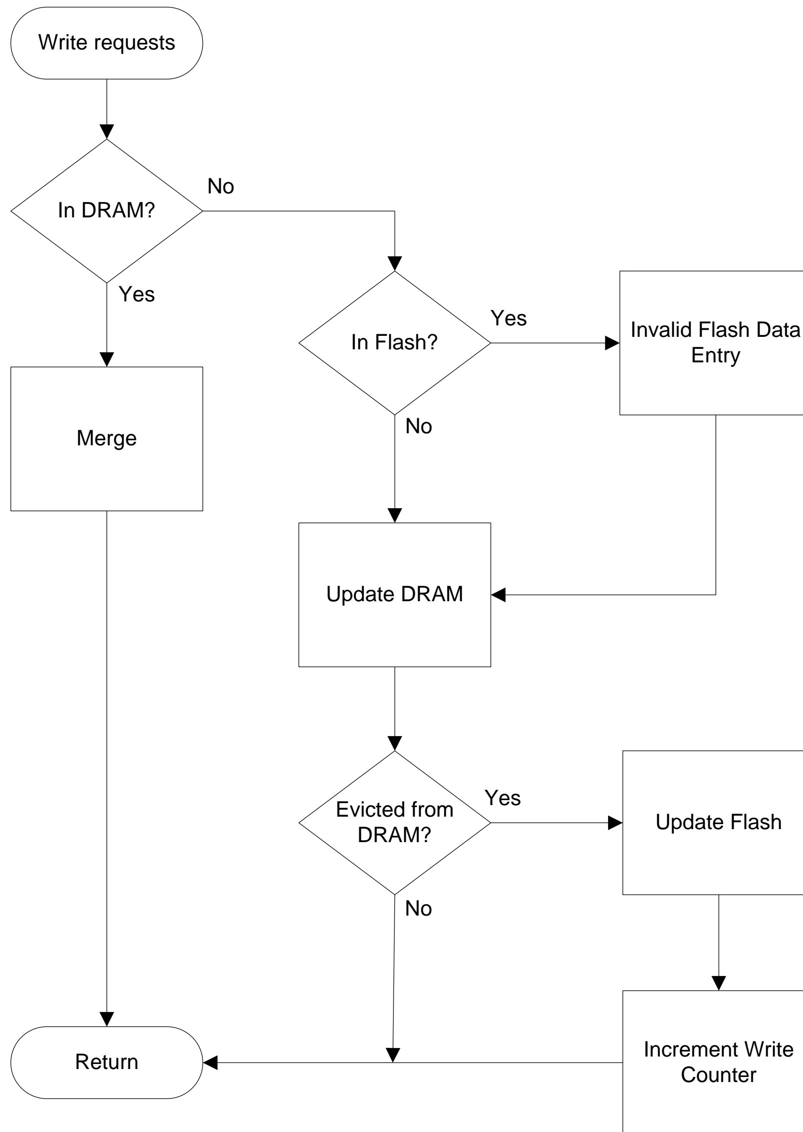


Figure 6.4: Lazy update for writes

Parameter	Value
Form	3.5”
Capacity	9100MB
Seek time/track	7.5/0.8ms
Cache/Buffer	1024KB
Data transfer rate	19MB/S int 40MB/S ext
Bytes/Sector	512

Table 6.1: Disk parameters (QUANTUM QM39100TD-SW)

6.5 Simulation Parameters

In our simulation, we choose QUANTUM QM39100TD-SW SCSI hard disk drive, which was one of 10 disk drives that have been validated for DiskSim 4.0. Its basic parameters are shown in Table 6.1. We have tried other brands of hard disk drives, which have little impact on our findings since we concentrated our research on lifetime of flash and disk cache. Therefore, in this dissertation, we use QUANTUM QM39100TD-SW to report our experimental results.

The parameter file (victim.parv) we used in running DiskSim is listed below in the two-column text. In addition, we changed the following parameters via command line parameter interface:

- DRAM cache size
- Flash cache size
- Devno

```
disksim_global Global {
Init Seed = 42,
Real Seed = 42,
Device to Trace = 1,
Map Devices to devno-0 = 1,
With Flash Cache = 1,
Flash Update Policy = 1,
Flash Cache Size = 1,
Flash Page Size = 8,
Flash Cost Per Page = 0.0,
Stat definition file = statdefs
}

disksim_stats Stats {
iodriver stats = disksim_iodriver_stats {
Print driver size stats = 1,
Print driver locality stats = 0,
Print driver blocking stats = 0,
Print driver interference stats = 0,
Print driver queue stats = 1,
Print driver crit stats = 0,
Print driver idle stats = 1,
Print driver intarr stats = 1,
Print driver streak stats = 1,
Print driver stamp stats = 1,
Print driver per-device stats = 1 },
bus stats = disksim_bus_stats {
Print bus idle stats = 1,
Print bus arbwait stats = 1 },
ctlr stats = disksim_ctlr_stats {
Print controller cache stats = 1,
Print controller size stats = 1,
Print controller locality stats = 1,
Print controller blocking stats = 1,
Print controller interference stats = 1,
Print controller queue stats = 1,
Print controller crit stats = 1,
Print controller idle stats = 1,
Print controller intarr stats = 1,
Print controller streak stats = 1,
Print controller stamp stats = 1,
Print controller per-device stats = 1 },
device stats = disksim_device_stats {
Print device queue stats = 0,
Print device crit stats = 0,
Print device idle stats = 0,
Print device intarr stats = 0,
Print device size stats = 0,
Print device seek stats = 1,
Print device latency stats = 1,
Print device xfer stats = 1,
Print device acctime stats = 1,
Print device interfere stats = 0,
Print device buffer stats = 1 },
process flow stats = disksim_pf_stats {
Print per-process stats = 1,
Print per-CPU stats = 1,
Print all interrupt stats = 1,
Print sleep stats = 1
}
} # end of stats block

disksim_iodriver DRIVER0 {
type = 1,
Constant access time = 0.0,
Scheduler = disksim_ioqueue {
Scheduling policy = 3,
Cylinder mapping strategy = 1,
Write initiation delay = 0.0,
Read initiation delay = 0.0,
Sequential stream scheme = 0,
Maximum concat size = 128,
Overlapping request scheme = 0,
Sequential stream diff maximum = 0,
Scheduling timeout scheme = 0,
Timeout time/weight = 6,
Timeout scheduling = 4,
Scheduling priority scheme = 0,
Priority scheduling = 4
}, # end of Scheduler
Use queueing in subsystem = 1
} # end of DRV0 spec

disksim_bus BUS0 {
type = 1,
Arbitration type = 1,
Arbitration time = 0.0,
Read block transfer time = 0.0,
Write block transfer time = 0.0,
Print stats = 0
```

```
} # end of BUS0 spec

disksim_bus BUS1 {
type = 1,
Arbitration type = 1,
Arbitration time = 0.0,
Read block transfer time = 0.0512,
Write block transfer time = 0.0512,
Print stats = 1
} # end of BUS1 spec

disksim_ctlr CTLR0 {
type = 1,
Scale for delays = 0.0,
Bulk sector transfer time = 0.0,
Maximum queue length = 0,
Print stats = 1
} # end of CTLR0 spec

source atlas_III.diskspecs
# component instantiation
instantiate [ statfoo ] as Stats
instantiate [ bus0 ] as BUS0
instantiate [ bus1 ] as BUS1
instantiate [ disk0 .. disk2 ] as
QUANTUM_QM39100TD-SW
instantiate [ driver0 ] as DRIVER0
instantiate [ ctlr0 ] as CTLR0

# system topology
topology disksim_iodriver driver0 [
disksim_bus bus0 [
disksim_ctlr ctlr0 [
disksim_bus bus1 [
disksim_disk disk0 [],
disksim_disk disk1 [],
disksim_disk disk2 []
]
]
]
]

disksim_logorg org0 {
Addressing mode = Parts,
Distribution scheme = Asis,
Redundancy scheme = Noredun,
devices = [ disk0 ],
Stripe unit = 2056008,
Synch writes for safety = 0,
Number of copies = 2,
Copy choice on read = 6,
RMW vs. reconstruct = 0.5,
Parity stripe unit = 64,
Parity rotation type = 1,
Time stamp interval = 0.000000,
Time stamp start time = 60000.000000,
Time stamp stop time =
10000000000.000000,
Time stamp file name = stamps
} # end of logorg org0 spec

disksim_pf Proc {
Number of processors = 1,
Process-Flow Time Scale = 1.0
} # end of process flow spec

disksim_synthio Synthio {
Number of I/O requests to generate =
20000,
Maximum time of trace generated =
10000.0,
System call/return with each request = 0,
Think time from call to request = 0.0,
Think time from request to return = 0.0,
Generators = [
disksim_synthgen { # generator 0
Storage capacity per device = 17938986,
devices = [ disk0 ],
Blocking factor = 8,
Probability of sequential access = 0.7,
Probability of local access = 0.7,
Probability of read access = 0.5,
Probability of time-critical request = 0.2,
Probability of time-limited request = 0.3,
Time-limited think times = [ normal, 30.0,
100.0 ],
General inter-arrival times = [ exponential,
0.0, 25.0 ],
Sequential inter-arrival times = [ normal,
0.0, 0.0 ],
Local inter-arrival times = [ exponential,
0.0, 0.0 ],
```

```
Local distances = [ normal, 0.0, 40000.0 ],
Sizes = [ exponential, 0.0, 8.0 ]
}
] # end of generator list
} # end of synthetic workload spec
```

6.6 Performance Metrics

As discussed in Chapter 5, we use *relative traffic, miss ratio*, and *response time* as performance metrics.

6.7 Experimental Results

We ran OpenMail, Synthetic, and Websearch3 workload traces against early update policy and lazy update policy, during which relative traffic, miss ratio, and average response time were observed. We especially watched the impact of DRAM size and flash size on relative traffic savings. The simulation results are shown in Figure 6.5 through Figure 6.28. Several observations can be made from these figures.

6.7.1 Write Traffic Savings

OpenMail (Figure 6.5 and Figure 6.6): relative traffic with lazy update policy is 50% of that with early update policy when DRAM size reaches 25MB (0.33 for lazy update policy vs. 0.68 for early update policy).

Synthetic workload (Figure 6.7 and Figure 6.8): relative traffic with lazy update policy is 50% of that with early update policy when DRAM size reaches 60MB (roughly 0.5 for lazy update policy vs. 1 for early update policy).

Websearch3 (Figure 6.9 and Figure 6.10): relative traffic does not see improvement with Websearch3 because Websearch3 is a read-only workload. For read-only workloads, there is no overwrite that can be coalesced using lazy update policy.

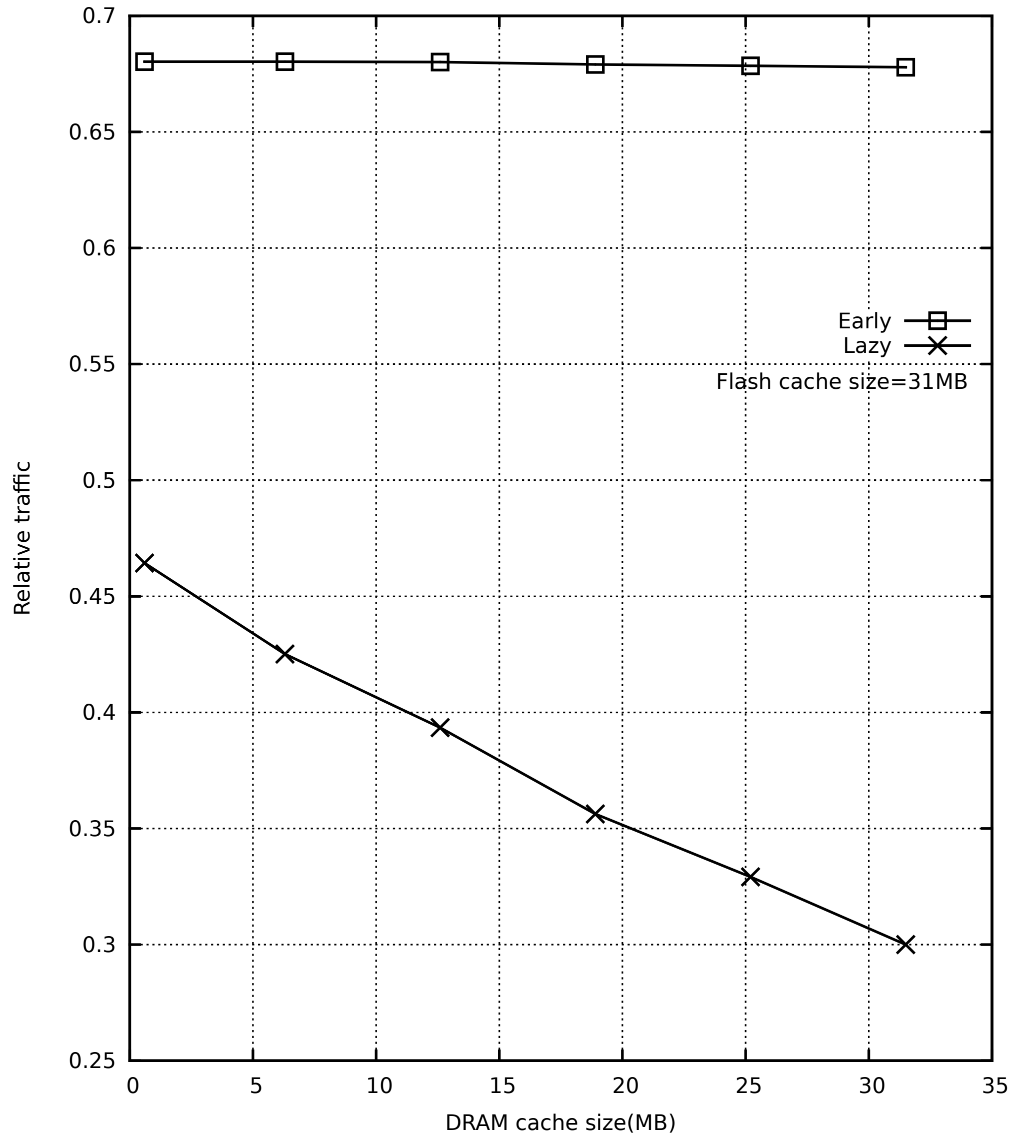


Figure 6.5: Relative traffic for OpenMail

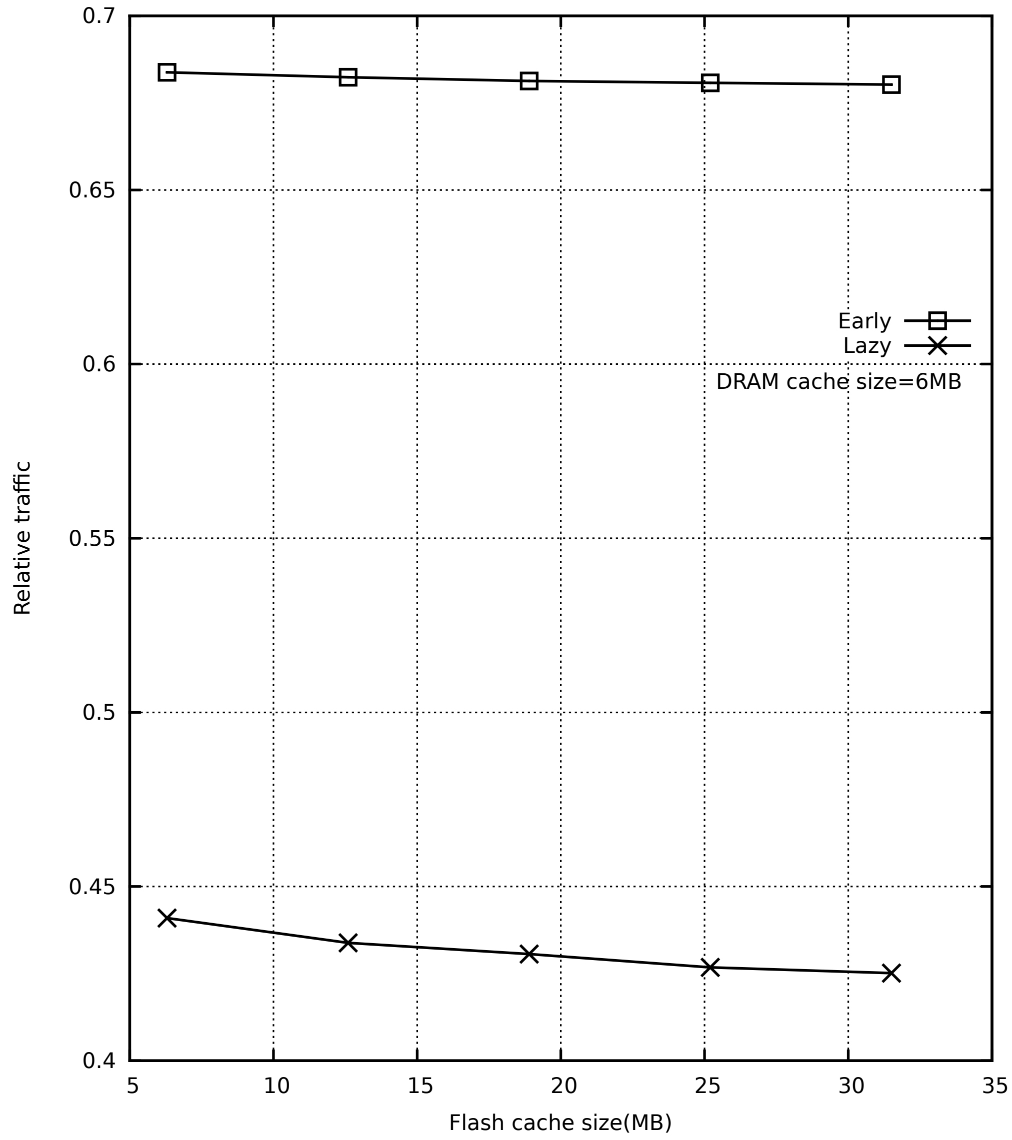


Figure 6.6: Relative traffic for OpenMail

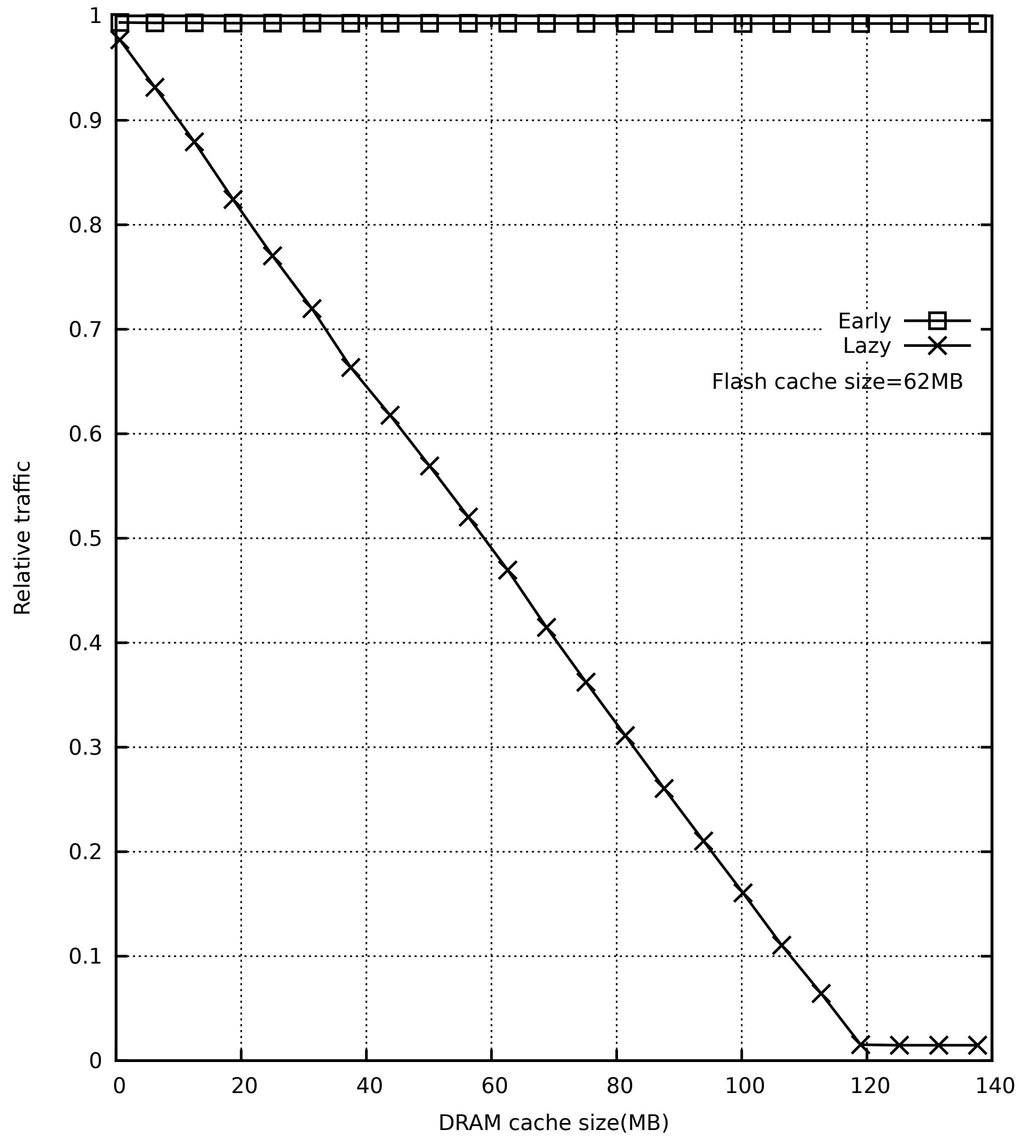


Figure 6.7: Relative traffic for Synthetic workload

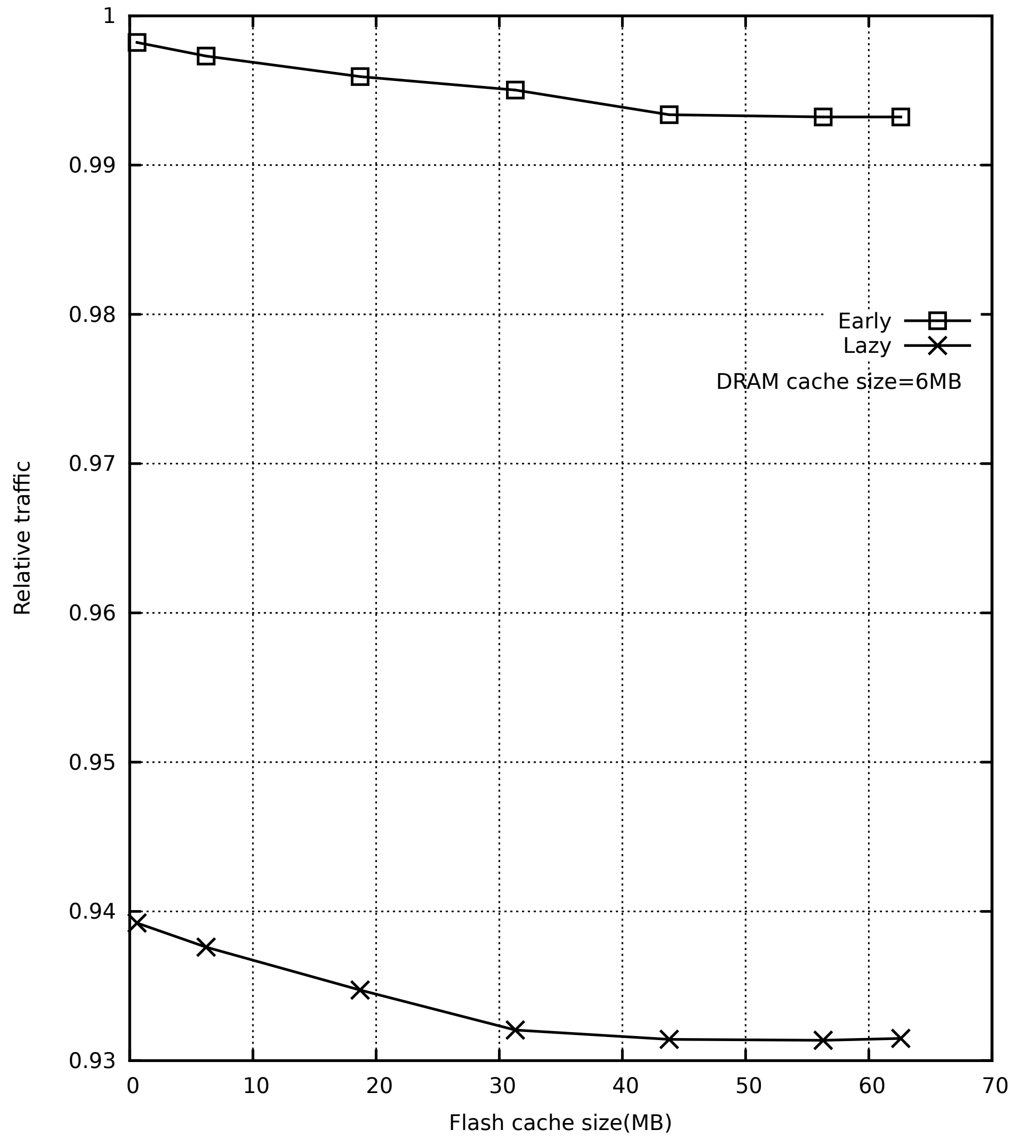


Figure 6.8: Relative traffic for Synthetic workload

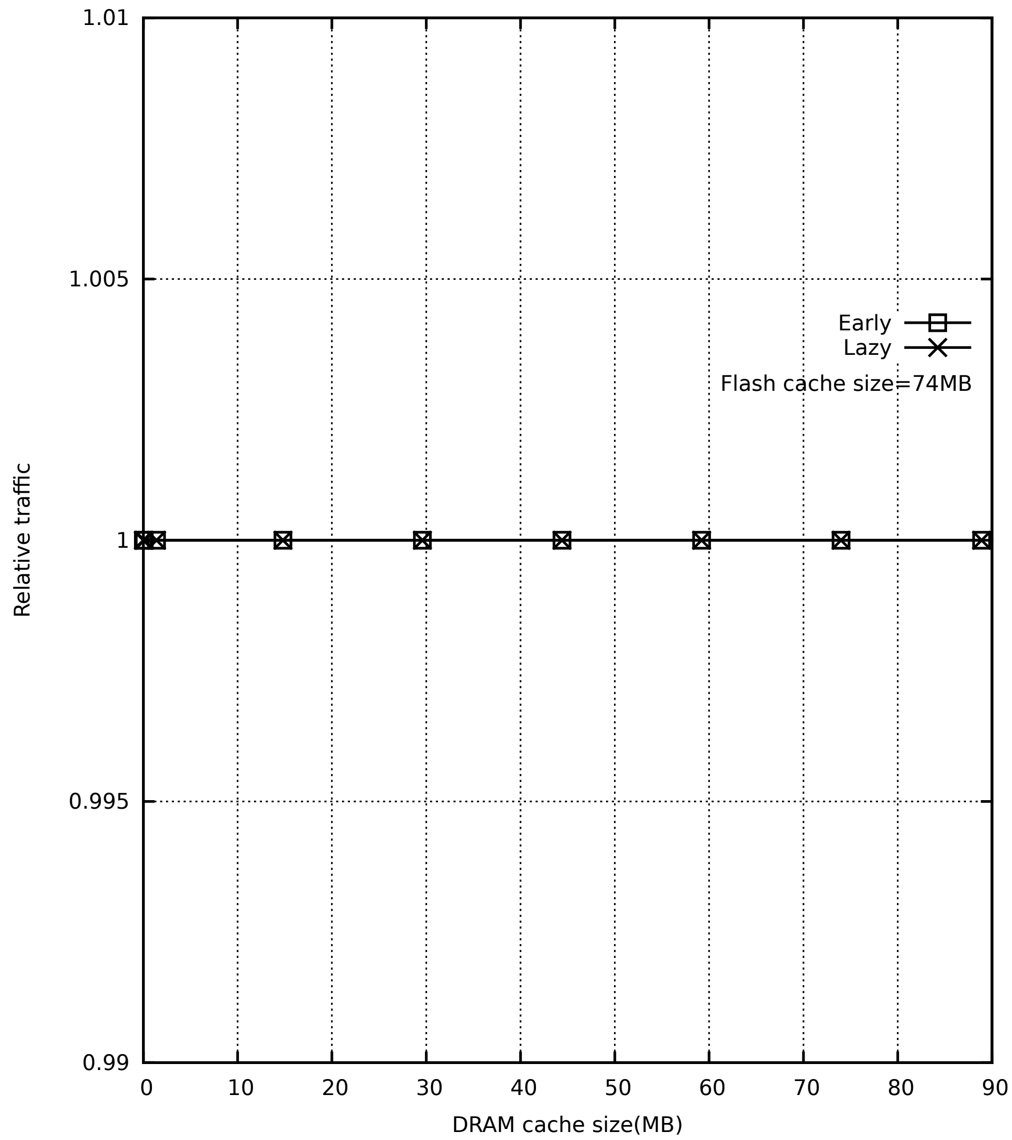


Figure 6.9: Relative traffic for Websearch3

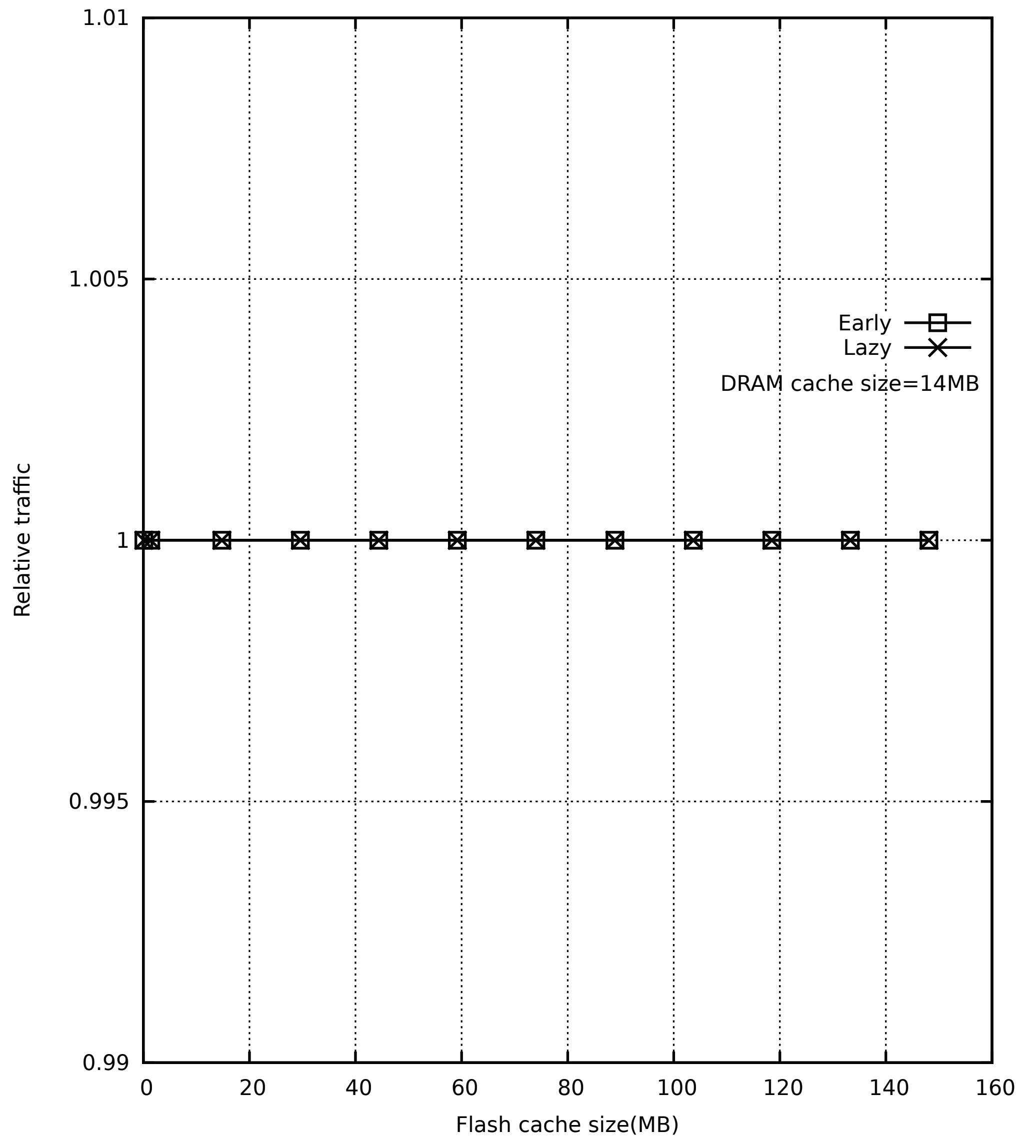


Figure 6.10: Relative traffic for Websearch3

6.7.2 DRAM Size Need Not to be Very Large to be Effective

DRAM size ranges from 15MB to 60MB depending on the workloads, which is shown in Figure 6.5 and Figure 6.7.

6.7.3 Flash size has little effect on Write Traffic Savings

As can be seen from Figure 6.6 and Figure 6.8, flash size has little impact on write traffic savings when compared to varying DRAM size.

We see that the write traffic savings depend mainly on DRAM size rather than flash size. This is a great characteristic since a moderate size of DRAM cache ($<$ 60MB) can extend the flash lifetime significantly no matter how large the flash is.

6.7.4 Miss Ratio Is improved Slightly

From Figure 6.11, Figure 6.12, Figure 6.13 and Figure 6.14, we notice that miss ratio with lazy update policy is slightly better than that with early update policy. However, miss ratio with lazy update policy for Websearch3 does not see any improvement, as shown in Figure 6.15 and Figure 6.16.

6.7.5 Response Time Is Improved Slightly

From Figure 6.17, Figure 6.18, Figure 6.19 and Figure 6.20, we observe that response time with lazy update policy is slightly better than that with early update. Like miss ratio, response time does not show improvement with Websearch3 workload shown in Figure 6.21 and 6.22.

6.7.6 Summary

The lazy update policy improvement over early update policy for OpenMail, Websearch3, and synthetic workloads is presented in Figure 6.23, Figure 6.24, Figure 6.25, Figure 6.26, Figure 6.27, and Figure 6.28 respectively.

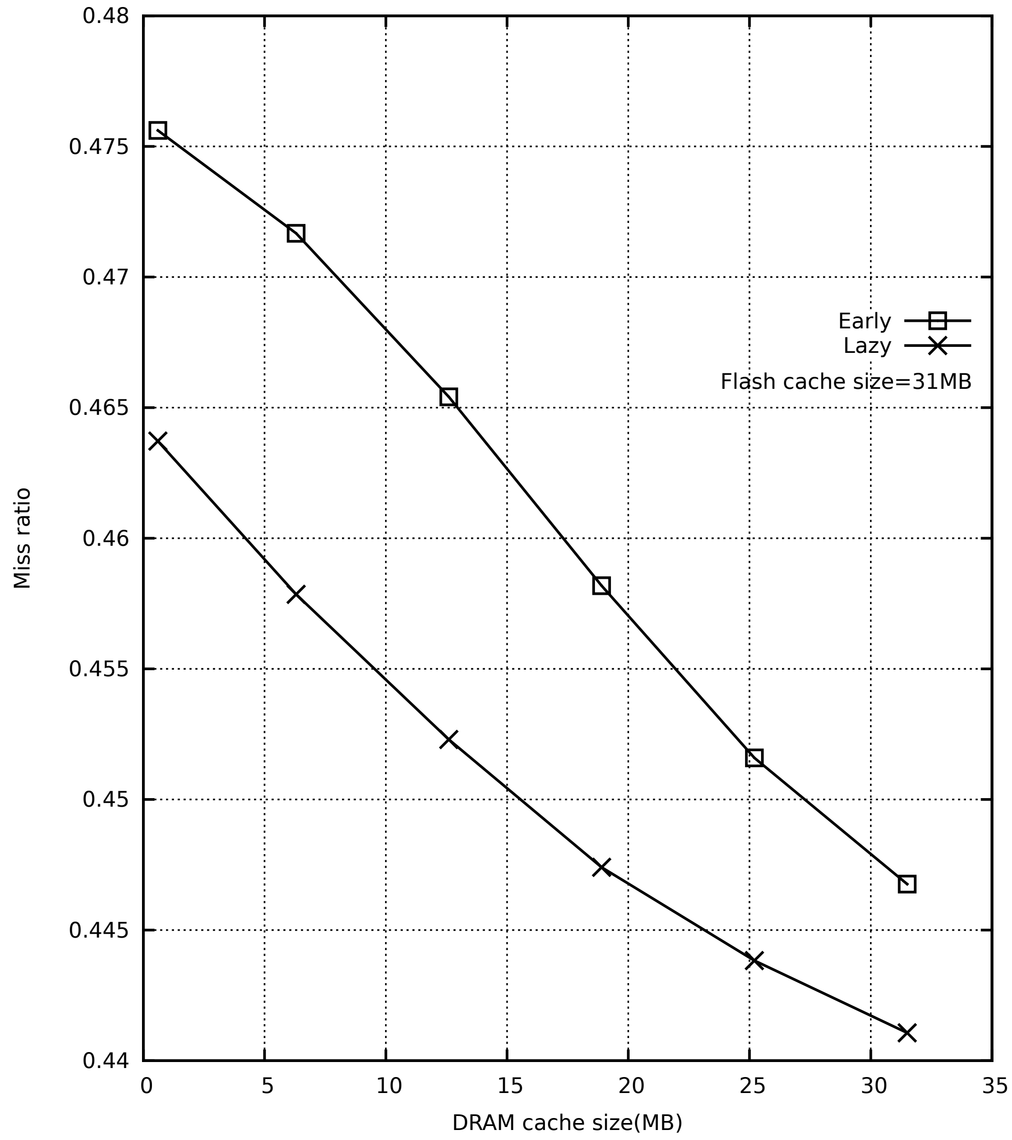


Figure 6.11: Miss Ratio for OpenMail

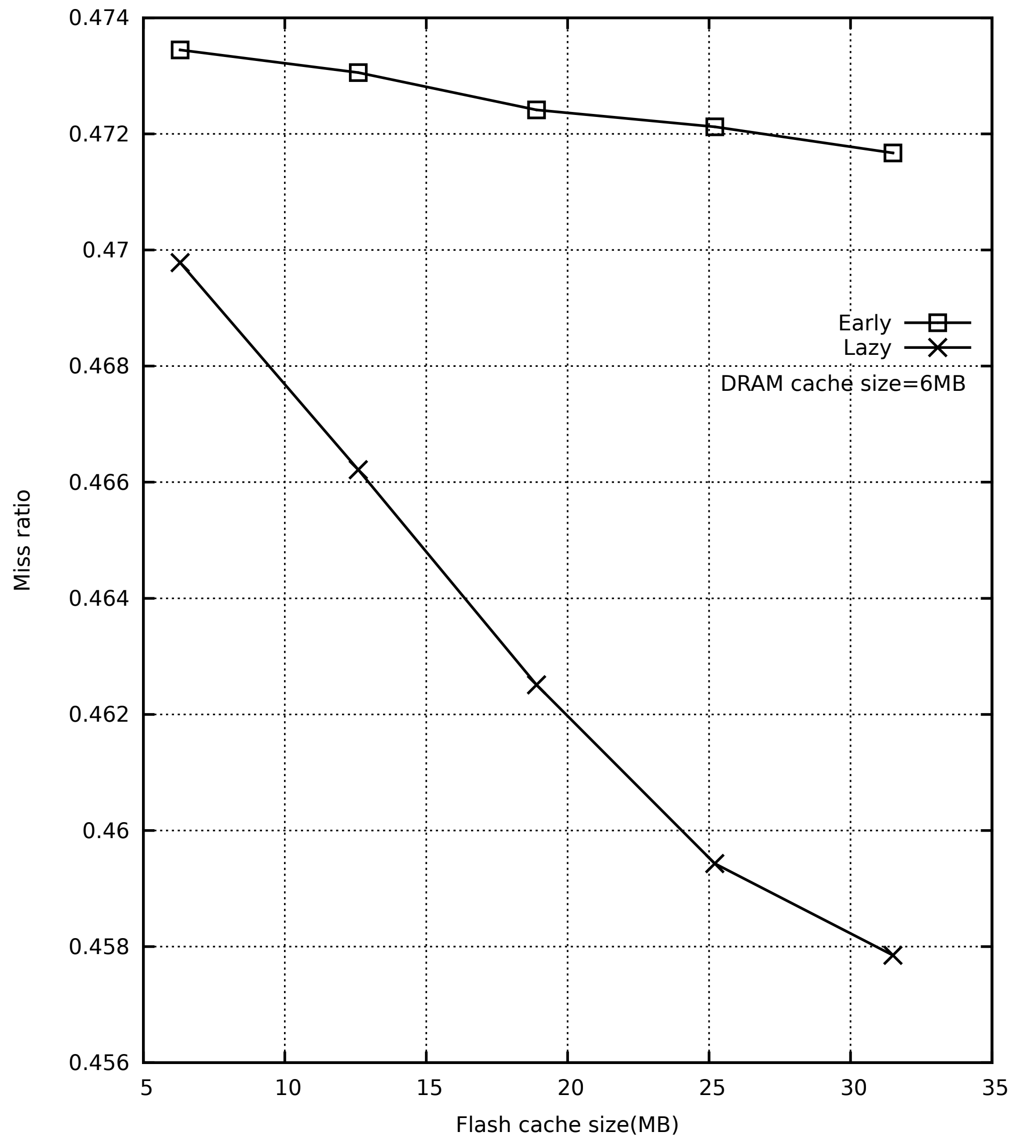


Figure 6.12: Miss Ratio for OpenMail

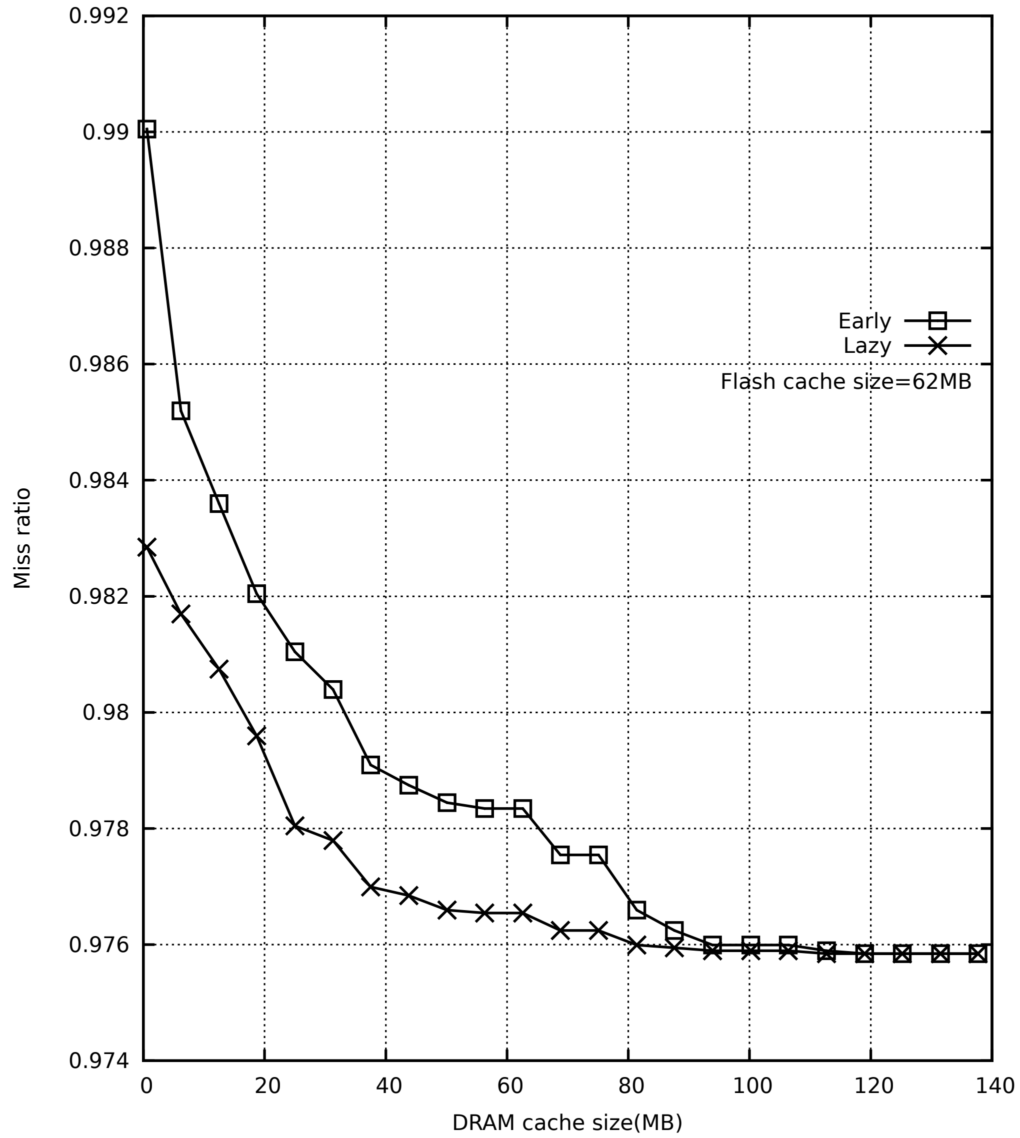


Figure 6.13: Miss Ratio for Synthetic workload

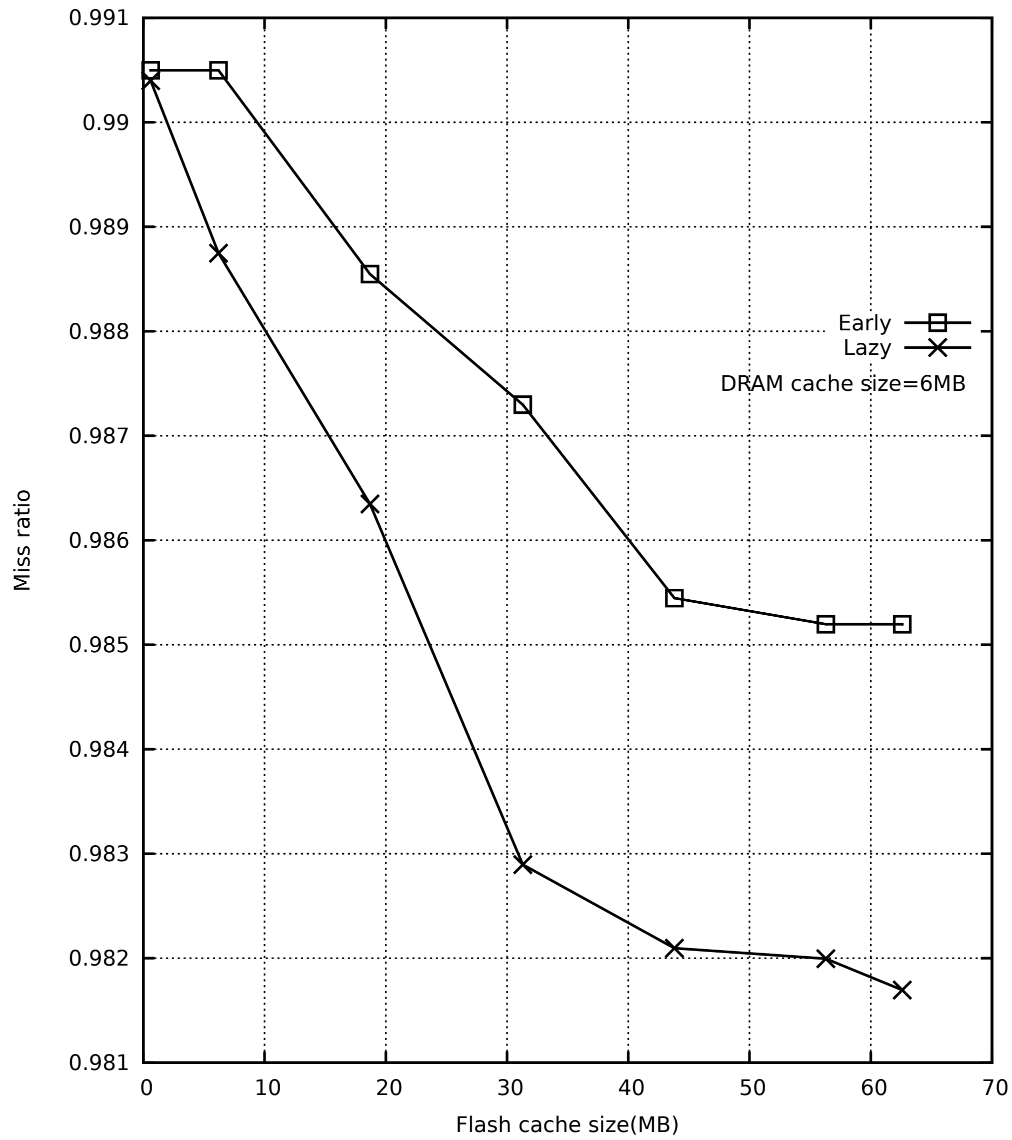


Figure 6.14: Miss Ratio for Synthetic workload

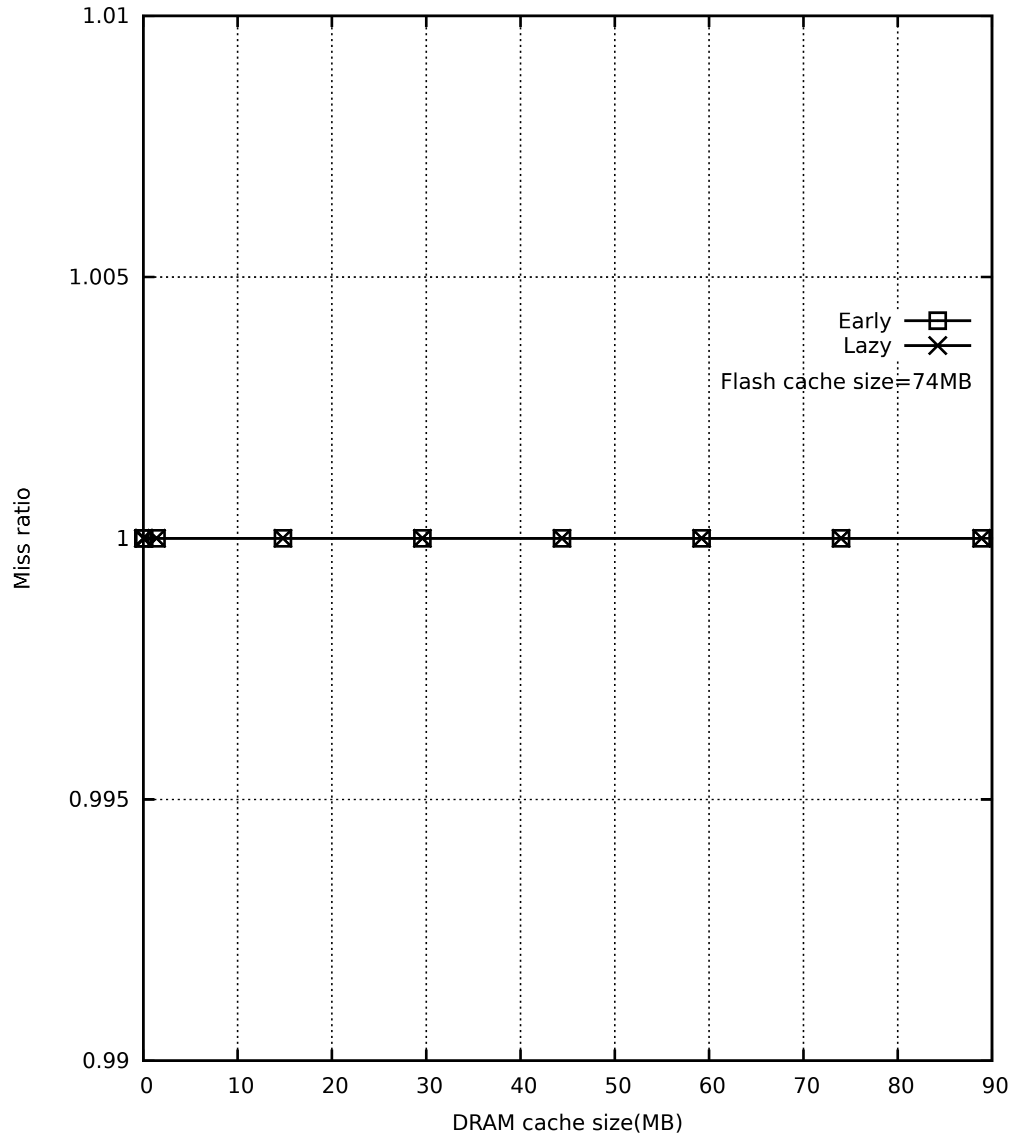


Figure 6.15: Miss Ratio for Websearch3

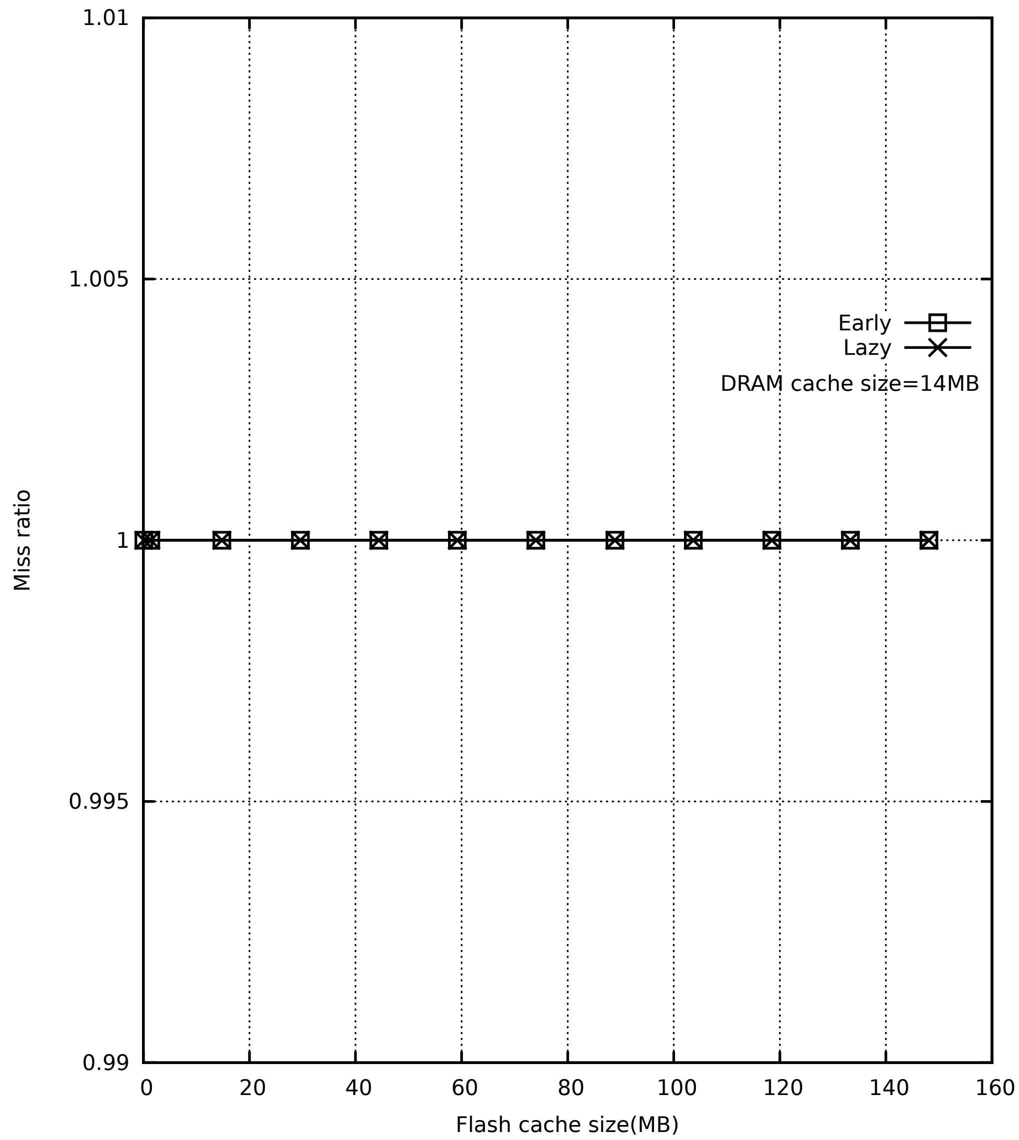


Figure 6.16: Miss Ratio for Websearch3

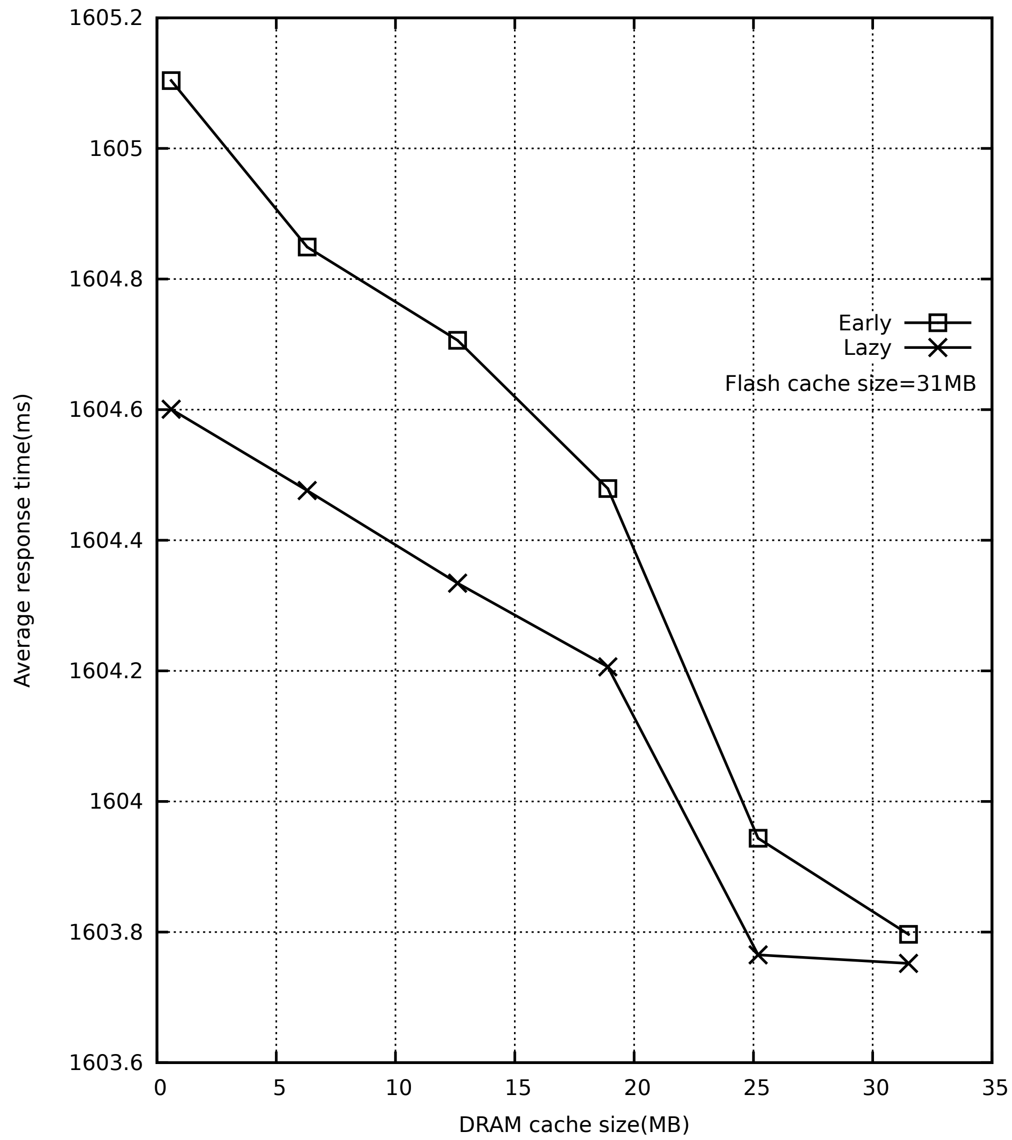


Figure 6.17: Response Time for OpenMail

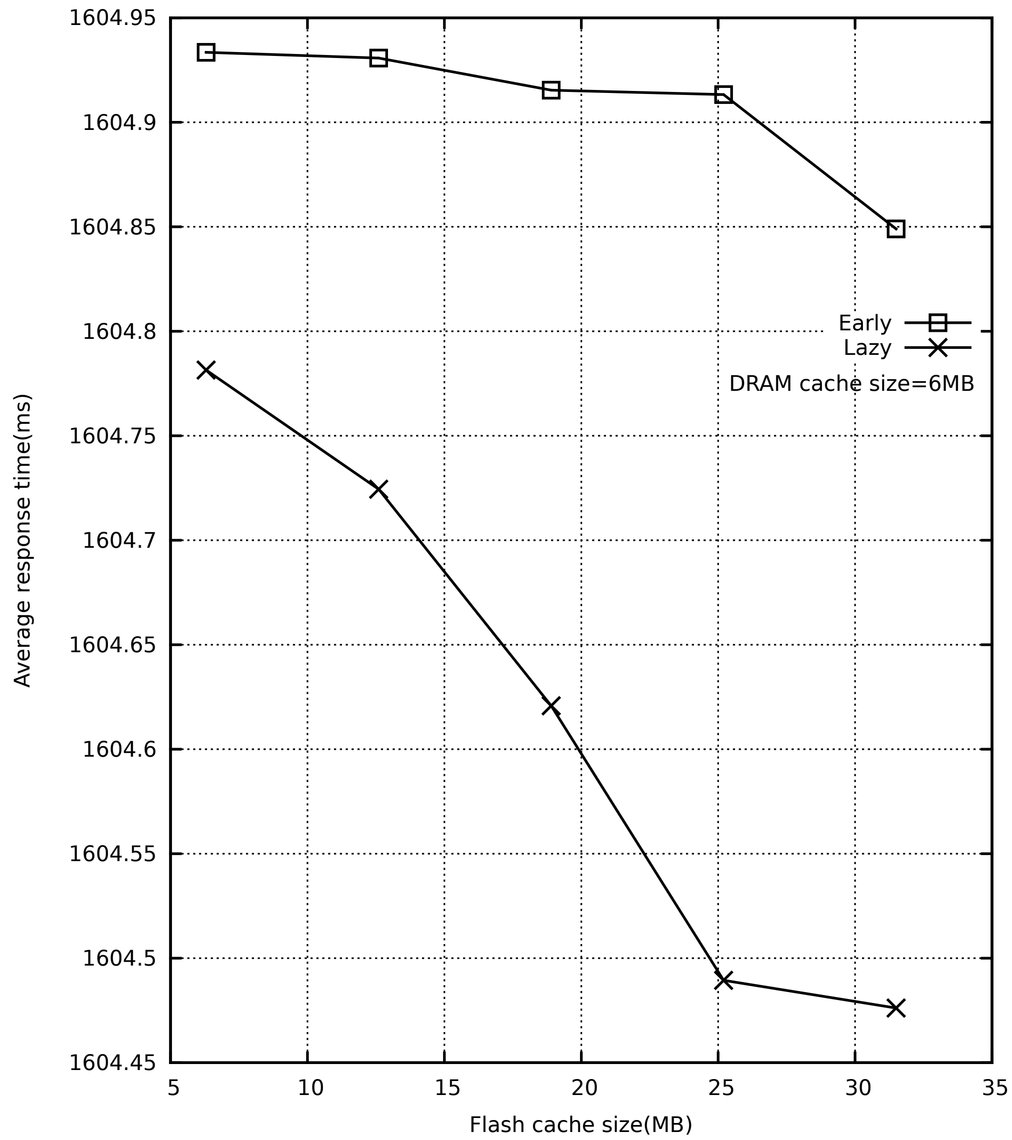


Figure 6.18: Response Time for OpenMail

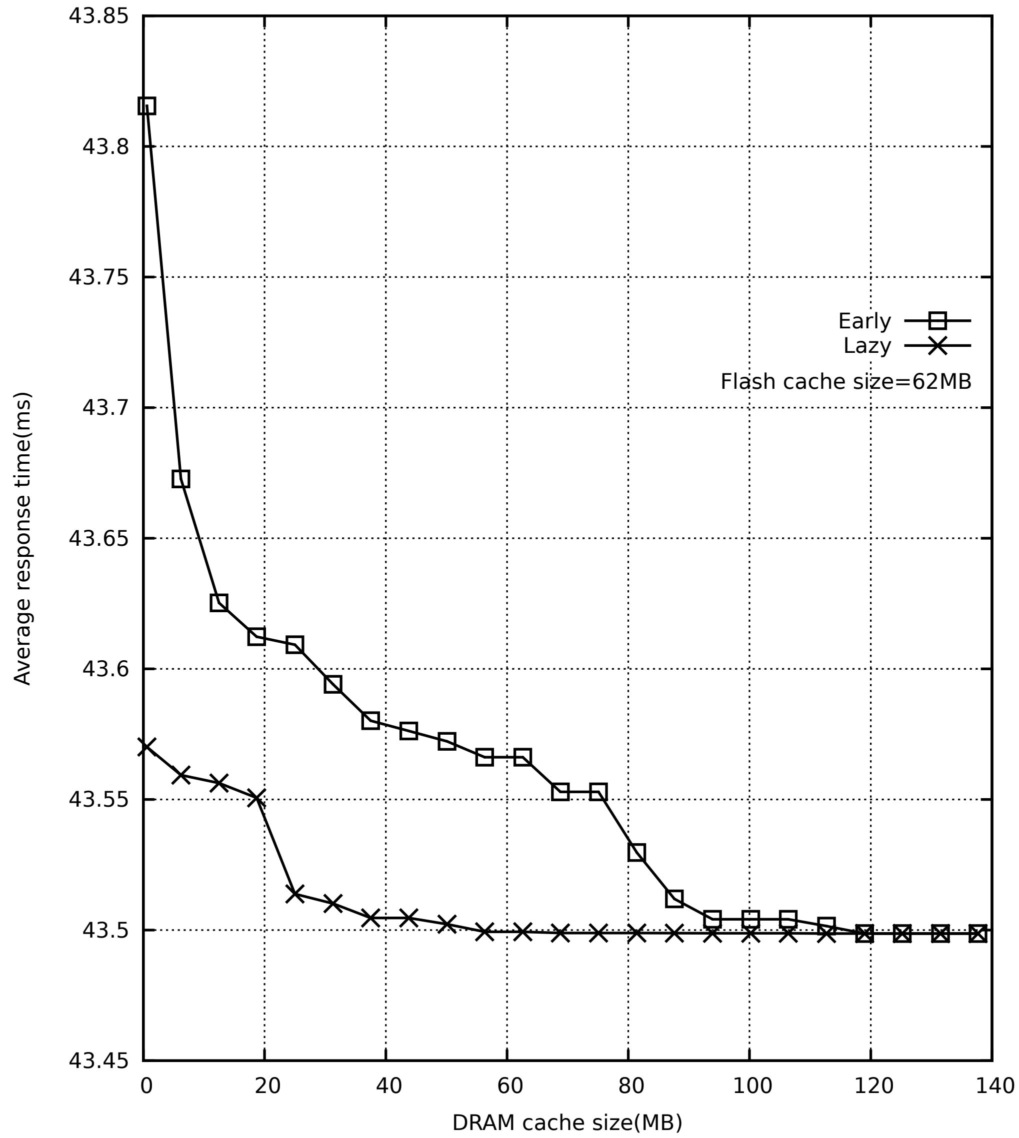


Figure 6.19: Response Time for Synthetic workload

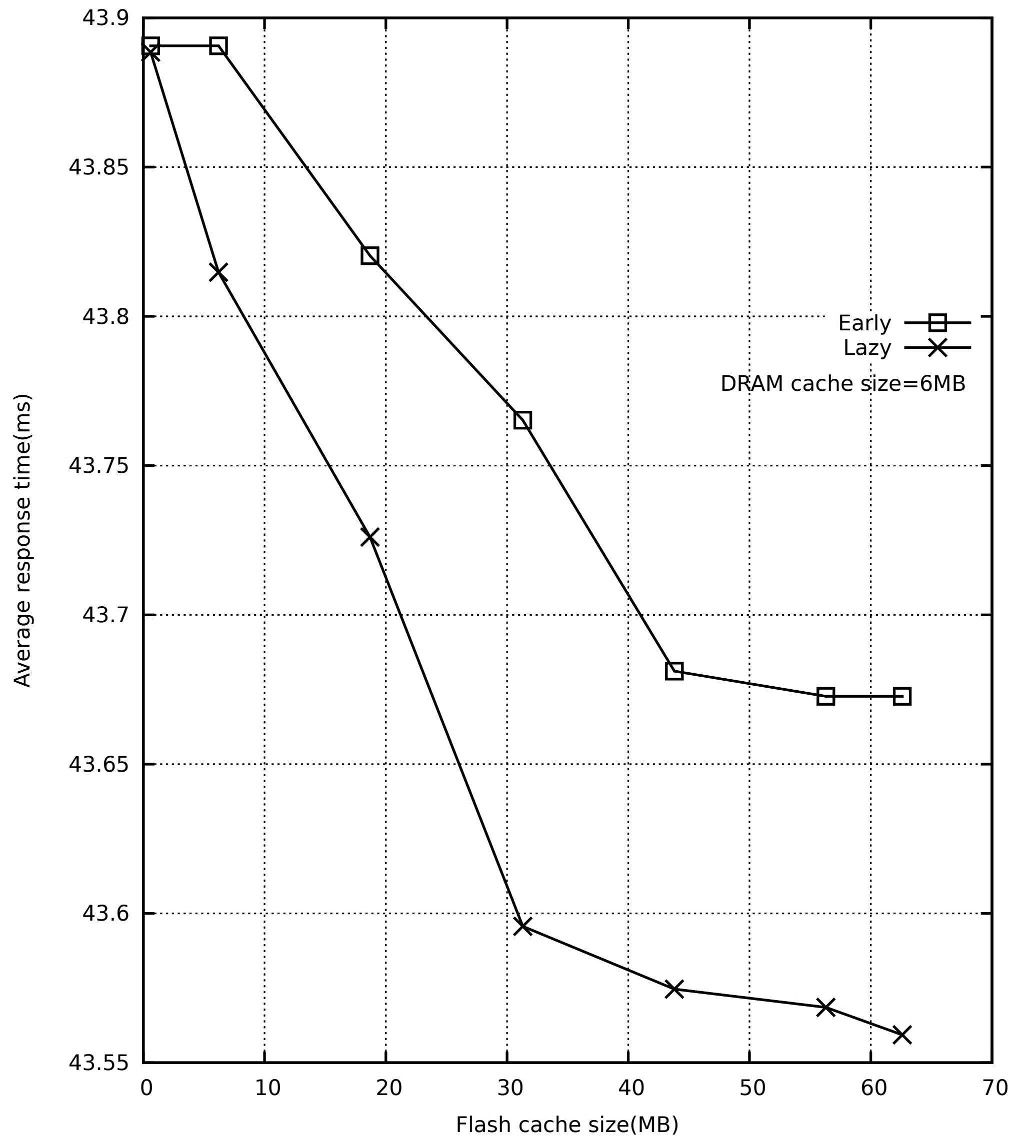


Figure 6.20: Response Time for Synthetic workload

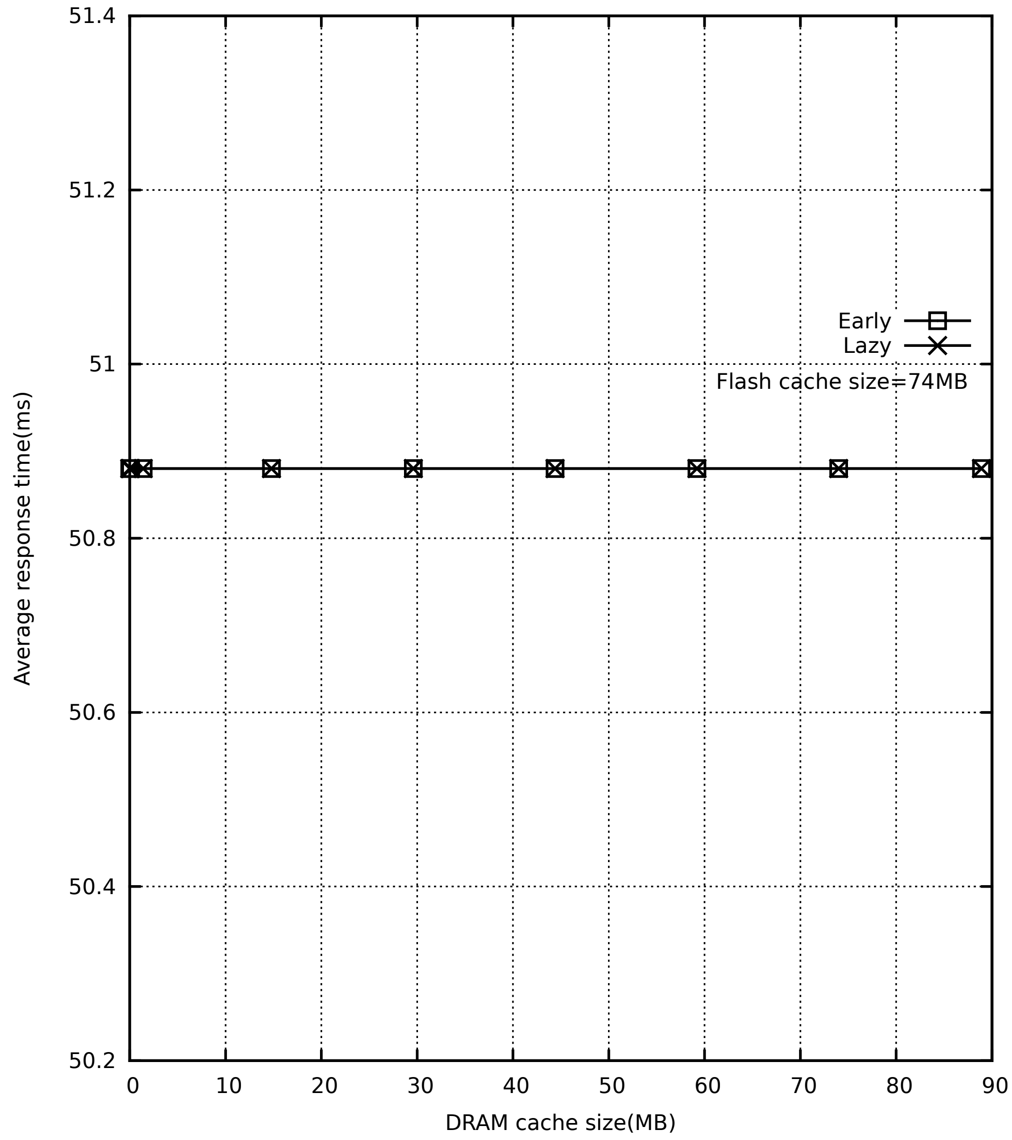


Figure 6.21: Response Time for Websearch3

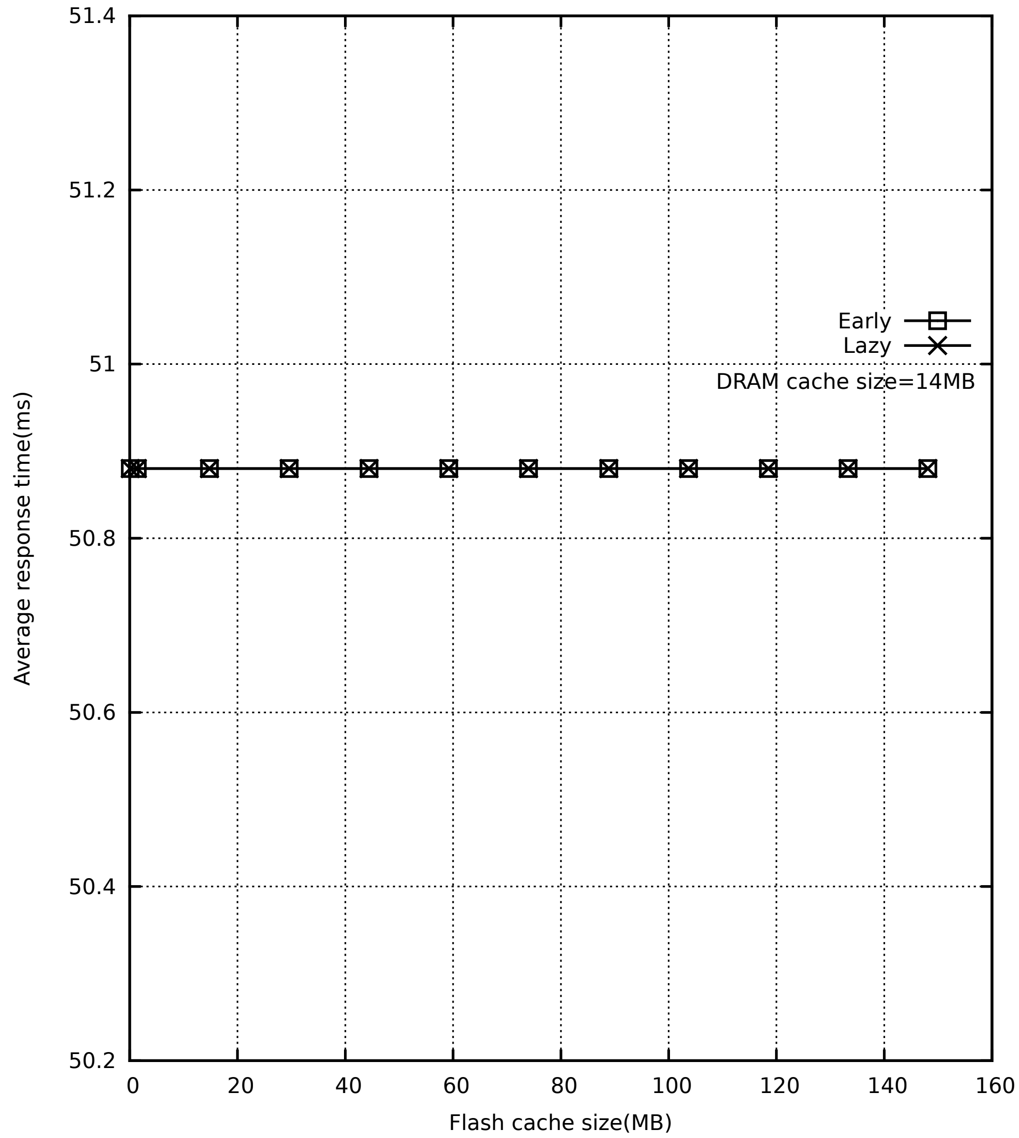


Figure 6.22: Response Time for Websearch3

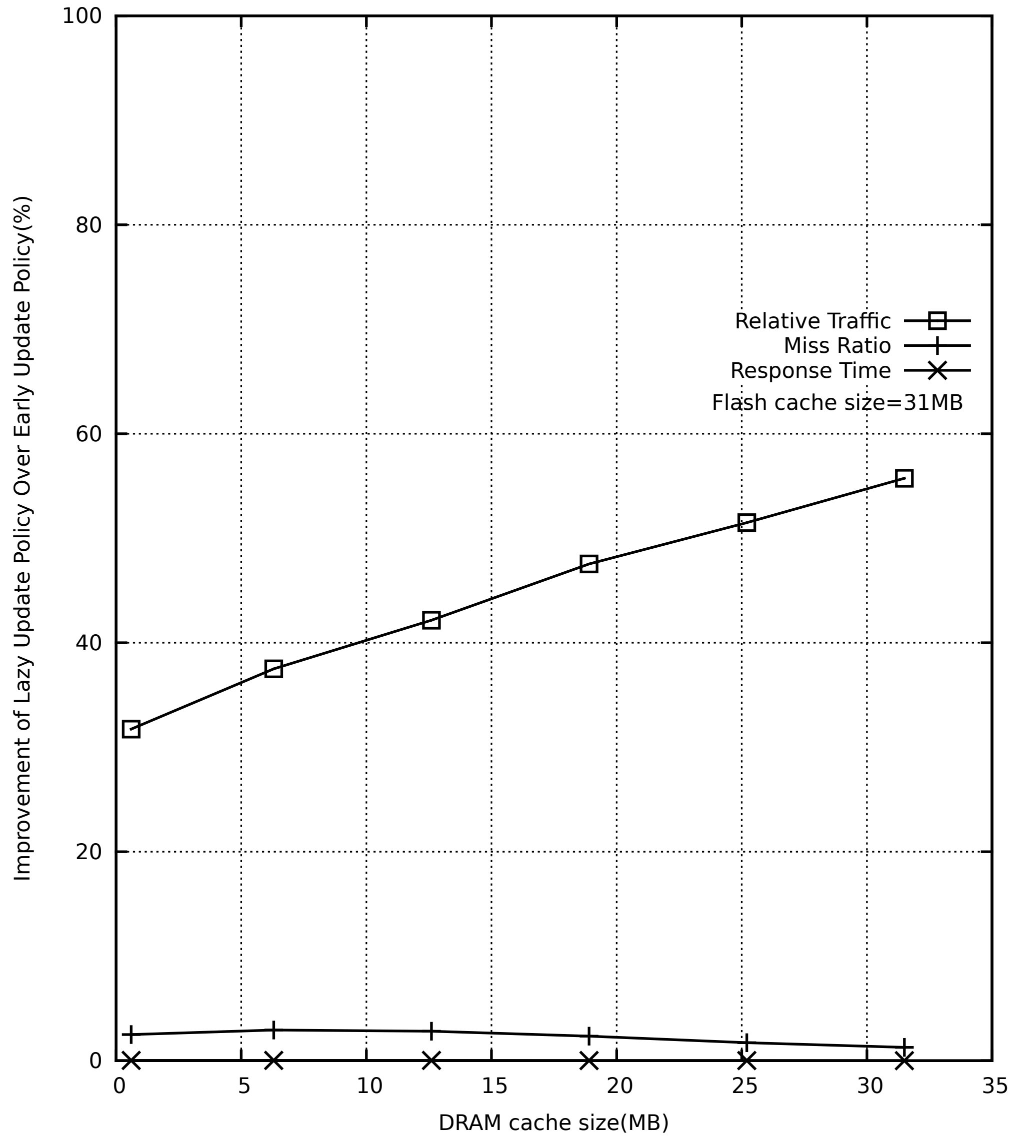


Figure 6.23: Improvement of lazy update policy over early update policy for OpenMail

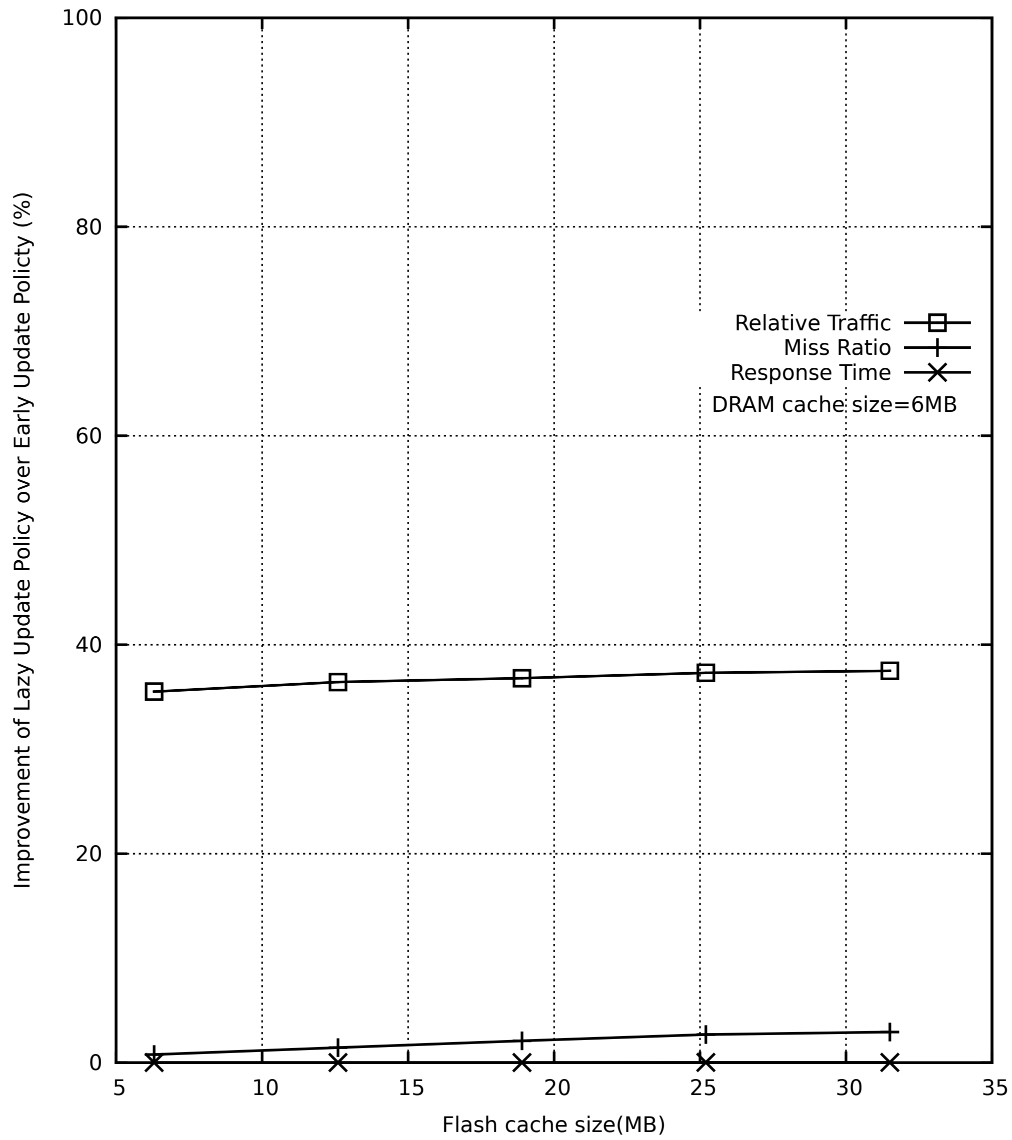


Figure 6.24: Improvement of lazy update policy over early update policy for OpenMail

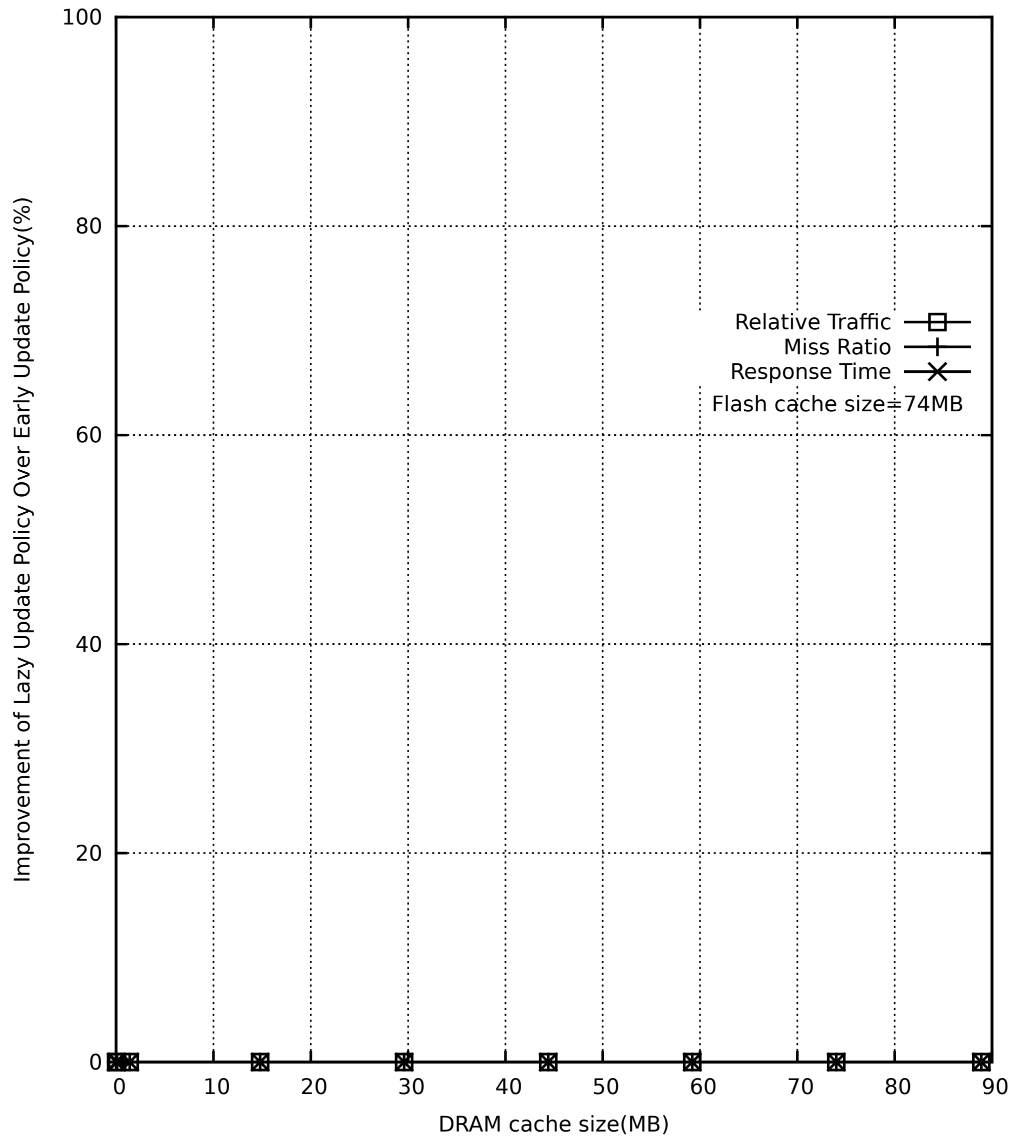


Figure 6.25: Improvement of lazy update policy over early update policy for UMTR(websearch3)

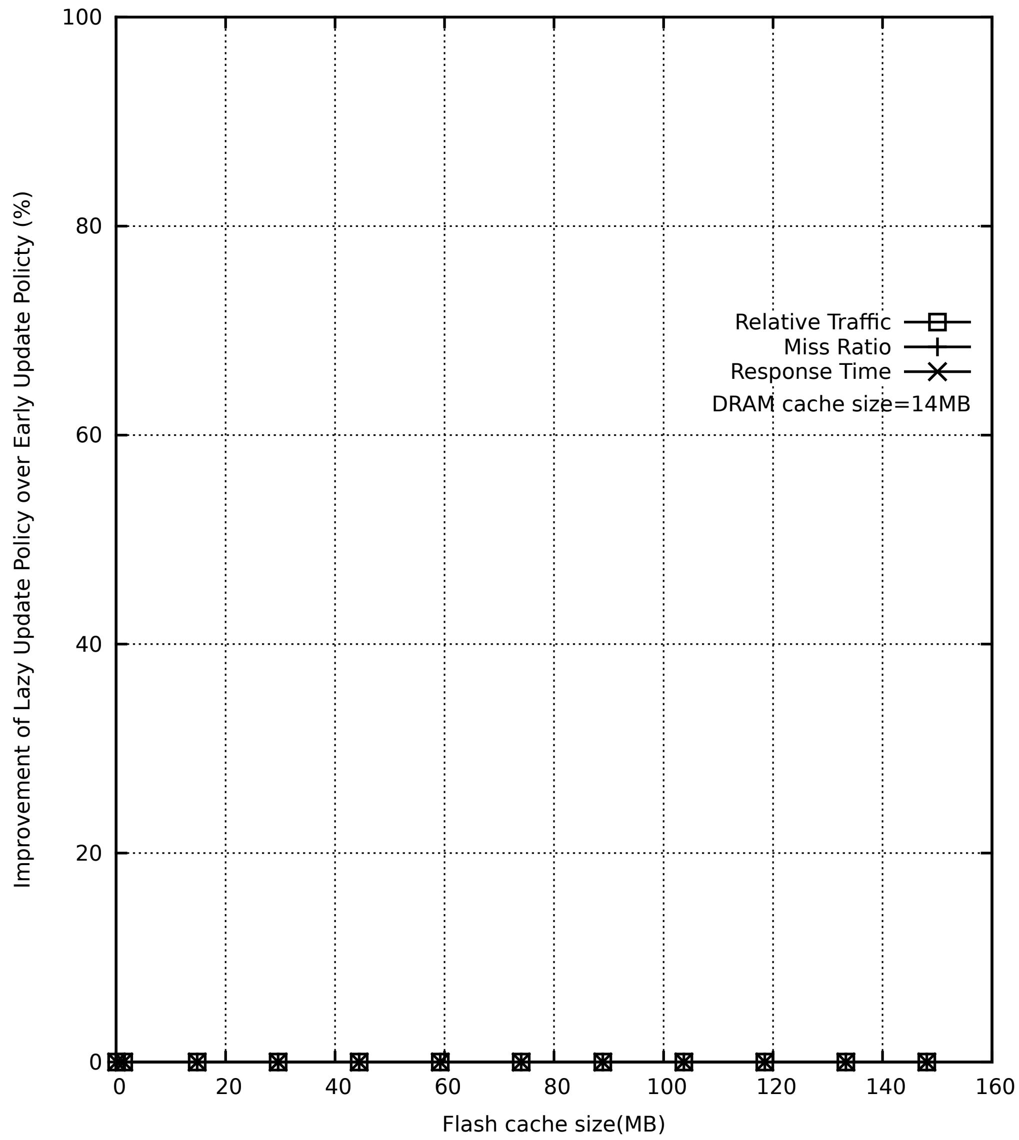


Figure 6.26: Improvement of lazy update policy over early update policy for UMTR(websearch3)

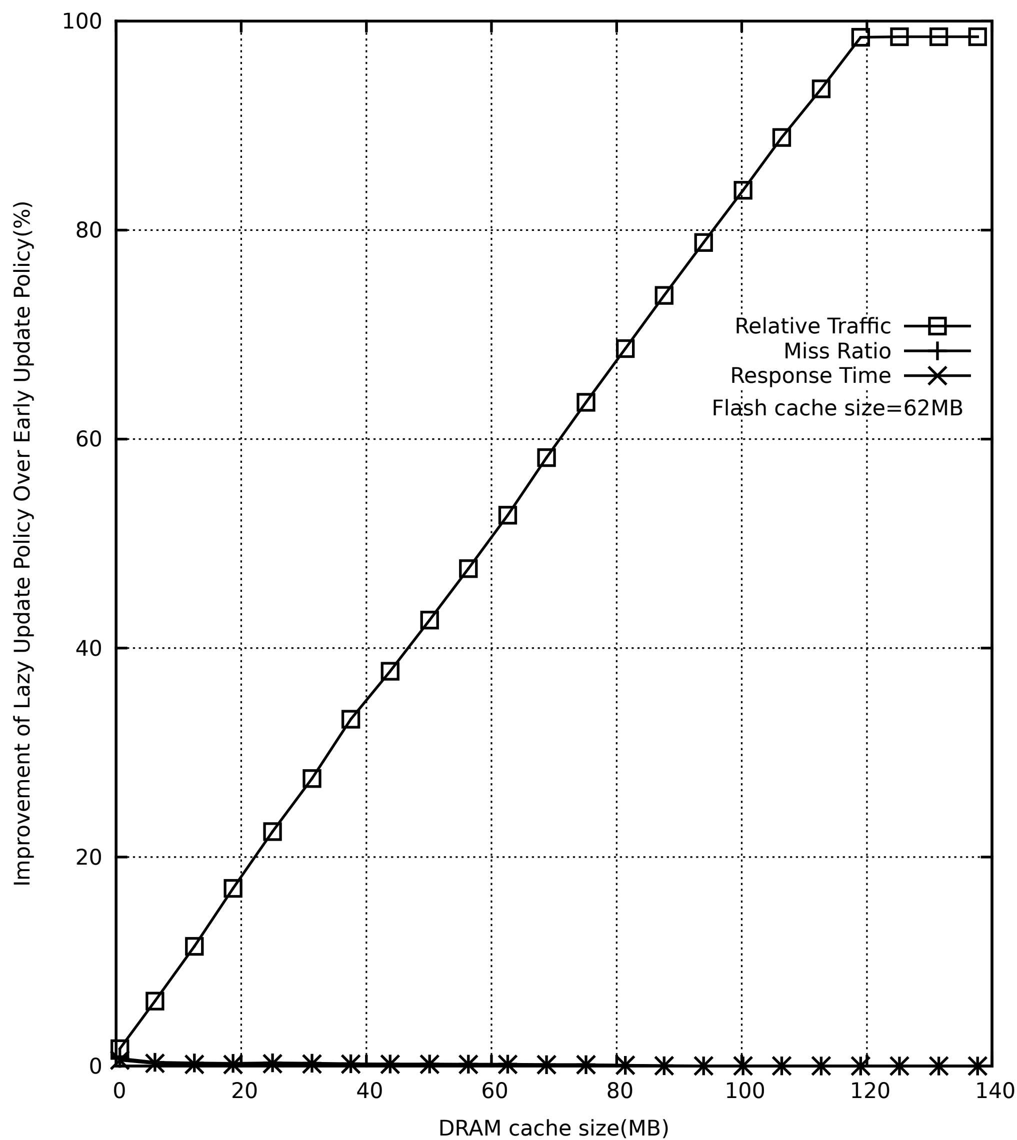


Figure 6.27: Improvement of lazy update policy over early update policy for synthetic workload

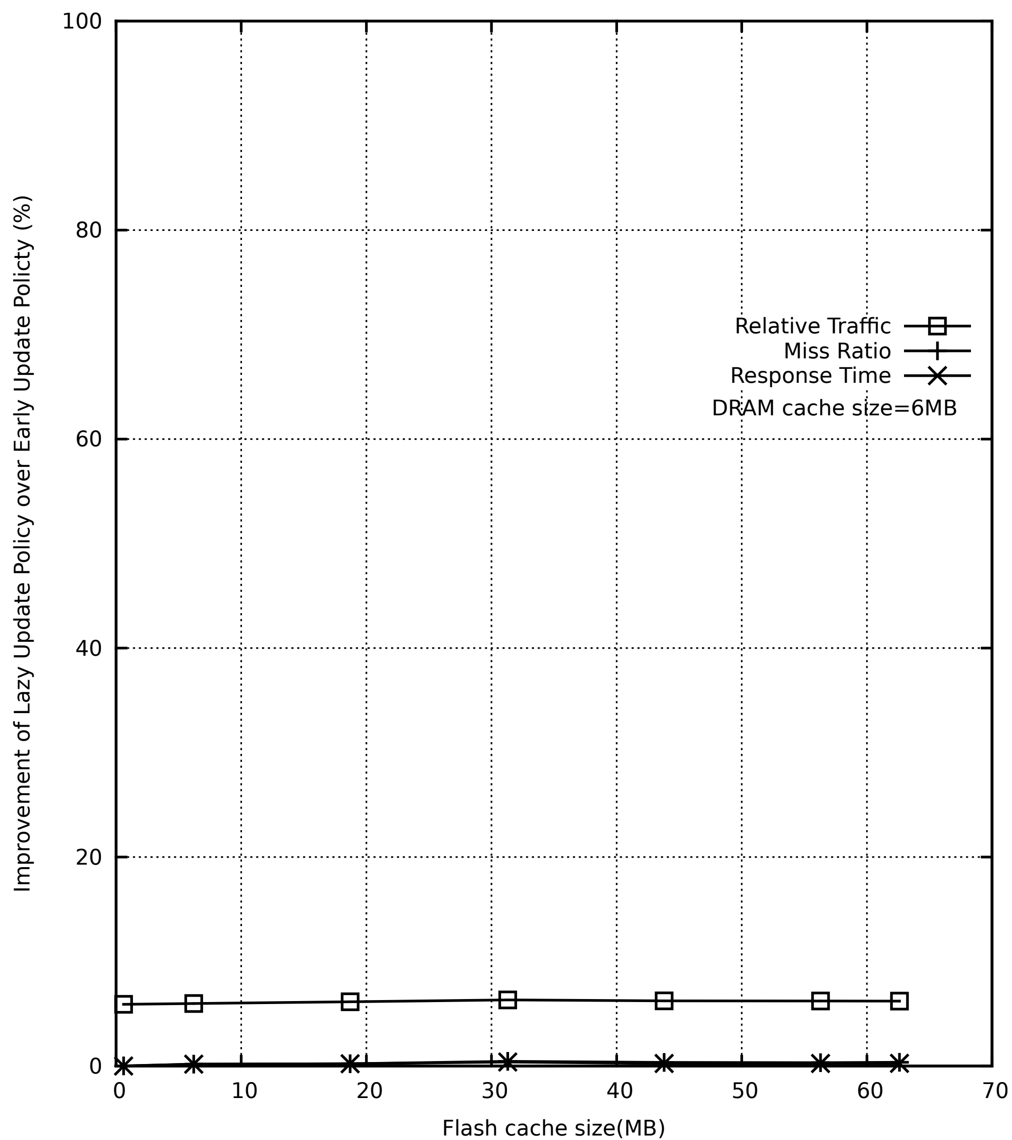


Figure 6.28: Improvement of lazy update policy over early update policy for synthetic workload

Although we see no big difference in terms of miss ratio and average response time with lazy update policy, the write traffic is significantly reduced (between 5-97% depending upon write content of traffic).

Chapter 7

Traffic Savings for Flash as a Major Storage Medium

In this chapter, we focus our research on flash used as a primary store instead of a victim disk cache, whose architecture was shown in Figure 3.2. Unlike a disk cache, read requests would not entail write traffic to flash. Moreover, as a major storage medium, we assume the flash size is much larger than a disk cache. We still concentrated on the impact that update policies have on the lifetime of flash. Compared with the early update policy, our study shows that using flash as a victim device (which corresponds to lazy update policy) can save a lot of lifetime. At the same time, the performance in terms of response time is improving as well, which is consistent with the findings in Chapter 6.

The rest of the chapter is organized as follows: Section 7.1 introduces early update for reads. In Section 7.2, we discuss early update policy for writes. In Section 7.3, we talk about lazy update policy for reads. In Section 7.4, we present lazy update for writes. Section 7.5 shows the simulation parameters. Section 7.6 talks about performance metrics used in this chapter. Finally, Section 7.7 presents the experimental results.

7.1 Early Update for reads

The flowchart of early update for reads is shown in Figure 7.1. When a read request arrives, DRAM cache is first checked. If found, then the request is satisfied. Otherwise, data is read from flash. Meanwhile, DRAM cache is updated upon receiving data from flash.

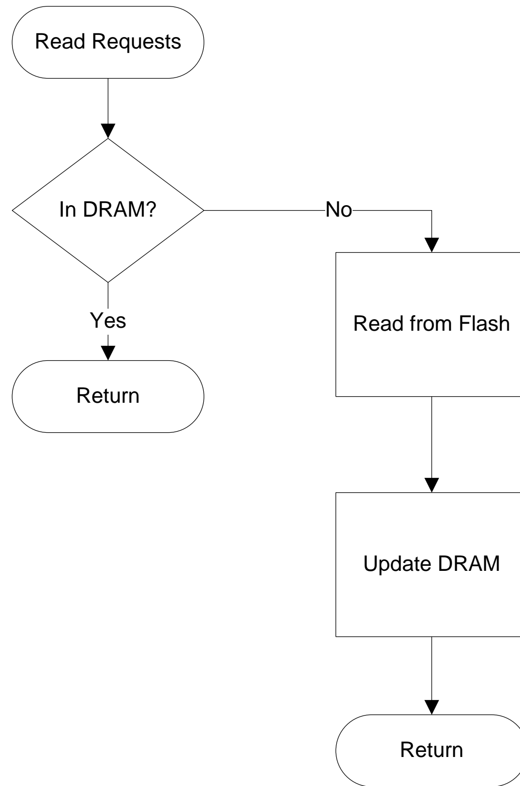


Figure 7.1: Early update for reads

7.2 Early Update for writes

The flowchart of early update for write is shown in Figure 7.2. When a write request comes, search in DRAM cache first. If found, merge the data. Otherwise, DRAM cache is updated. Next, update flash with Write Counter incremented.

7.3 Lazy Update for reads

The flowchart of lazy update for reads is shown in Figure 7.3. When a read request arrives, search in DRAM cache. If found, return the data. If not, read the data from flash. Next, update DRAM cache with the new data. If a dirty data entry is evicted from DRAM cache, update flash with the Write Counter incremented.

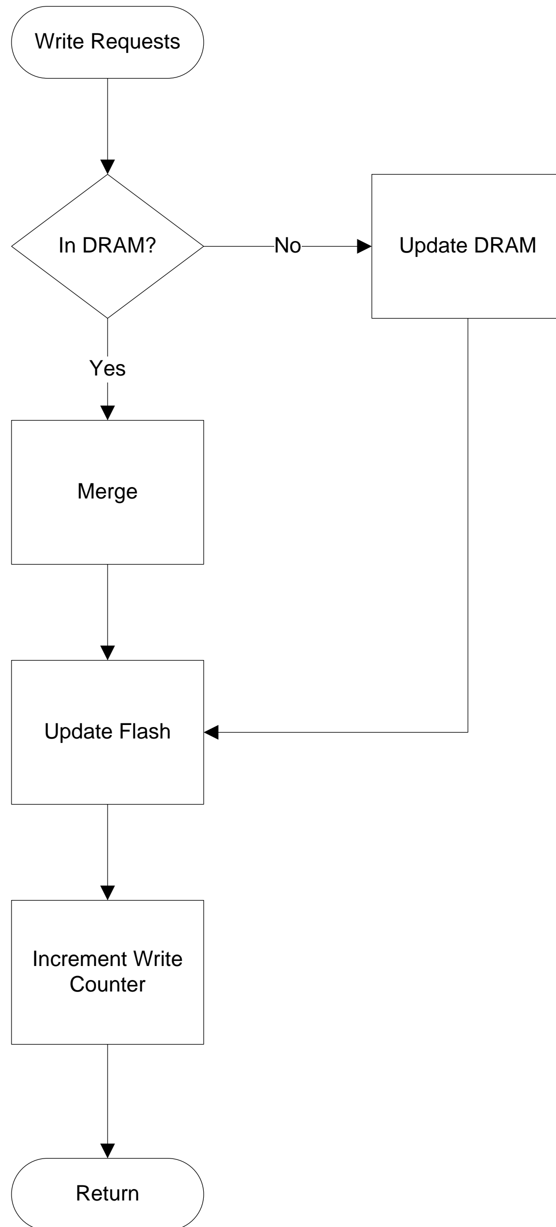


Figure 7.2: Early update for writes

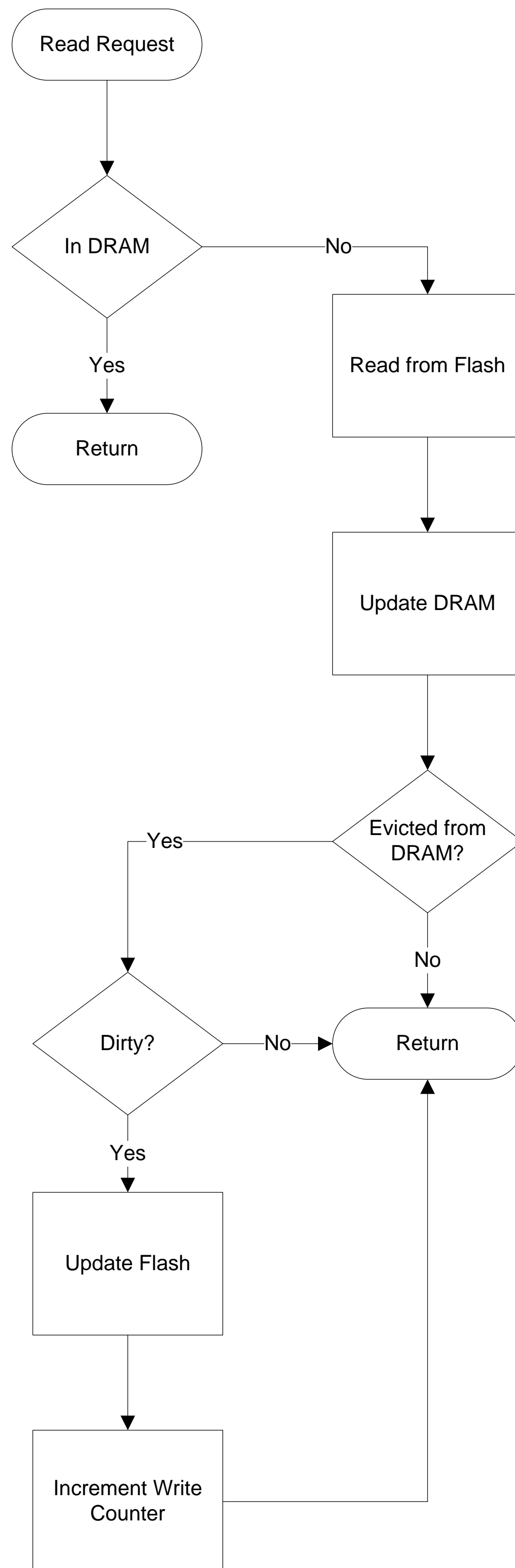


Figure 7.3: Lazy update for reads

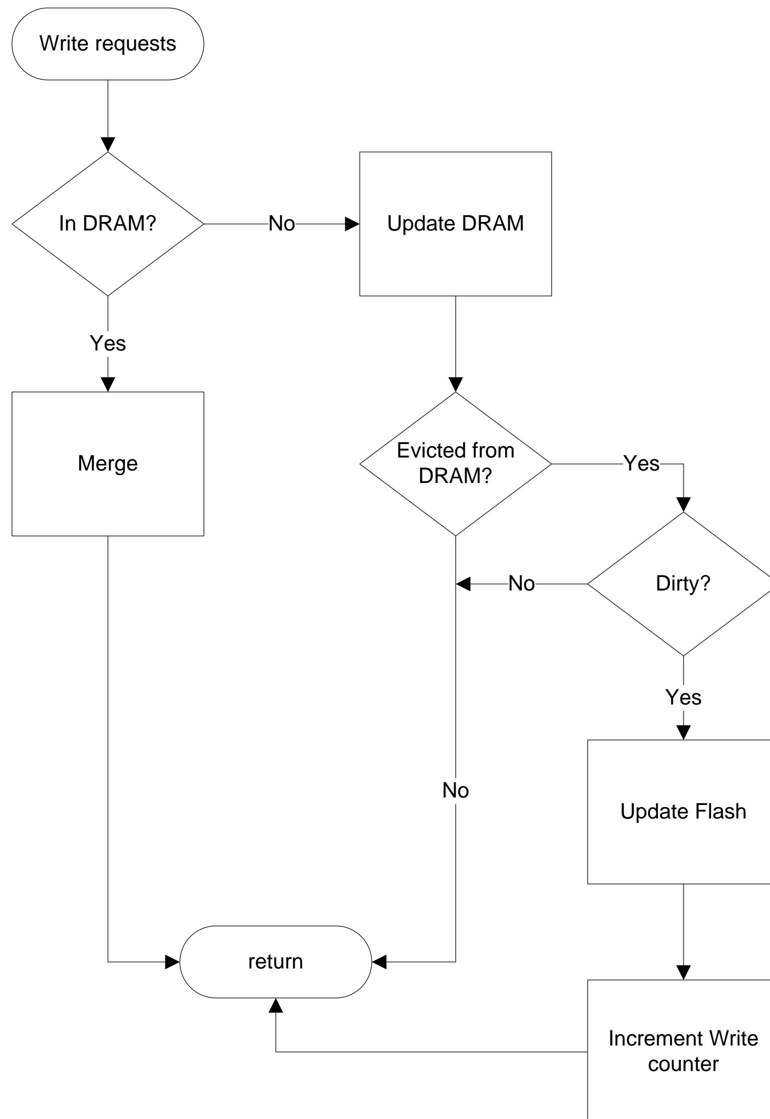


Figure 7.4: Lazy update for writes

7.4 Lazy Update writes

The flowchart of lazy update for writes is shown in Figure 7.4. When a write request arrives, search in DRAM cache. If found, merge the data. Otherwise, DRAM cache is updated. If dirty data is evicted from DRAM cache, update the flash with the Write Counter incremented.

7.5 Simulation Parameters

The part of the parameter file(ssd-victim.parv) we used in running DiskSim is listed below.

```
ssdmodel_ssd SSD {
        # vp - this is a percentage of total pages in the ssd
        Reserve pages percentage = 15,

        # vp - min percentage of free blocks needed. if the free
        # blocks drop below this, cleaning kicks in
        Minimum free blocks percentage = 5,

        # vp - a simple read-modify-erase-write policy = 1 (no longer supported)
        # vp - osr write policy = 2
        Write policy = 2,

        # vp - random = 1 (not supp), greedy = 2, wear-aware = 3
        Cleaning policy = 2,

        # vp - number of planes in each flash package (element)
        Planes per package = 8,

        # vp - number of flash blocks in each plane
        Blocks per plane = 2048,

        # vp - how the blocks within an element are mapped on a plane
        # simple concatenation = 1, plane-pair stripping = 2 (not tested),
        # full stripping = 3
        Plane block mapping = 3,

        # vp - copy-back enabled (1) or not (0)
        Copy back = 1,

        # how many parallel units are there?
        # entire elem = 1, two dies = 2, four plane-pairs = 4
        Number of parallel units = 1,

        # vp - we use diff allocation logic: chip/plane
        # each gang = 0, each elem = 1, each plane = 2
        Allocation pool logic = 1,

        # elements are grouped into a gang
        Elements per gang = 1,

        # shared bus (1) or shared control (2) gang
        Gang share = 1,
```

```
# when do we want to do the cleaning?
Cleaning in background = 0,

Command overhead = 0.00,
Bus transaction latency = 0.0,

# Assuming PCI-E, with 8 lanes with 8b/10b encoding.
# This gives 2.0 Gbps per lane and with 8 lanes we get about
# 2.0 GBps. So, bulk sector transfer time is about 0.238 us.
# Use the "Read block transfer time" and "Write block transfer time"
# from disksim_bus above.
Bulk sector transfer time = 0,

Flash chip elements = 8,

Page size = 8,

Pages per block = 64,

# vp - changing the no of blocks from 16184 to 16384
Blocks per element = 16384,

Element stride pages = 1,

Never disconnect = 1,
Print stats = 1,
Max queue length = 20,
Scheduler = disksim_ioqueue {
    Scheduling policy = 1,
    Cylinder mapping strategy = 0,
    Write initiation delay = 0,
    Read initiation delay = 0.0,
    Sequential stream scheme = 0,
    Maximum concat size = 0,
    Overlapping request scheme = 0,
    Sequential stream diff maximum = 0,
    Scheduling timeout scheme = 0,
    Timeout time/weight = 0,
    Timeout scheduling = 0,
    Scheduling priority scheme = 0,
    Priority scheduling = 1
},
Timing model = 1,

# vp changing the Chip xfer latency from per sector to per byte
Chip xfer latency = 0.000025,

Page read latency = 0.025,
Page write latency = 0.200,
```

```
        Block erase latency = 1.5
} # end of SSD spec
```

In addition, we change the following parameters via command line parameter interface:

- DRAM cache size
- Devno

7.6 Performance Metrics

As in Chapter 6, we use *relative traffic* and *response time* as performance metrics. However, we will not use *miss ratio* in this chapter since flash is not used as disk cache.

7.7 Experimental Results

We ran OpenMail, Synthetic, and Websearch3 workload traces against early update policy and lazy update policy, during which relative traffic and average response time were observed. We especially watched the impact of DRAM size on relative traffic savings. The simulation results are shown in Figure 7.5 through Figure 7.13. Several observations can be made from these figures.

7.7.1 Write Traffic Savings

OpenMail (Figure 7.5): relative traffic with lazy update policy is roughly 33% of that with early update policy when DRAM size reaches 10MB (0.33 for lazy update policy vs. 1 for early update policy).

Synthetic workload (Figure 7.6): relative traffic with lazy update policy is roughly 40% of that with early update policy when DRAM size reaches 10MB (roughly 0.4 for lazy update policy vs. 1 for early update policy).

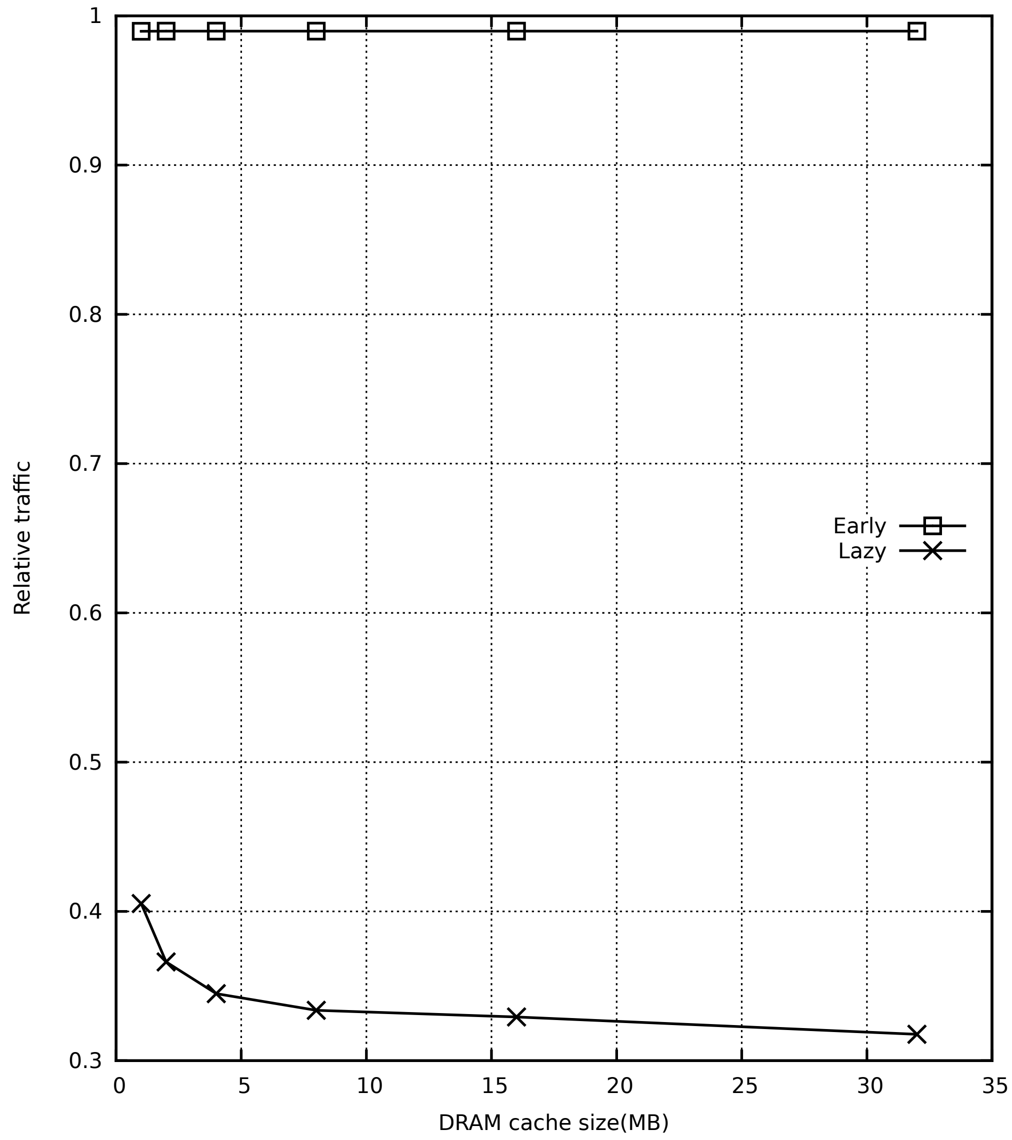


Figure 7.5: Relative traffic for OpenMail

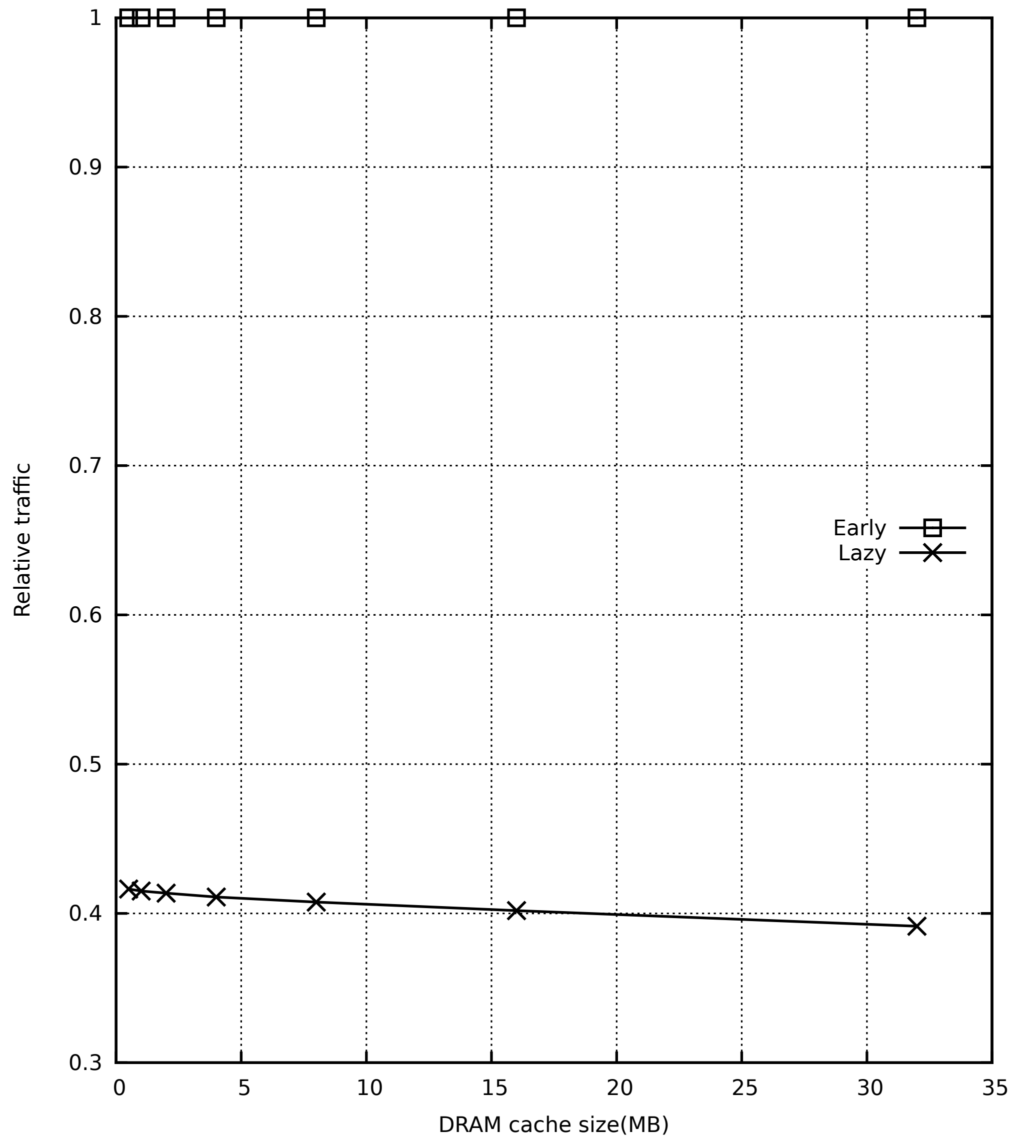


Figure 7.6: Relative traffic for Synthetic workloads

Websearch3 (Figure 7.7): relative traffic does not see improvement with Websearch3 because Websearch3 is a read-only workload. For read-only workloads, there is no overwrite that can be coalesced using lazy update policy.

7.7.2 DRAM Size Needs not to be Large to Be Effective

DRAM size larger than 10MB is effective to reduce write traffic significantly, which is shown in Figure 7.5 and Figure 7.6.

7.7.3 Response Time

OpenMail (Figure 7.8): Response time with lazy update policy is roughly 5.6% of that with early update policy (roughly 0.2 ms for lazy update policy, roughly 3.6 ms for early update policy).

Synthetic workload (Figure 7.9): Response time with lazy update policy is roughly 63.6% of that with early update policy (roughly 0.14 ms for lazy update policy, roughly 0.22 ms for early update policy).

Websearch3 (Figure 7.10): the response time for Websearch3 does not see improvement.

7.7.4 Summary

The lazy update policy improvement over early update policy for OpenMail, Websearch3, and synthetic workloads has been presented in Figure 7.11, Figure 7.12, and Figure 7.13 respectively.

Although we see no difference for read-only workloads, lazy update policy reduces write traffic significantly with read/write workloads. Performance in terms of average response time also sees marked improvement.

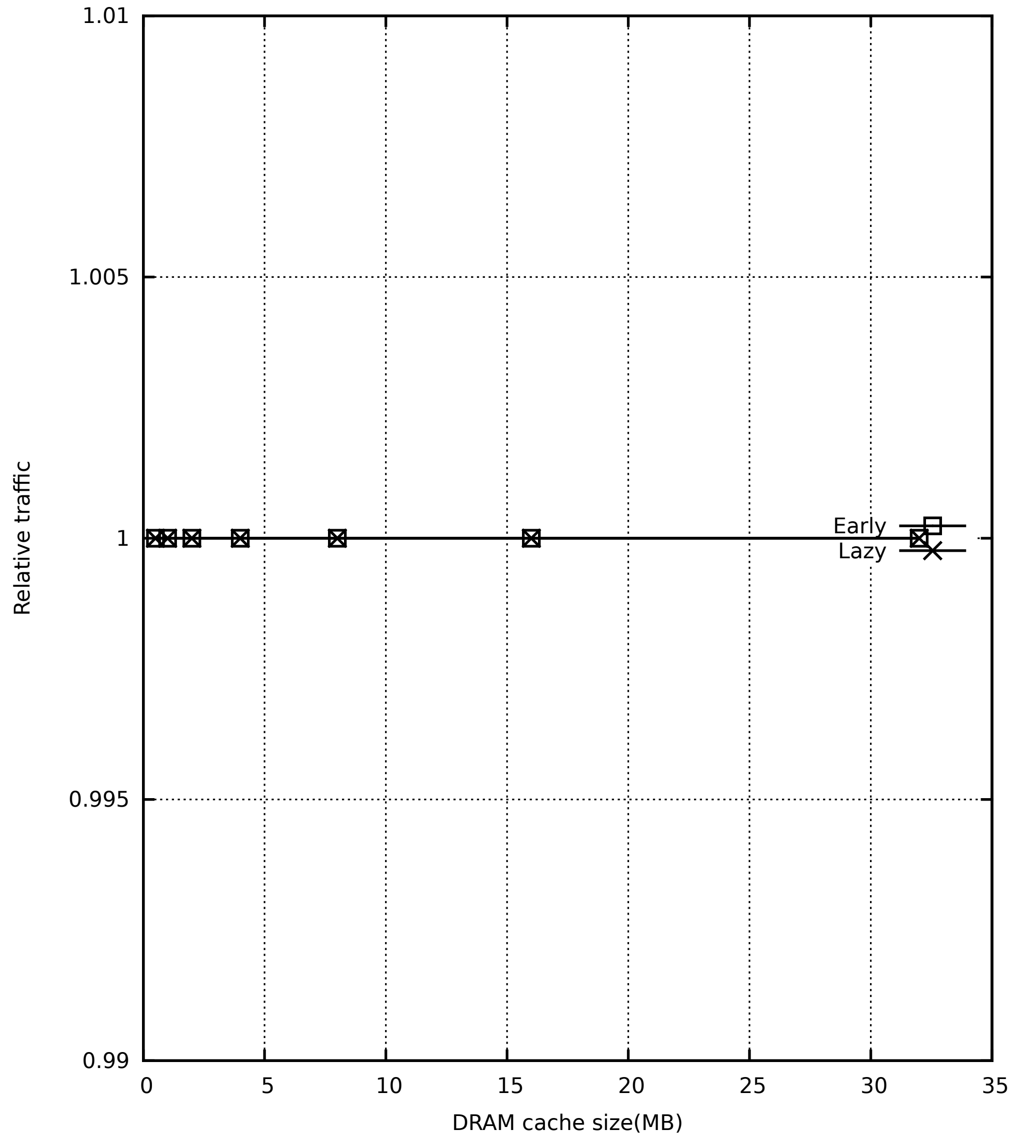


Figure 7.7: Relative traffic for Websearch3

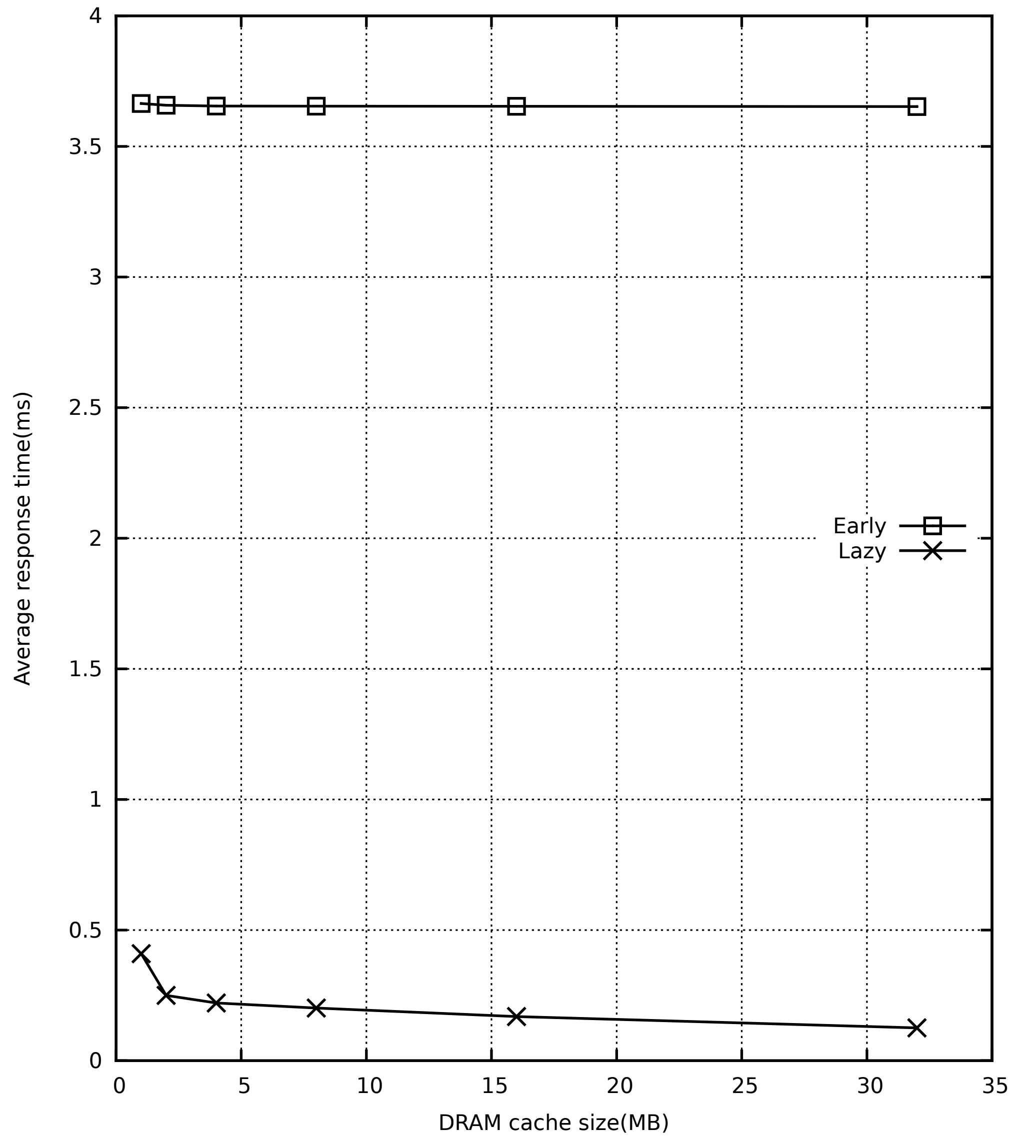


Figure 7.8: Response time for OpenMail

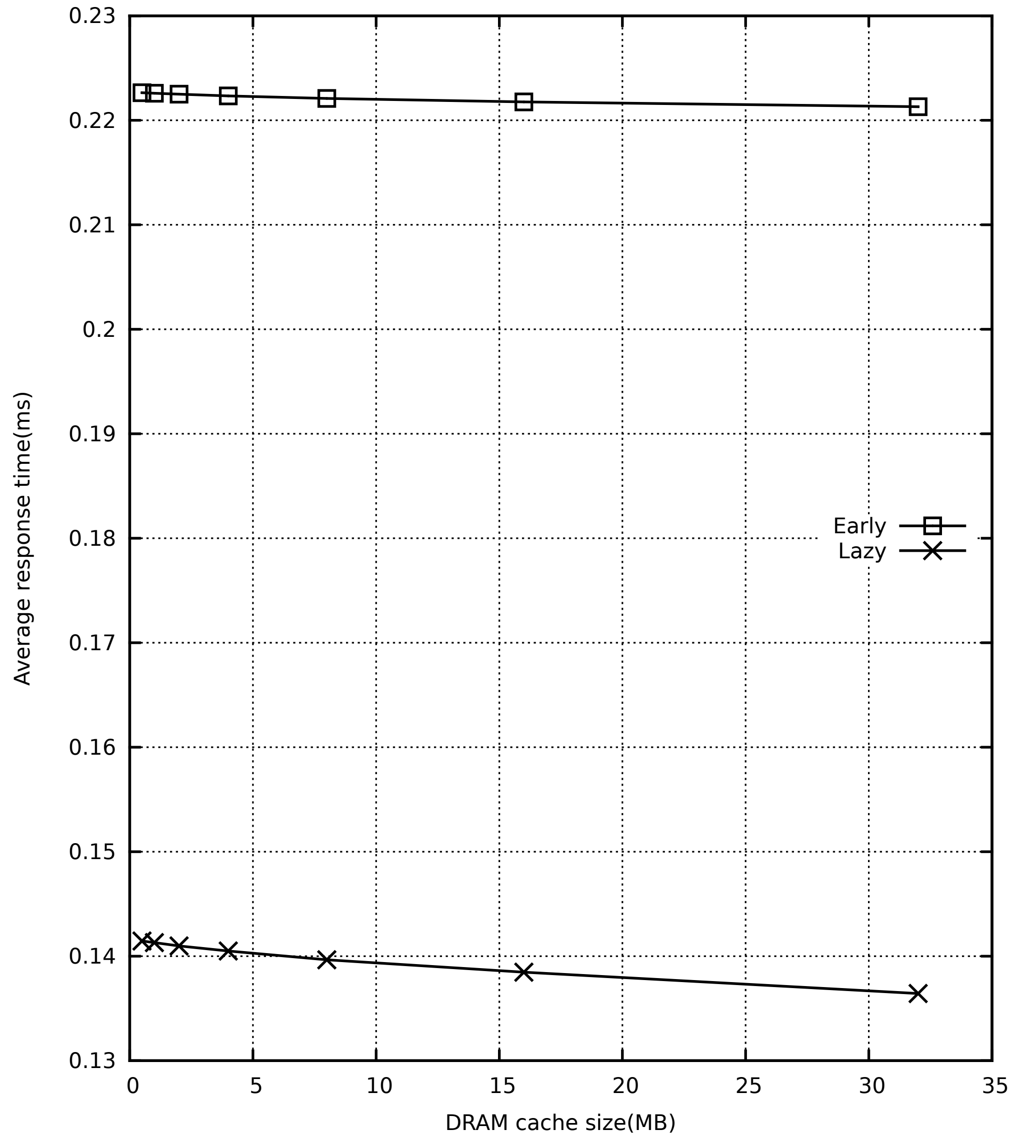


Figure 7.9: Response time for Synthetic workloads

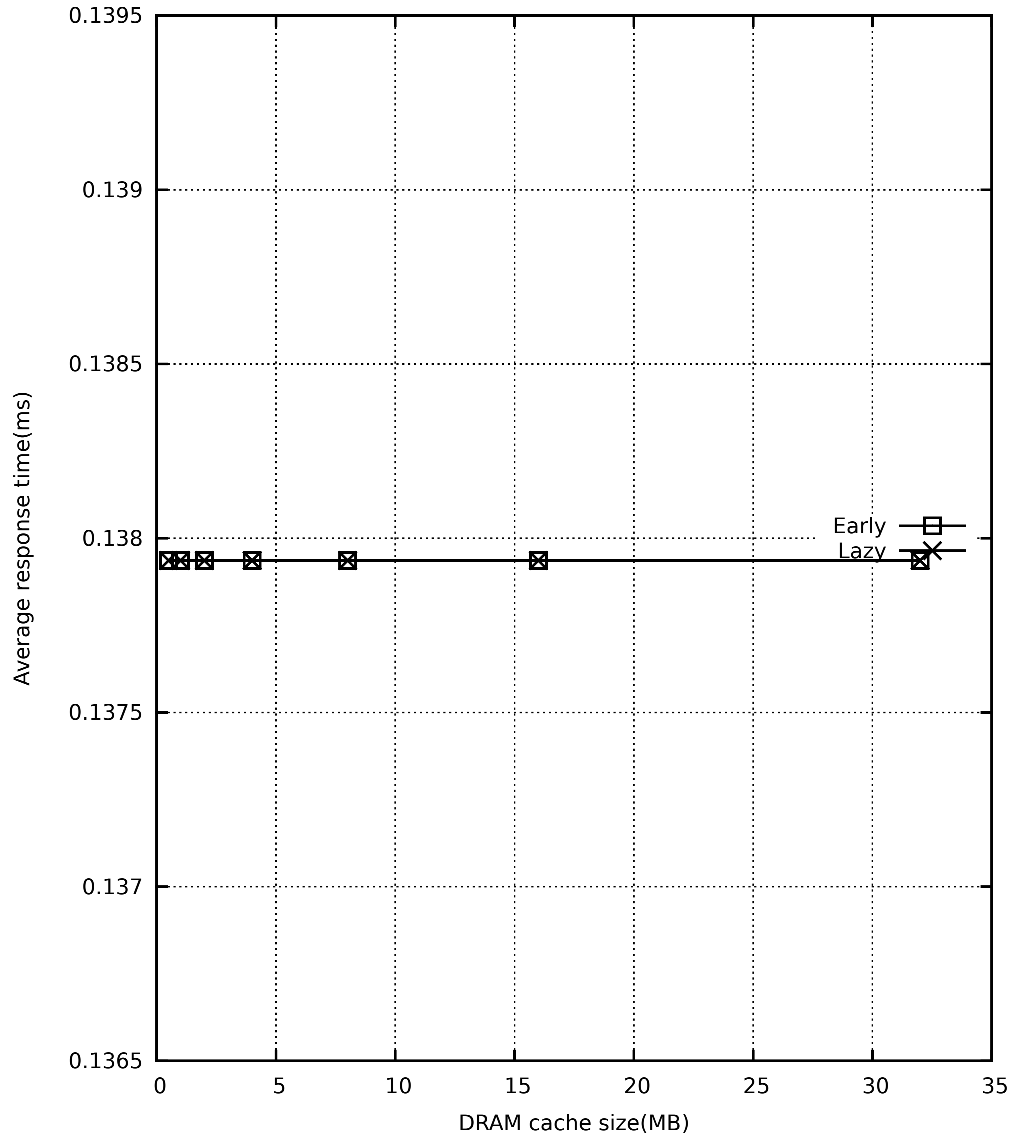


Figure 7.10: Response time for Websearch3

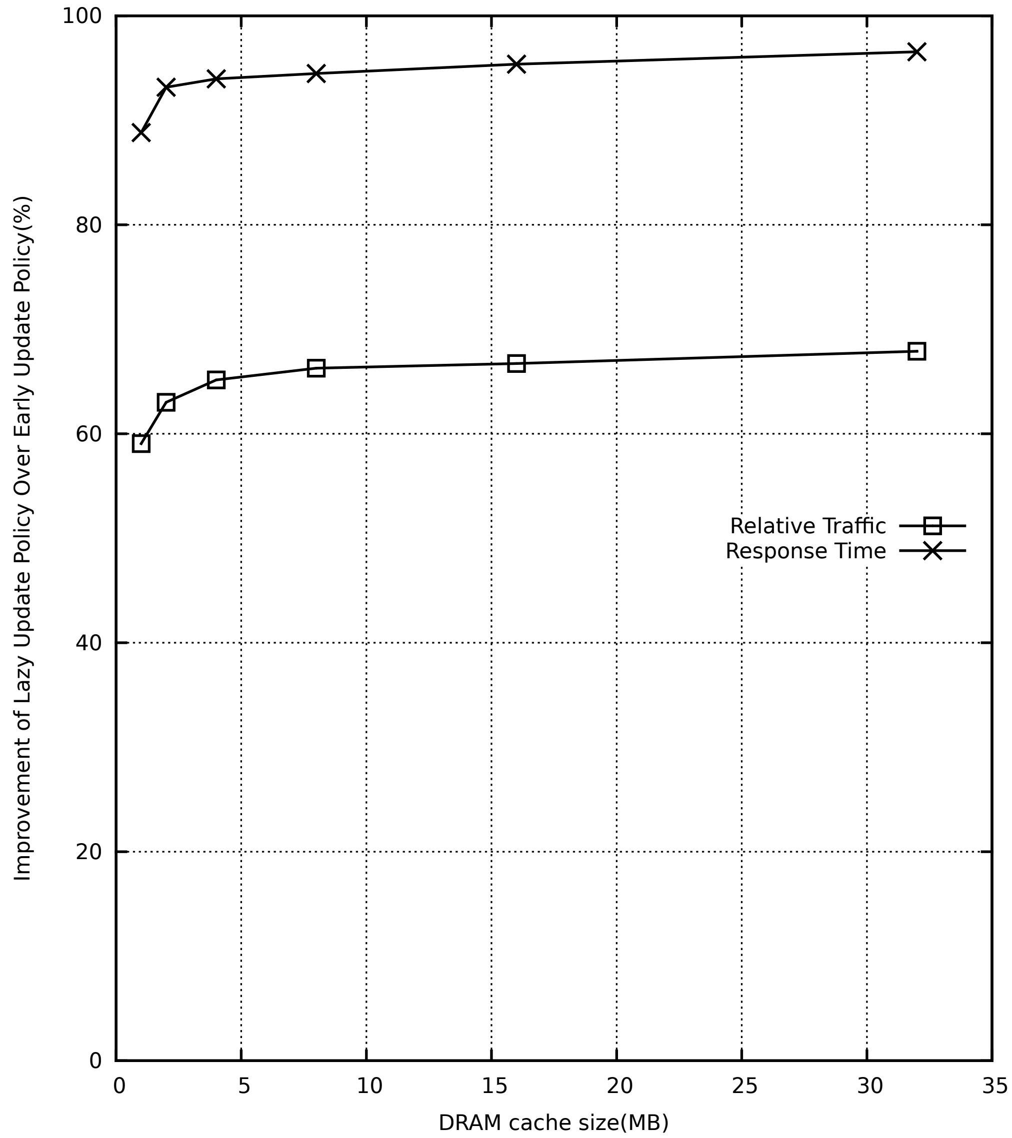


Figure 7.11: Improvement of lazy update policy over early update policy for OpenMail

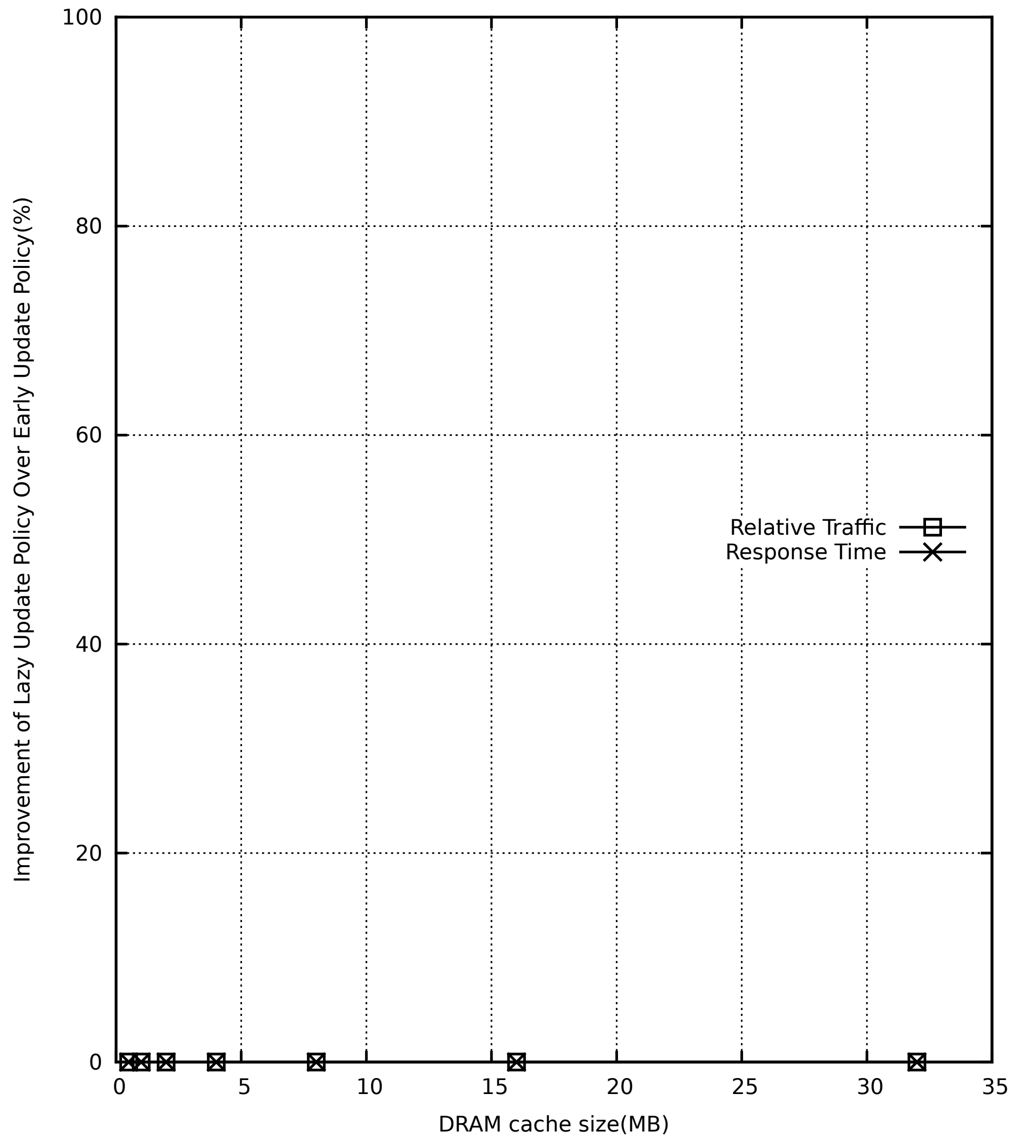


Figure 7.12: Improvement of lazy update policy over early update policy for UMTR(websearch3)

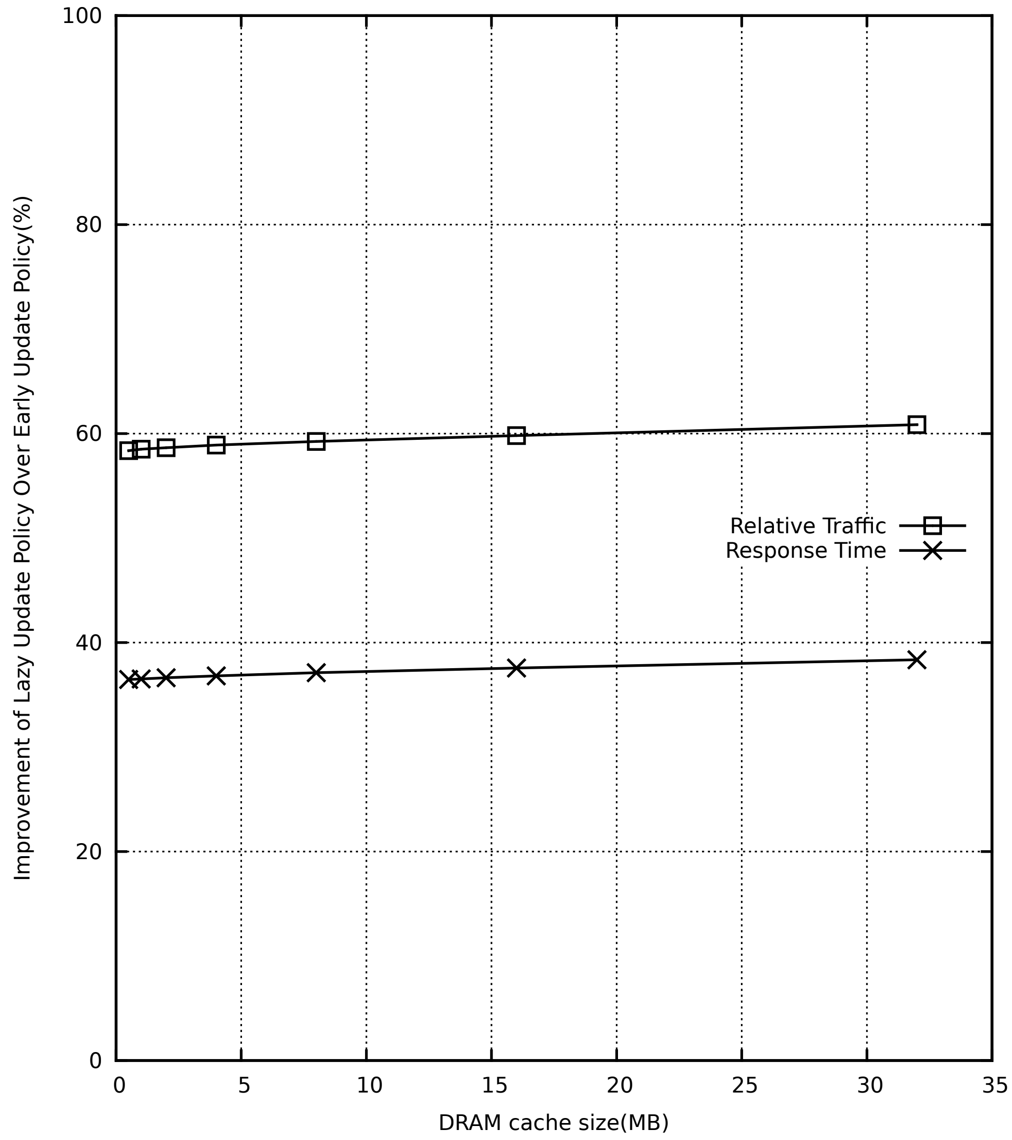


Figure 7.13: Improvement of lazy update policy over early update policy for synthetic workload

Chapter 8

Fault Tolerance Issue

As mentioned in Chapter 3, there is a fault tolerance issue with DRAM cache since DRAM is volatile memory, which will lose data when the power is off. There is battery-backed DRAM. However, batteries have many issues, such as lifetime issue, maintenance issue, recharge-time issue. In this disseration, we propose to use supercapacitors as backup power for the controller and DRAM as shown in Figure 8.1 (flash as major store) and Figure 8.2 (flash as disk cache).

Supercapacitor backup power has been used by Seagate [58] in their SSDs and Sun Oracle [59] in their storage systems. Although the use of supercapacitors as a backup power source is not new, we have not found it being used to solve the flash lifetime issue.

The rest of the chapter is organized as follows: Section 8.1 introduces supercapacitors. In Section 8.2, we discuss the reasons for using supercapacitors instead of batteries. Finally, Section 8.3 presents how to calculate the needed capacitance.

8.1 What are Supercapacitors?

A supercapacitor (also known as ultracapacitor) is an electrochemical capacitor that offers very high capacitance in a small package. The amount of energy a capacitor can hold is measured in microfarads or μF. ($1\mu F = 10^{-6}$ Farad). While small capacitors are rated in nano-farads ($nF=10^{-9}F$) and pico-farads ($1pF = 10^{-12}F$), supercapacitors come in farads.

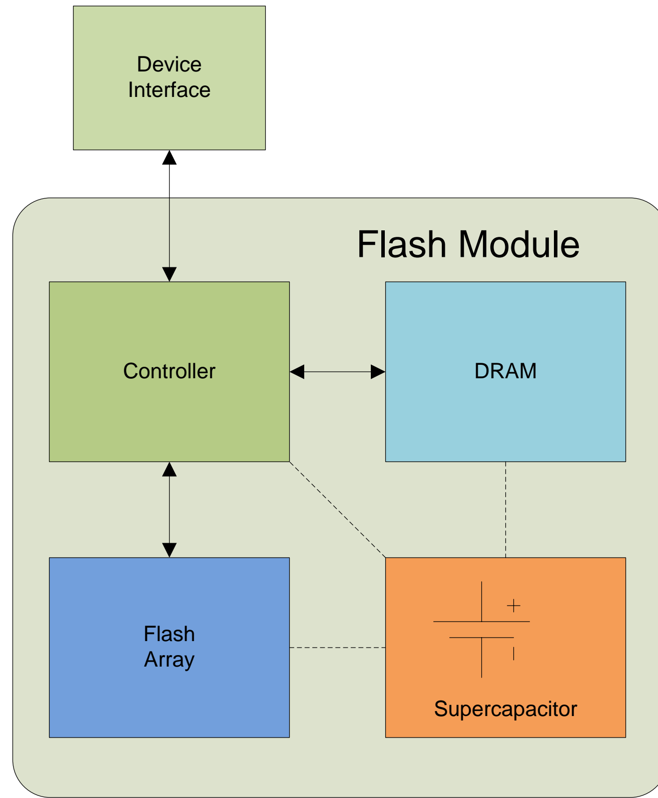


Figure 8.1: Flash as major store with a supercapacitor backup power

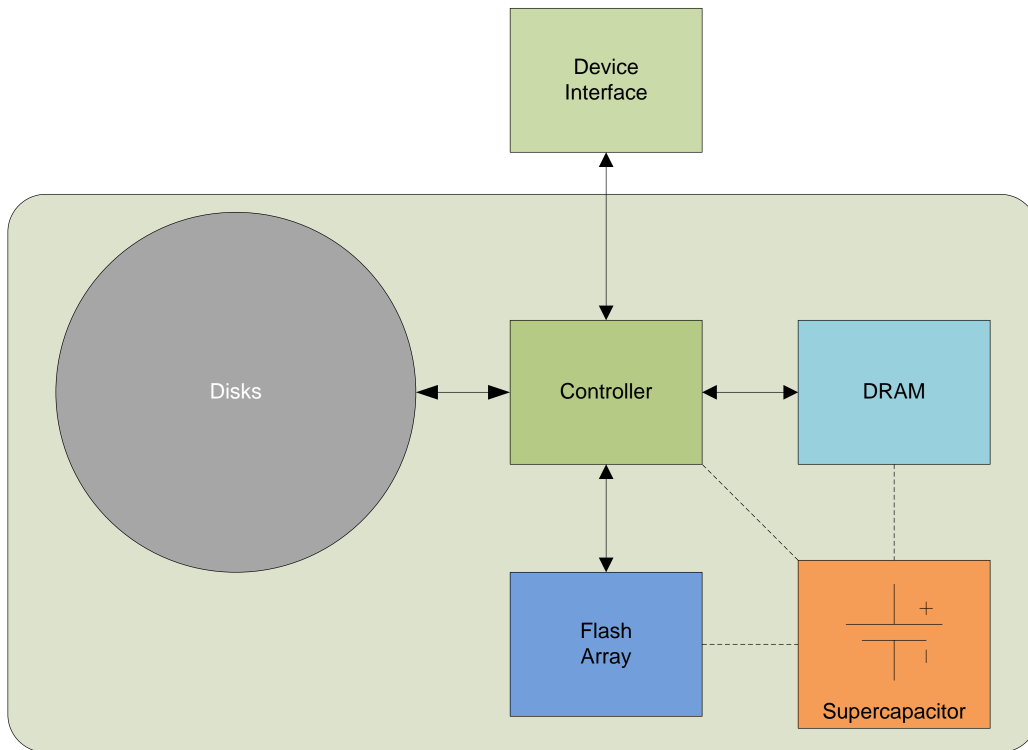


Figure 8.2: Flash as disk cache with a supercapacitor backup power

8.2 Why Supercapacitors not Batteries?

Unlike the electrochemical battery, there is very little wear and tear induced by cycling and age does not affect the supercapacitor much. In normal use, a supercapacitor deteriorates to about 80 percent after 10 years, which is long enough for most applications whereas a battery needs many replacements during the lifetime of a product. Additionally, supercapacitors do not need a full charge detection circuit like rechargeable batteries. They take as much energy as needed. When full, they stop accepting charge. There is no danger of overcharge.

The supercapacitor offers high power density although the energy density is far below that of the battery, as depicted in Figure 8.3. Today, supercapacitors can store 5%-10% as much energy as a modern lithium-ion battery of the same size. What supercapacitors lack in range, they make up in the ability to rapidly charge and discharge. They can be charged in seconds rather than in minutes or hours. Supercapacitors are already used extensively. Millions of them provide backup power for the memory used in microcomputers and cell phones. As mentioned above, supercapacitors have been used in enterprise SSDs.

8.3 How to calculate the Capacitance of Supercapacitors?

The value of a supercapacitor can be estimated [56] by equating the energy needed during the hold-up period to the energy decrease in the supercapacitor, starting at V_{wv} and ending at V_{min}.

The energy (E_x) needed during the hold-up period (t):

$$E_x = I\frac{V_{wv} + V_{min}}{2}t \tag{8.1}$$

The energy decrease (E_y) as voltage drops from V_{wv} to V_{min}:

$$E_y = \frac{CV_{wv}^2}{2} - \frac{CV_{min}^2}{2} \tag{8.2}$$

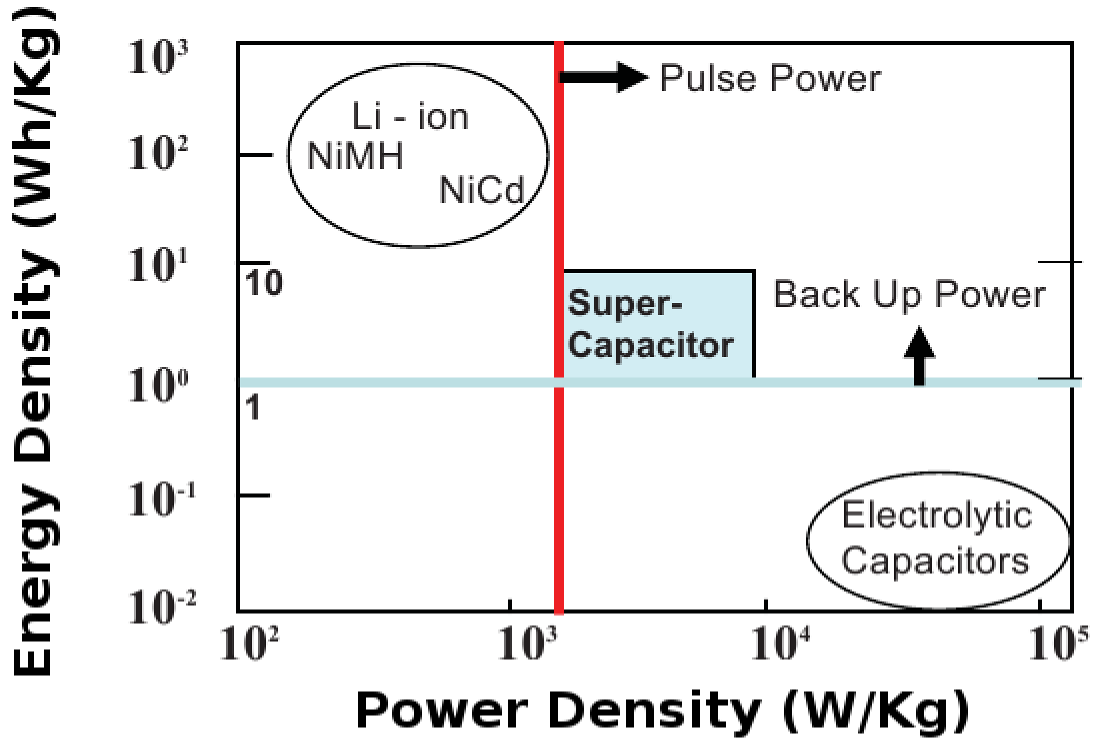


Figure 8.3: Supercapacitor and rechargeable batteries

Since $E_x = E_y$, the minimum capacitance value that guarantees hold-up to V_{min} is:

$$C = I \frac{V_{wv} + V_{min}}{V_{mv}^2 - V_{min}^2} t \tag{8.3}$$

When the main power is off for any reason, the supercapacitor backup power needs to provide the temporary power long enough for the controller to transfer the dirty data into flash.

Suppose we use 64MB of DRAM cache, which is large enough to have an effective write traffic savings. We use Intel X-25M SSD drives [60] to estimate the transfer time. The drive needs 5V (+/-5%) power and the active power is 150mW (current is 0.15/5=0.03A). The sustained sequential write speed is 70MB for 80 GB SSD drives. Therefore, in less than one second, 64MB will be moved into flash. We use 2 seconds to calculate the minimum capacitance value. Applying (8.3), we get C=0.12F. According to Maxwell [61], supercapacitor prices will be approaching $0.01 per farad in production volumes of millions. Apart from

supercapacitors, a power control circuit should be added. However, the total cost would not be high.

Chapter 9

Conclusions and Future Work

In this dissertation, we have concentrated our research on extending the flash lifetime. We have investigated a new way to extend the lifetime of flash with DRAM cache. For data integrity, we propose to use a supercapacitor for backup power instead of batteries so that DRAM cache can be used without loss of data integrity. This chapter concludes the dissertation by summarizing the contributions and describing future directions.

The reminder of the chapter is organized as follows: Section 9.1 highlights the main contributions of the dissertation. In Section 9.2, we concentrate on some future directions, which are extensions of our past and current research on cache. Finally, Section 9.3 summarizes the results and their implications.

9.1 Main Contributions

This dissertation introduced the following main contributions that aim to extend the lifetime of flash:

- **Extend the lifetime of flash in a new way**: DRAM cache has been used to improve performance in terms of response time. Due to its volatile nature, researchers are reluctant to use it to save write traffic. Soundararajan et al. [8] put it this way, "RAM can make for a fast and effective write cache, however the overriding problem with RAM is that it is not persistent (absent some power-continuity arrangements)." The potential of DRAM cache being a write traffic saver has been overlooked. We conducted extensive simulation based on real workloads and synthetic workloads on how effective a DRAM cache can be a write traffic saver and how large the DRAM

cache should be to be effective. Our results show that with a medium-sized DRAM cache, the lifetime of the backing flash can be doubled. Therefore, DRAM cache is a effective write traffic saver.

- **Solve the data integrity**: To be effective and fast is not enough to justify that DRAM cache is a correct candidate unless the data can be guaranteed safe out of power outage. An Achilles' heel is that the data will be lost in case of power failure. A common way to solve this issue is to have a battery backup power. However, batteries have limited charge/discharge cycles and need regular maintenance or replacement, which limits their use in many environments. Instead, we proposed to use a supercapacitor backup power. Supercapacitors are perfect at supplying short period of power in this scenario. Supercapacitors are not new. However, using it to solve the lifetime of flash, to our best knowledge, has not been done.

- **Enhance DiskSim simulator**: DiskSim 4.0 is a well-regarded disk simulator. However, it lacks support of two levels of disk cache (DRAM primary disk cache and secondary flash disk cache). Microsoft SSD add-on does not support a read/write cache. Besides, it does not work well with Type 3 Smart Controller, which supports read/write cache at controller level. Moreover, trace formats (SRT 1.6 and SPC) are not included. We added those missing components to DiskSim and successfully passed all validation tests.

9.2 Future Work

Our work in this dissertation is at device level. However, we realize that the concept of supercapacitor backup power can be applied to computer main memory to enhance data integrity of computer systems as well.

A conventional practice to alleviate data loss due to unexpected power failure is to enforce a 30-second flush rule, as Unix operating systems do. The negative side of the rule is

that disk fragmentation is increased, which will decrease its performance. Some important computer systems are even equipped with Uninterruptible Power Supply (UPS) to protect its data.

With a supercapacitor backup power, backing flash, and controller, the content of main memory can be backed up into flash on power loss. The data can be recovered on power resumption. In this context, 30-second flush rule is no longer needed, which would reduce disk fragmentation and improve data integrity as well as performance.

9.3 Conclusions

We have demonstrated through simulation that a medium-sized DRAM cache can save up to 50% write traffic to flash, which is translated into at least doubling the lifetime of flash. Meanwhile, performance in terms of response time and miss ratio sees improvement as well. Furthermore, our findings can be applied to computer main memory to enhance data integrity of computer systems.

Bibliography

[1] W. Hsu and A. J. Smith, "The performance impact of I/O optimizations and disk improvements," *IBM J. Res. Dev.*, vol. 48, no. 2, pp. 255–289, 2004.

[2] Solaris^{TM} ZFS^{TM} enables hybrid storage pools–Shatters economic and performance barriers. [Online]. Available: http://download.intel.com/design/flash/nand/SolarisZFS_SolutionBrief.pdf

[3] "Understanding the flash translation layer (FTL) specification," Intel Corporation, Tech. Rep., 1998.

[4] Y. Kim, A. Gupta, and B. Urgaonkar, "MixedStore: An enterprise-scale storage system combining Solid-state and Hard Disk Drives ," The Pennsylvania State University, Tech. Rep., 2008.

[5] "NAND evolution and its effects on Solid State Drive (SSD) useable life," Western Digital, White paper WP-001-01R, 2009.

[6] J. Hutchby and M. Garner. Assessment of the potential & maturity of selected emerging research memory technologies. Workshop & ERD/ERM Working Group Meeting (April 6-7, 2010). [Online]. Available: http://www.itrs.net/Links/2010ITRS/2010Update/ToPost/ERD_ERM_2010FINALReportMemoryAssessment_ITRS.pdf

[7] (2010) Process integration, devices & structures. The International Technology Roadmap for Semiconductors. [Online]. Available: http://www.itrs.net/Links/2010ITRS/Home2010.htm

[8] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *Proceedings of the 8th USENIX conference on File and storage technologies*, ser. FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 8–8. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855511.1855519

[9] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 234–241. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176789

[10] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: flash-aware buffer management policy for portable media players," *Consumer Electronics, IEEE Transactions on*, vol. 52, no. 2, pp. 485 – 493, May 2006.

[11] H. Kim and S. Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 16:1–16:14. [Online]. Available: http://portal.acm.org/citation.cfm?id=1364813.1364829

[12] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2009, pp. 229–240.

[13] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, April 2003.

[14] Flash memory. [Online]. Available: http://en.wikipedia.org/wiki/Flash_memory

[15] K. Takeuchi, S. Satoh, T. Tanaka, K. Imamiya, and K. Sakui, "A negative V_{th} cell architecture for highly scalable, excellently noise-immune, and highly reliable NAND flash memories," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 5, pp. 675 –684, May 1999.

[16] J.-W. Hsieh, T.-W. Kuo, P.-L. Wu, and Y.-C. Huang, "Energy-efficient and performance-enhanced disks using flash-memory cache," in *Proceedings of the 2007 international symposium on Low power electronics and design*, ser. ISLPED '07. New York, NY, USA: ACM, 2007, pp. 334–339. [Online]. Available: http://doi.acm.org/10.1145/1283780.1283851

[17] T. Bisson, S. A. Brandt, and D. D. E. Long, "A hybrid disk-aware spin-down algorithm with I/O subsystem support." in *IPCCC'07*, 2007, pp. 236–245.

[18] T. Bisson and S. A. Brandt, "Reducing Hybrid Disk write latency with flash-backed I/O requests," in *Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 402–409. [Online]. Available: http://portal.acm.org/citation.cfm?id=1474555.1475519

[19] ——, "Flushing policies for NVCache enabled hard disks," in *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 299–304. [Online]. Available: http://portal.acm.org/citation.cfm?id=1306871.1306922

[20] R. Panabaker. (2006, May) Hybrid hard disk and Ready-Drive technology: Improving performance and power for Windows Vista mobile PCs. [Online]. Available: http://www.microsoft.com/whdc/winhec/pres06.mspx

[21] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng, "A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems," in *Proceedings of the 8th ACM international conference on Embedded software*, ser. EMSOFT '08. New York, NY, USA: ACM, 2008, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1450058.1450064

[22] Y. Joo, Y. Cho, K. Lee, and N. Chang, "Improving application launch times with hybrid disks," in *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2009, pp. 373–382.

[23] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel® Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *Trans. Storage*, vol. 4, no. 2, pp. 1–24, 2008.

[24] T. Kgil and T. Mudge, "Flashcache: a NAND flash memory file cache for low power web servers," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 103–112. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176774

[25] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 327–338. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.32

[26] D. Roberts, T. Kgil, and T. Mudge, "Integrating NAND flash devices onto servers," *Commun. ACM*, vol. 52, pp. 98–103, April 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498791

[27] Solid-state drive. [Online]. Available: http://en.wikipedia.org/wiki/Solid-state_drive

[28] Wear leveling. [Online]. Available: http://en.wikipedia.org/wiki/Wear_levelling

[29] Journalling flash file system version 2. [Online]. Available: http://en.wikipedia.org/wiki/JFFS2

[30] Yaffs (yet another flash file system). [Online]. Available: http://en.wikipedia.org/wiki/YAFFS

[31] ZFS. [Online]. Available: http://en.wikipedia.org/wiki/Zfs

[32] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508271

[33] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Improving flash wear-leveling by proactively moving static data," *IEEE Trans. Comput.*, vol. 59, pp. 53–65, January 2010. [Online]. Available: http://dx.doi.org/10.1109/TC.2009.134

[34] ——, "Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 212–217. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278533

[35] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proceedings of the 2007 ACM symposium on Applied computing*, ser. SAC '07. New York, NY, USA: ACM, 2007, pp. 1126–1130. [Online]. Available: http://doi.acm.org/10.1145/1244002.1244248

[36] A. Ban, "Wear leveling of static areas in flash memory," U.S. Patent 6 732 221, 2004.

[37] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 13–13. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267411.1267424

[38] H.-j. Kim and S.-g. Lee, "A new flash memory management for flash storage system," in *23rd International Computer Software and Applications Conference*, ser. COMPSAC '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 284–. [Online]. Available: http://portal.acm.org/citation.cfm?id=645981.674620

[39] K. M. J. Lofgren, R. D. Norman, G. B. Thelin, and A. Gupta, "Wear leveling techniques for flash EEPROM systems," U.S. Patent 6 850 443, February, 2005. [Online]. Available: http://www.freepatentsonline.com/6850443.html

[40] E. Jou and J. H. Jeppesen, III, "Flash memory wear leveling system providing immediate direct access to microprocessor," U.S. Patent 5 568 423, 1996. [Online]. Available: http://www.freepatentsonline.com/5568423.html

[41] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee, "A group-based wear-leveling algorithm for large-capacity flash memory storage systems," in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '07. New York, NY, USA: ACM, 2007, pp. 160–164. [Online]. Available: http://doi.acm.org/10.1145/1289881.1289911

[42] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, July 2007. [Online]. Available: http://doi.acm.org/10.1145/1275986.1275990

[43] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, ser. EMSOFT '06. New York, NY, USA: ACM, 2006, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/1176887.1176911

[44] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, ser. RTAS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 187–. [Online]. Available: http://portal.acm.org/citation.cfm?id=827265.828513

[45] Y.-S. Chu, J.-W. Hsieh, Y.-H. Chang, and T.-W. Kuo, "A set-based mapping strategy for flash-memory reliability enhancement," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 405–410. [Online]. Available: http://portal.acm.org/citation.cfm?id=1874620.1874717

[46] Y.-L. Tsai, J.-W. Hsieh, and T.-W. Kuo, "Configurable NAND flash translation layer," in *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06) - Volume 01*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 118–127. [Online]. Available: http://portal.acm.org/citation.cfm?id=1136649.1137086

[47] G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, Y. Chen, and H. Li, "A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 9-14 2010, pp. 1 –12.

[48] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[49] "The basics of phase change memory (PCM) technology," Numonyx White Paper. [Online]. Available: www.numonyx.com/Documents/WhitePapers/PCM_Basics_WP.pdf

[50] J. S. Bucy, J. Schindler, S. Schlosser, G. R. Ganger, and Contributors, *The DiskSim simulation environment version 4.0 reference manual*, Carnegie Mellon University, Pittsburgh, PA, 2008.

[51] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70. [Online]. Available: http://portal.acm.org/citation.cfm?id=1404014.1404019

[52] HP open source software. [Online]. Available: http://tesla.hpl.hp.com/opensource/

[53] Block I/O traces from SNIA. [Online]. Available: http://iotta.snia.org/traces/list/BlockIO

[54] C. Ruemmler and J. Wilkes, "Unix disk access patterns." in *USENIX Winter'93*, 1993, pp. 405–420.

[55] University of Massachusetts trace. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[56] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009, pp. 10:1–10:9. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534544

[57] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 8–8. [Online]. Available: http://portal.acm.org/citation.cfm?id=1247360.1247368

[58] Pulsar. Seagate. [Online]. Available: http://www.seagate.com/staticfiles/support/disc/manuals/ssd/100596473a.pdf

[59] Sun Storage F5100 flash array. Sun Oracle. [Online]. Available: http://www.oracle.com/us/products/servers-storage/storage/disk-storage/043970.pdf

[60] Intel® X18-M/X25-M SATA Solid State Drive-34 nm product line. Intel Corporation. [Online]. Available: http://download.intel.com/design/flash/nand/mainstream/322296.pdf

[61] A. Schneuwly, G. Sartorelli, J. Auer, and B. Maher. Ultracapacitor applications in the power electronic world. Maxwell Technologies. [Online]. Available: http://www.maxwell.com/ultracapacitors/white-papers/power_electronic_applications.pdf

Appendices

Appendix A
DiskSim 4.0 Flowcharts, Modifications,
and Post Processing Scripts

A.1 Overview

DiskSim 4.0 is a well regarded tool to simulate the storage system. We used DiskSim 4.0 along with Microsoft SSD add-on in our storage research, during which we made modifications on DiskSim 4.0. We added two trace formats (SRT 1.6 and SPC) to DiskSim and fixed bugs related to smart controller (type:3) for Microsoft SSD add-on. Moreover, we wrote Bash scripts to automatically plot figures using gnuplot.

I/O Block Traces (e.g., OpenMail, Cello) that can be downloaded from SNIA (http://www.snia.org) and HP Labs (http://tesla.hpl.hp.com/opensource) use SRT 1.6 while DiskSim 4.0 only supports SRT 1.4. SPC is another important trace format that DiskSim 4.0 does not support. We have a set of workloads in SPC format that can be downloaded from http://traces.cs.umass.edu/index.php/Storage/Storage. We implemented the two trace formats in DiskSim 4.0. In addition, we added five parameters to DiskSim to filter traces, e.g., tracing read only I/O requests.

Microsoft SSD add-on does not include a write or read cache in the implementation, which is a problem for those who need the write or read cache. One way to solve this problem is to use Smart Controller (type:3). However, the SSD implementation is not compatible with type 3 smart controller. We have an update to solve this issue.

In order to make these modifications, we read thoroughly through the DiskSim 4.0 source code and stepped through the source code. Based on these, we drew flowcharts of reads, writes both for HDD and SSD, which will help us understand how DiskSim 4.0 works internally. With these flowcharts, it is much easier to comprehend DiskSim 4.0 source code. We would like to share these flowcharts to help those who are reading DiskSim source code. So far, we have not found such kinds of published materials.

Last but not least, the task of running DiskSim 4.0 and collecting data can be tiresome since most likely we need to change a series of parameters to see the impact, which will produce many files, from which we extract the wanted outcomes to produce figures. With our Bash scripts, we obtain the figures by running a single Bash shell program.

A.2 System Diagram

DiskSim is a widely used disk drive simulator both in academia and industry alike. The system diagram of DiskSim 4.0 is shown in Figure A.1. DiskSim is an event-driven simulator. It emulates a hierarchy of storage components such as device drivers, buses, and controllers as well as disks. With Microsoft SSD extension, it can also simulate flash-based Solid State Drives. Furthermore, it is designed to be easily integrated into a full system simulator, like SimOS.

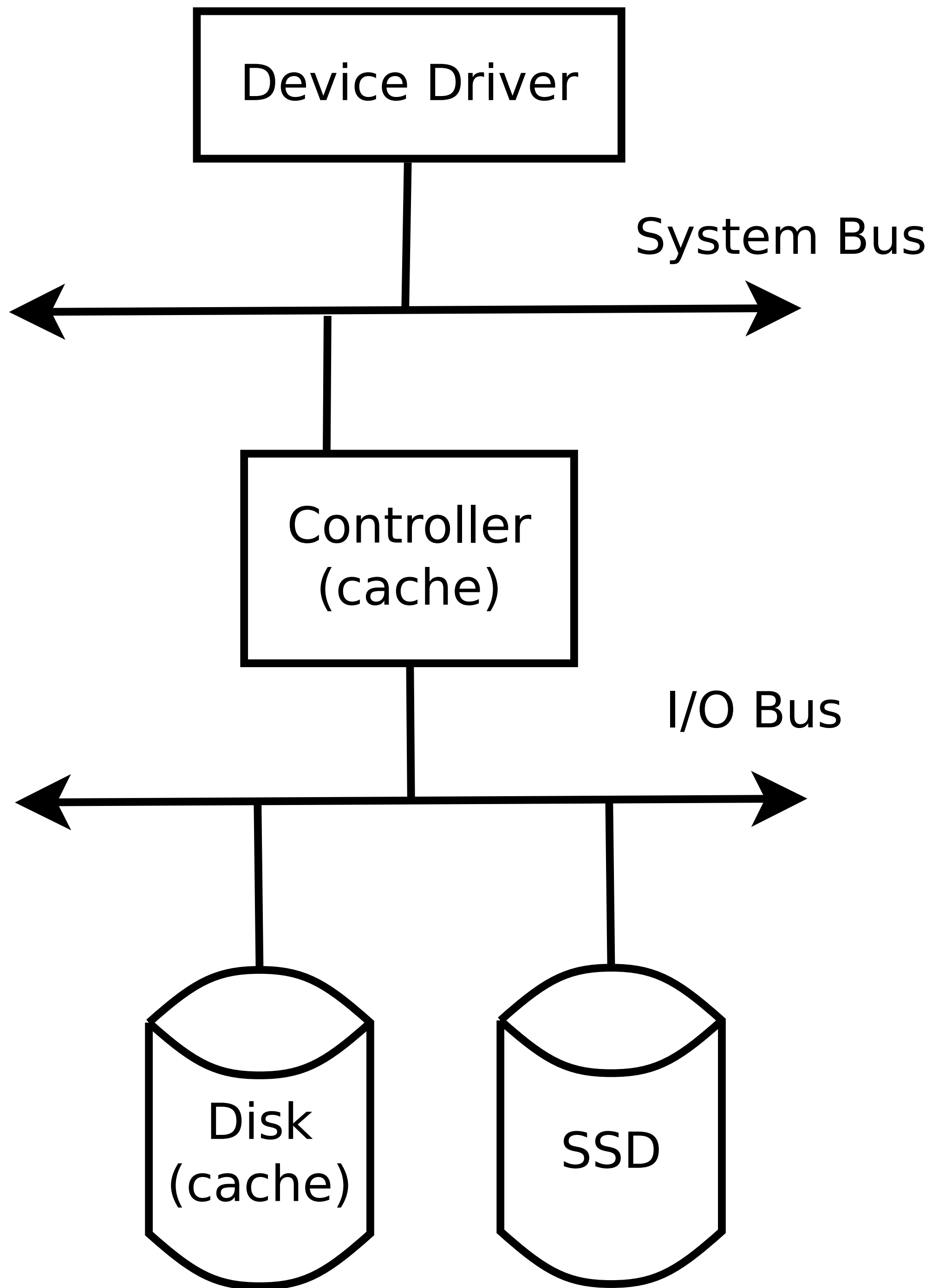


Figure A.1: System architecture of DiskSim 4.0

A.3 DiskSim 4.0 flowcharts

A.3.1 Main loop

DiskSim is written mainly in C with a small portion of perl and python code. The main loop of DiskSim in disksim_main.c is shown in Figure A.2. After initialization, it runs simulation (void disksim_run_simulation ()) until all I/O requests are processed or it runs out of time. The function disksim_run_simulation() is shown in Figure A.3. It gets the next event and processes the event according to event types: IO_event, PF, Timeout, and Checkout. The detailed io_internal_event is shown in Figure A.4. It handles events based on the event types: driver event, bus event, controller event, and device event.

A.3.2 Message routing

As mentioned earlier, DiskSim is a event-driven simulator. Events are put into queues to be handled. There are two kinds of queues: internal Queue and extra Queue shown in Figure A.5. Extra Q is a pool of queues, which is in charge of memory allocation. A queue can be gotten via getfromextraq() and it must be returned via addtoextraq() upon completion. Internal Q, on the other hand, is a list of active messages (allocated from Extra Q) passing among the device driver, bus, controller, and devices. An event is added to the Internal Q via addtointq() and removed via getfromintq(). For example, an I/O request is put into the Internal Q by the workload generator. Then, it is processed along the path: device driver, system bus, controller, I/O bus, and devices.

A.3.3 HDD reads with dumb controller (type:1)

The flowchart for HDD reads is shown in Figure A.6. The flowchart is based on the trace to the run of DiskSim with parameter file atlas_III.parv. Different parameter settings may have slightly different flowcharts. In this category, the dumb controller passes all messages without processing them upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. Then, it reaches the device (disk). The disk issues IIA(disconnect) to disconnect the I/O bus, which is completed upon receiving IIC(disconnect). When the data is ready, the disk issues IIA(reconnect) to reconnect the I/O bus, which is finished upon receiving IIC(reconnect). Then, data is transfered via DDTC, which is done upon receiving DDTC. Finally, the disk issues IIA(completion) to complete the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

A.3.4 HDD writes with dumb controller (type:1)

The flowchart for HDD writes is shown in Figure A.7. The flowchart is based on the trace to the run of DiskSim with parameter file atlas_III.parv. Like reads, the dumb controller passes all messages without processing them upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. Then, it reaches the device (disk). The disk issues IIA(reconnect) to reconnect the I/O bus, which is completed upon receiving IIC(reconnect). Then, it issues DDTC to start data transfer,

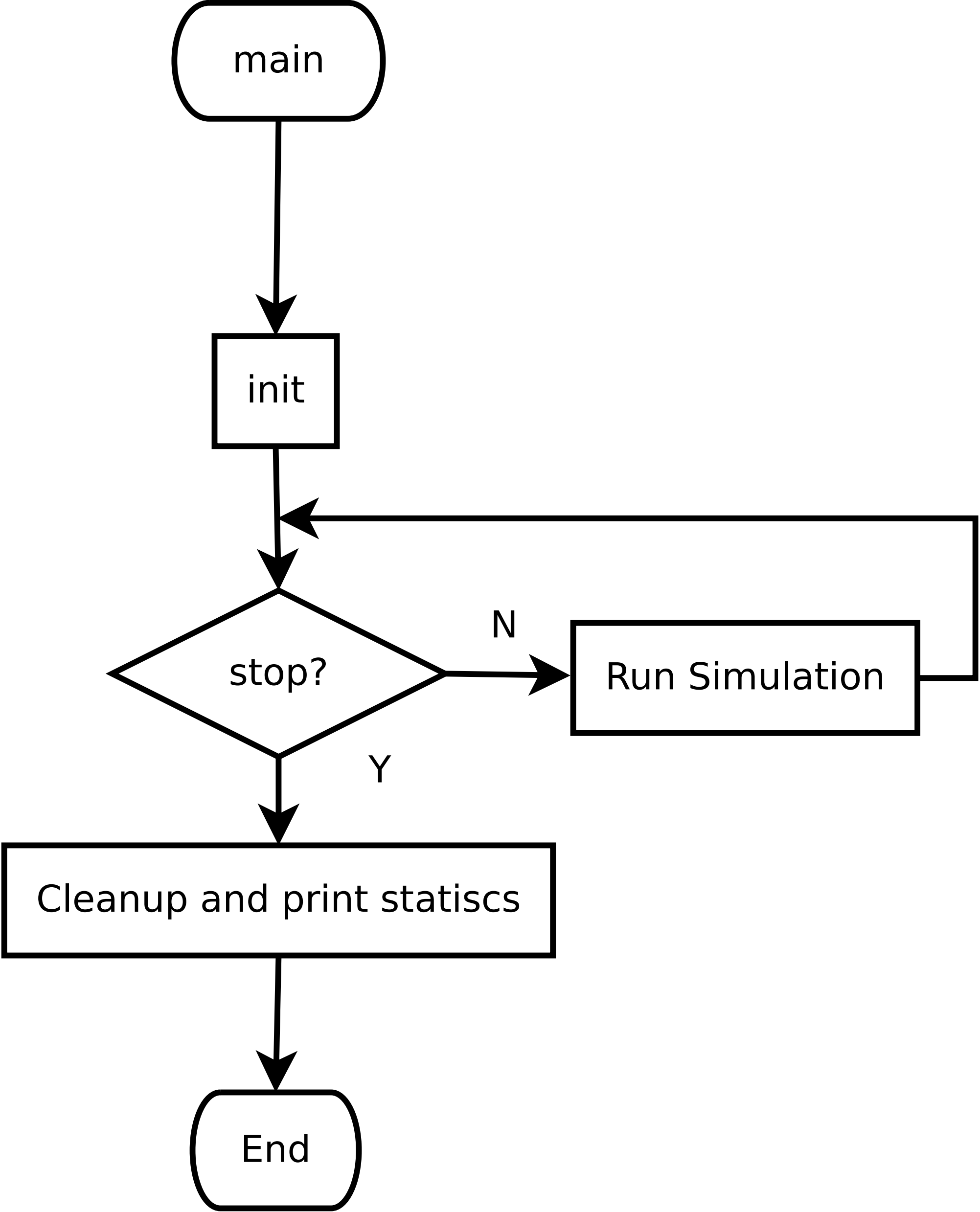


Figure A.2: The main loop of DiskSim 4.0

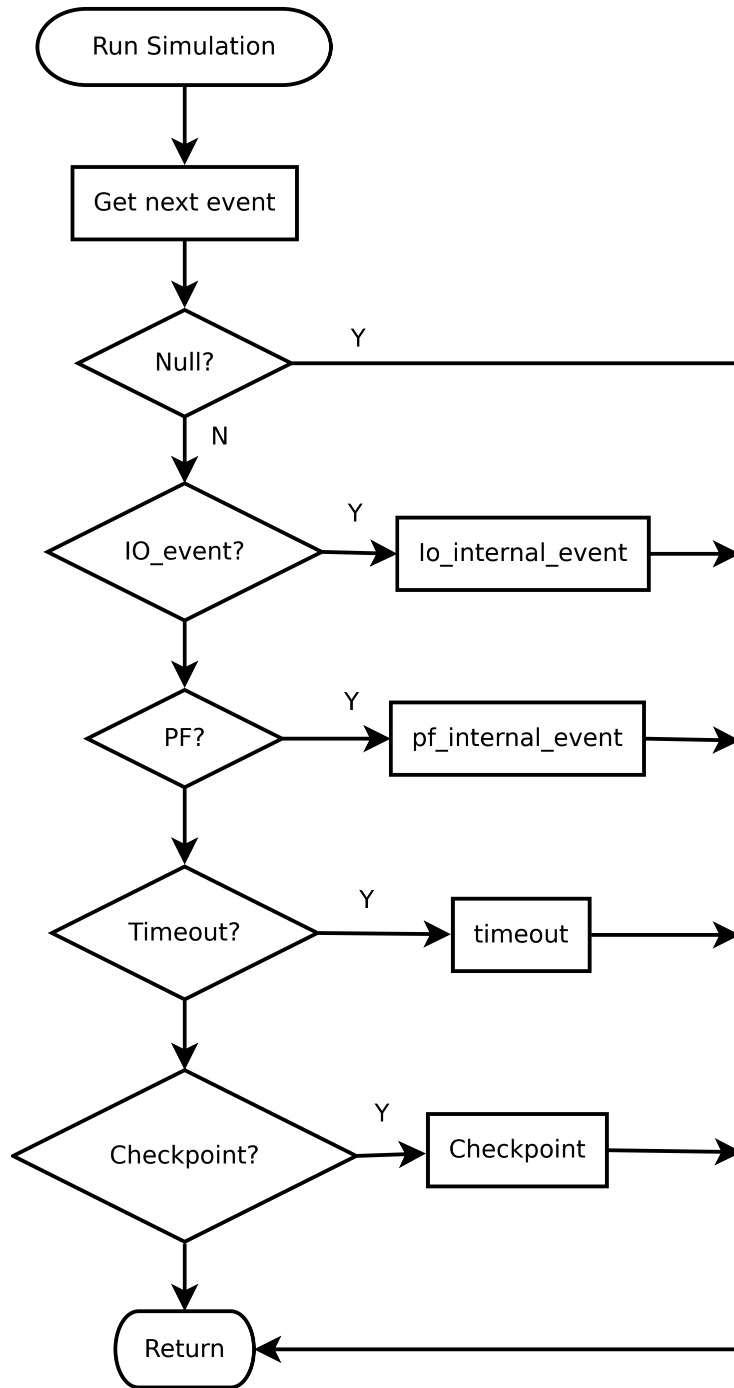


Figure A.3: Run simulation

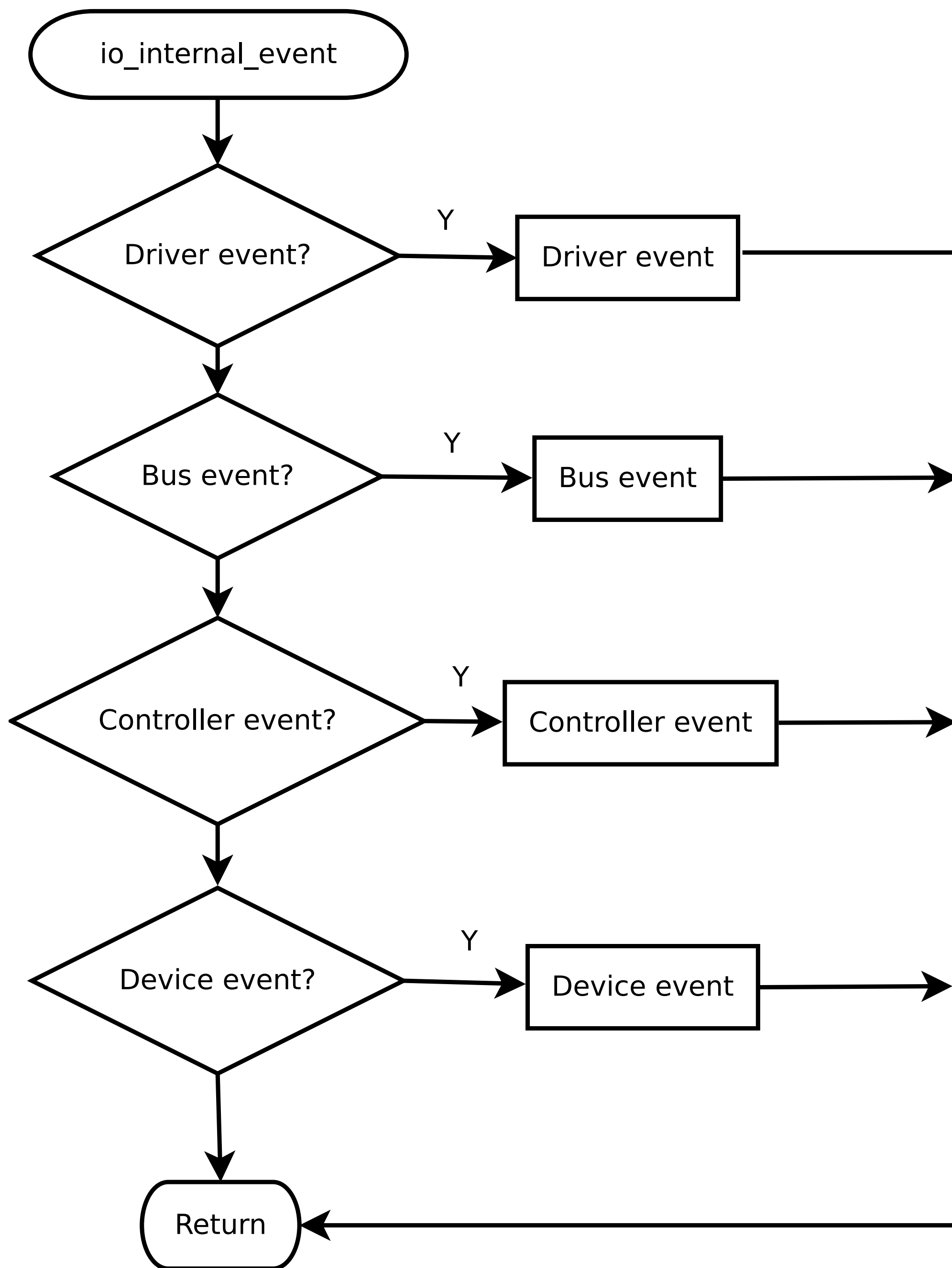


Figure A.4: io-internal-event

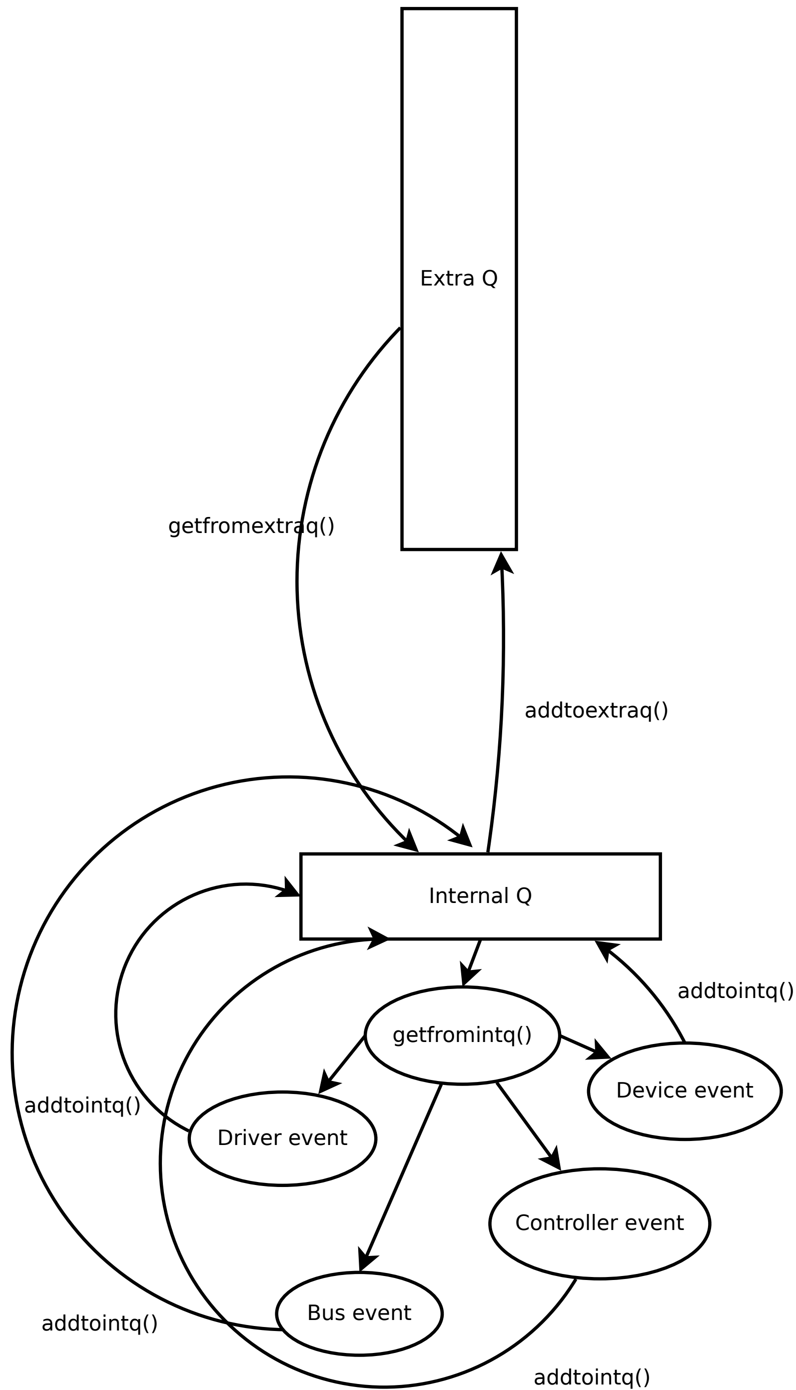
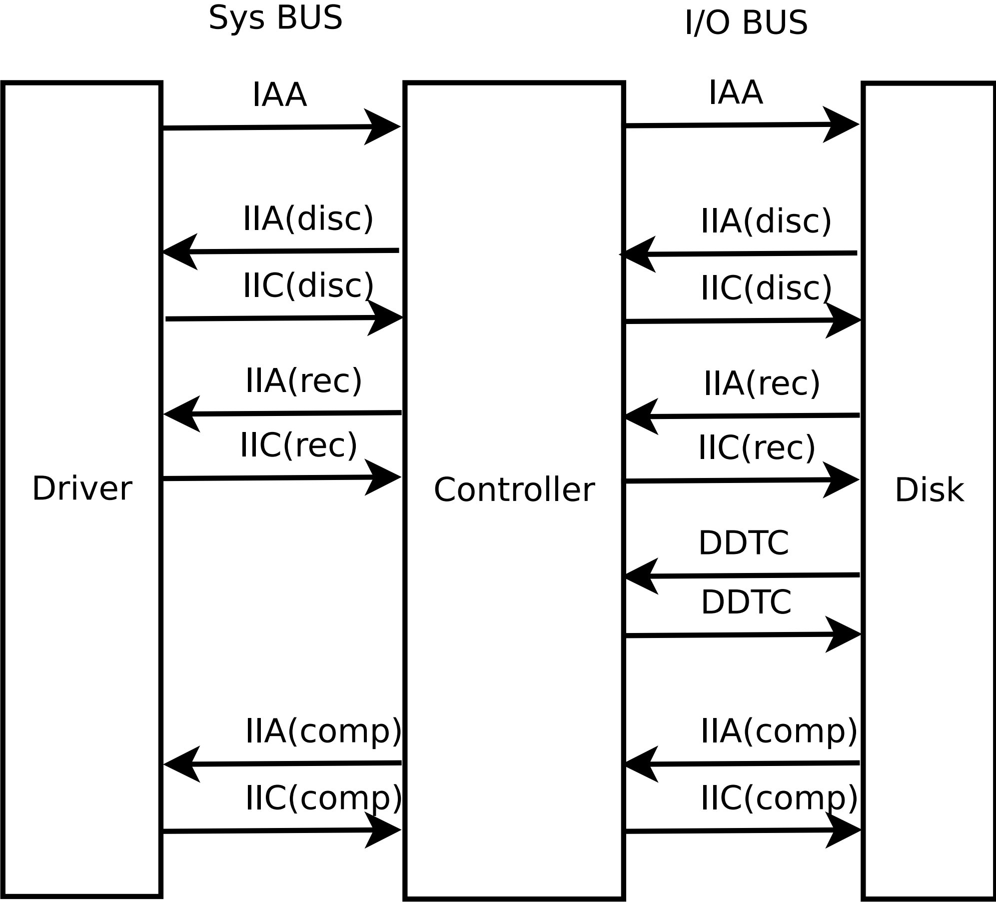


Figure A.5: Message routing



IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE

Figure A.6: Flowchart for HDD reads (controller type:1)

which is done upon receiving DDTC. Finally, the disk sends IIA(completion) to complete the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

A.3.5 HDD reads with smart controller (type:3)

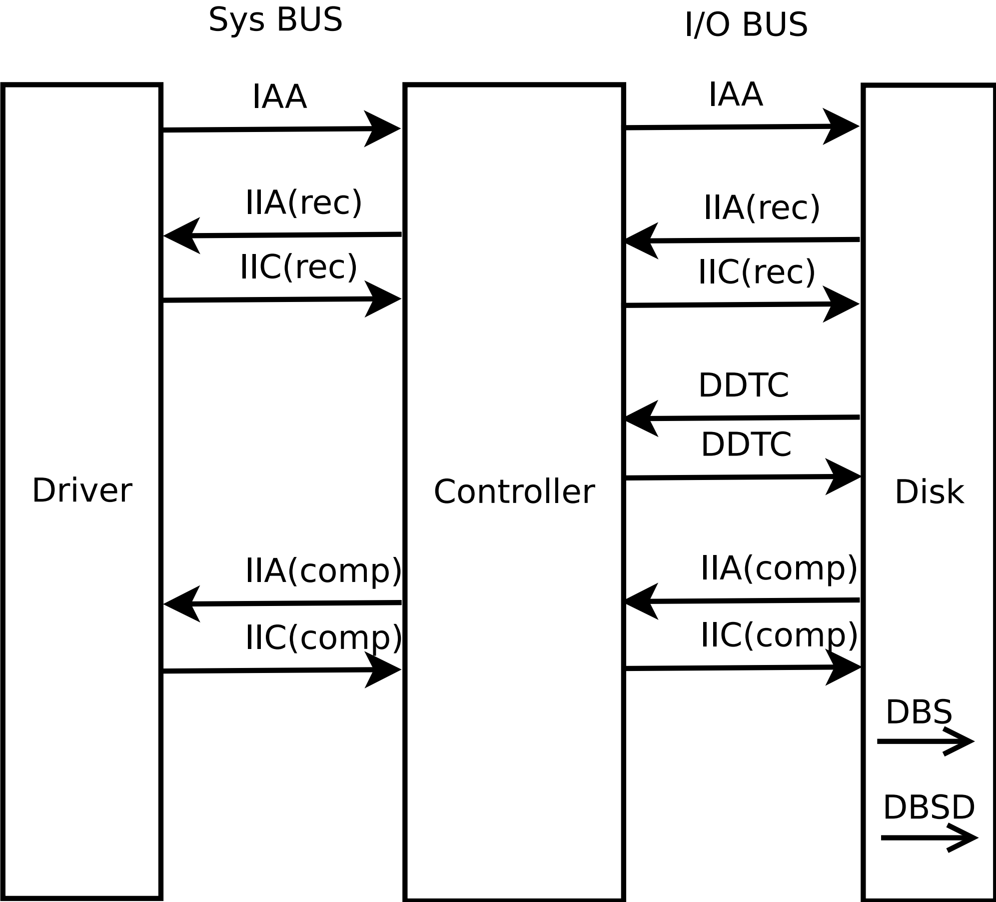
The flowchart for HDD reads is shown in Figure A.8. The flowchart is based on the trace to the run of DiskSim with parameter file atlas_III3.parv, which sets up smart controller (type:3). In this category, the smart controller processes all messages upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. Then, it reaches the device (disk). The disk issues IIA(disconnect) to disconnect the I/O bus, which is completed upon receiving IIC(disconnect). Unlike dumb controller, smart controller processes IIA and returns IIC. When the data is ready, the disk issues IIA(reconnect) to reconnect the I/O bus, which is finished upon receiving IIC(reconnect). Then, data is transfered via DDTC, which is done upon receiving DDTC. The disk issues IIA(completion) to complete the cycle. Data is transmitted between the device driver and the controller via CDTCH. Finally, the controller issues an IIA(completion) to wrap up the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

A.3.6 HDD writes with smart controller (type:3)

The flowchart for HDD writes is shown in Figure A.9. The flowchart is based on the trace to the run of DiskSim with parameter file atlas_III3.parv, which sets up smart controller (type:3). Like reads, the smart controller processes all messages upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. CDTCH/CDTCH starts data transfer between the device driver and the controller. Then, IAA reaches the device (disk). Next, the disk issues IIA(reconnect) to reconnect the I/O bus, which is completed upon receiving IIC(reconnect). Unlike dumb controller, smart controller processes IIA and returns IIC. Then, data is transfered via DDTC, which is done upon receiving DDTC. Next, the disk issues IIA(completion) to complete the cycle, which is done upon receiving IIC(completion). DBS and DBSD are internal events in the disk. Finally, the controller issues an IIA(completion) to wrap up the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

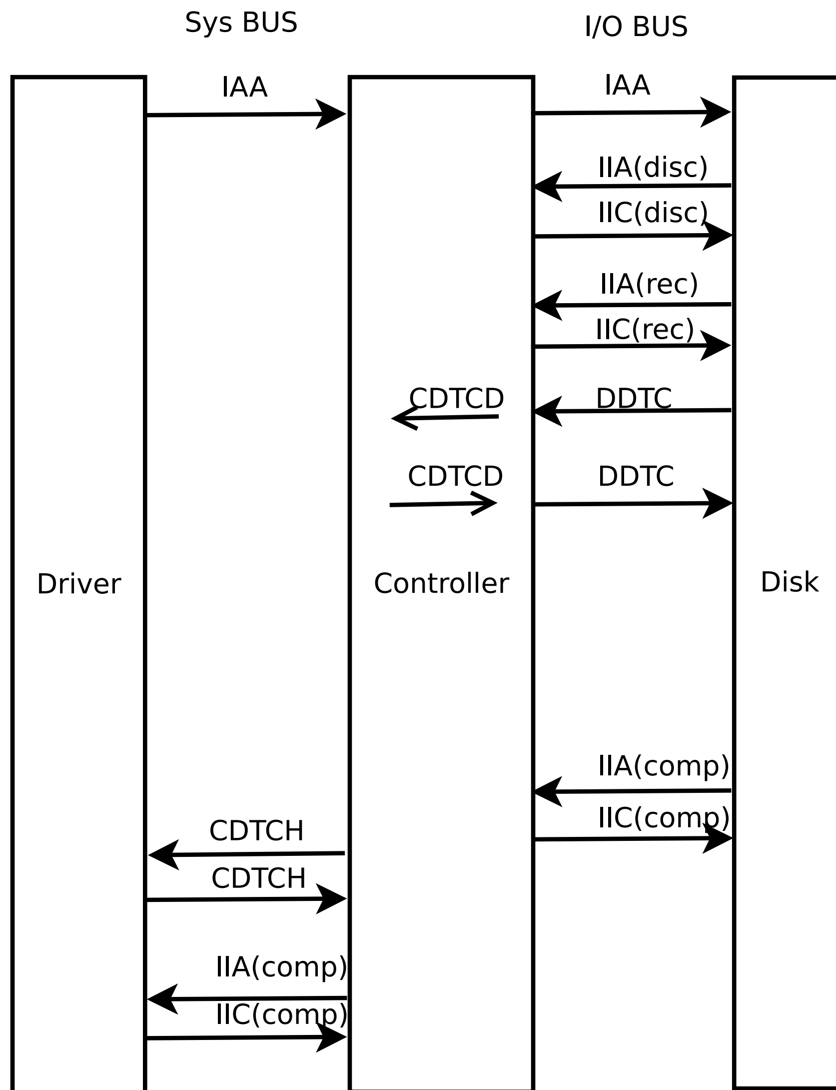
A.3.7 SSD reads with dumb controller (type:1)

The flowchart for SSD reads is shown in Figure A.10. The flowchart is based on the trace to the run of DiskSim with parameter file ssd-postmark.parv. In this category, the dumb controller passes all messages without processing them upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. Then, it reaches the device (SSD). When the data is ready, the SSD issues IIA(reconnect) to reconnect the I/O bus, which is finished upon receiving IIC(reconnect). Then, data is transfered via DDTC, which is done upon receiving DDTC. Finally, the SSD issues IIA(completion) to complete the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).



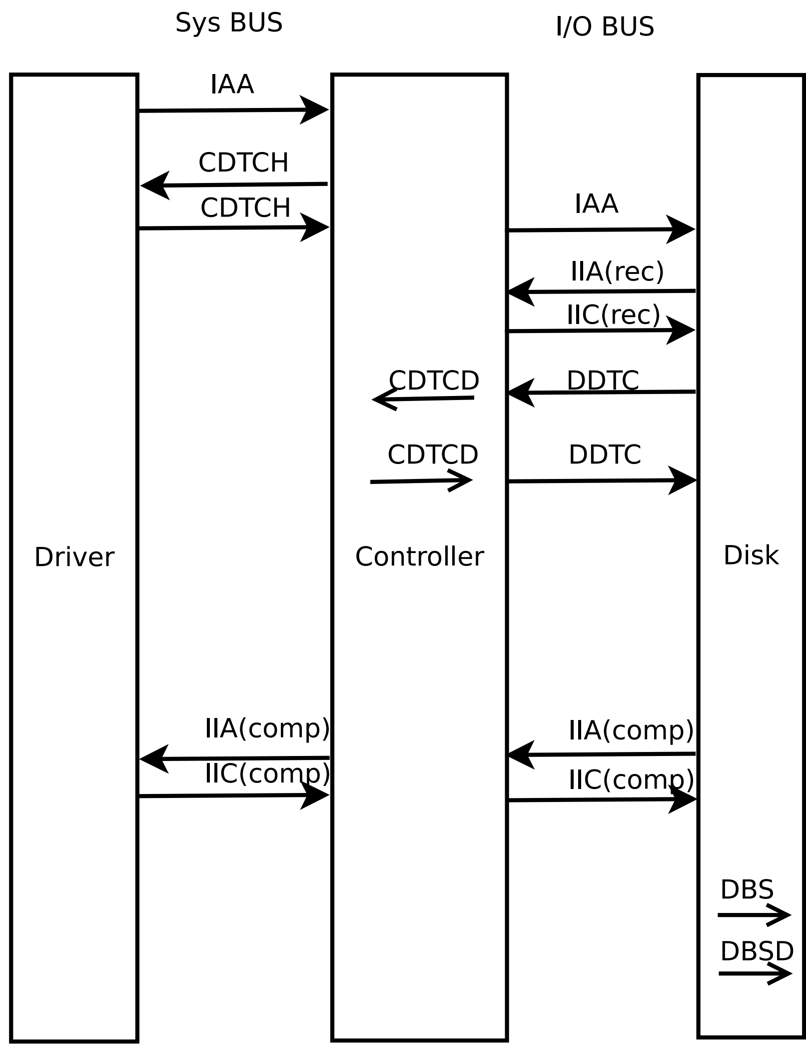
IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
DBS: DEVICE_BUFFER_SEEKDONE
DBSD: DEVICE_BUFFER_SECTOR_DONE

Figure A.7: Flowchart for HDD writes (controller type:1)



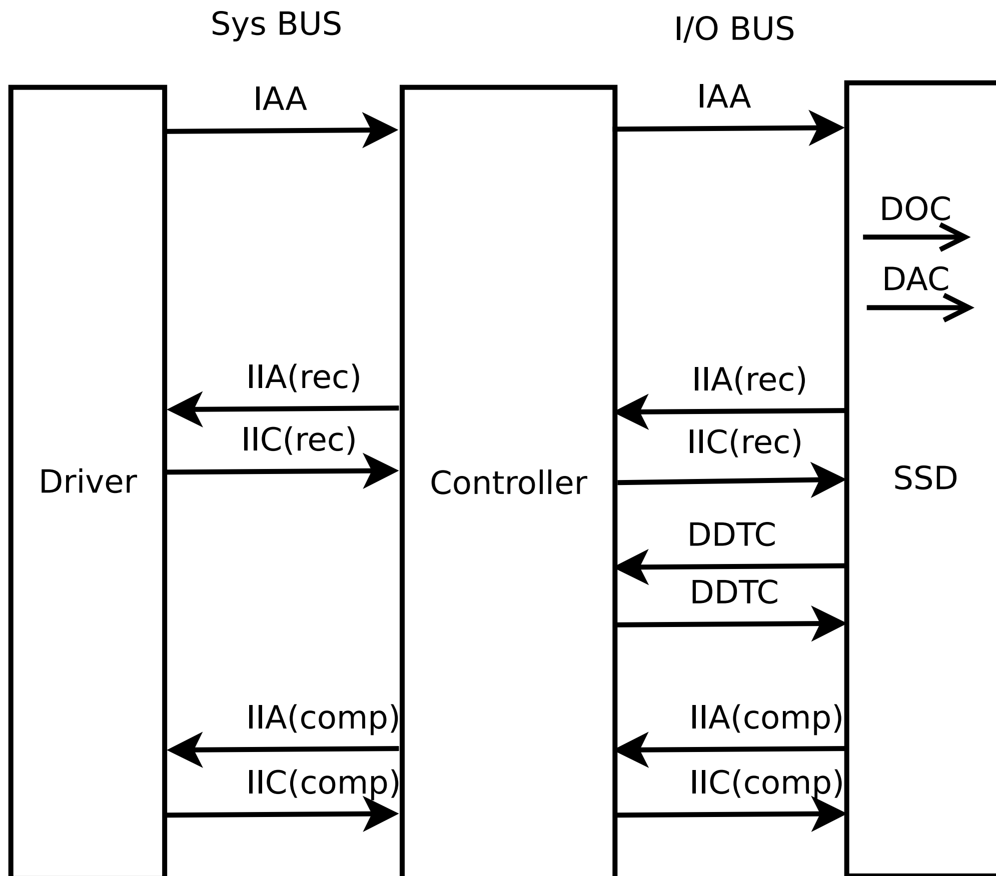
IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
CDTCH: CONTROLLER_DATA_TRANSFER_COMPLETE(host)
CDTCD: DEVICE_DATA_TRANSFER_COMPLETE(device)

Figure A.8: Flowchart for HDD reads (controller type:3)



IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
CDTCH: CONTROLLER_DATA_TRANSFER_COMPLETE(host)
CDTCD: DEVICE_DATA_TRANSFER_COMPLETE(device)
DBS: DISK_BUFFER_SEEKDONE
DBSD: DISK_BUFFER_SECTOR_DONE

Figure A.9: Flowchart for HDD writes (controller type:3)



IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
DOC: DEVICE_OVERHEAD_COMPLETE
DAC: DEVICE_ACCESS_COMPLETE

Figure A.10: Flowchart for SSD reads (controller type:1)

A.3.8 SSD writes with dumb controller (type:1)

The flowchart for SSD writes is shown in Figure A.11. The flowchart is based on the trace to the run of DiskSim with parameter file ssd-postmark.parv. Like reads, the dumb controller passes all messages without processing them upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. Then, it reaches the device (SSD). After DOC, which is an internal event, the SSD issues IIA(reconnect) to reconnect the I/O bus, which is completed upon receiving IIC(reconnect). Then, it issues DDTC to start data transfer, which is done upon receiving DDTC. Next, DAC is done internally in SSD. Finally, the SSD sends IIA(completion) to complete the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

A.3.9 SSD reads with smart controller (type:3)

The flowchart for SSD reads is shown in Figure A.12. The flowchart is based on the trace to the run of DiskSim with parameter file ssd-postmark3.parv, which sets up smart controller (type:3). In this category, the smart controller processes all messages upward or downward. A cycle is started while the device driver gets an IAA message. Next, IAA is passed down to the controller. Then, it reaches the device (SSD). After two internal events (DOC and DAC), the SSD issues DDTC to start data transfer, which is done upon receiving DDTC. The SSD issues IIA(completion) to complete the cycle. Then, Data is transmitted between the device driver and the controller via CDTCH. Finally, the controller issues an IIA(completion) to wrap up the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

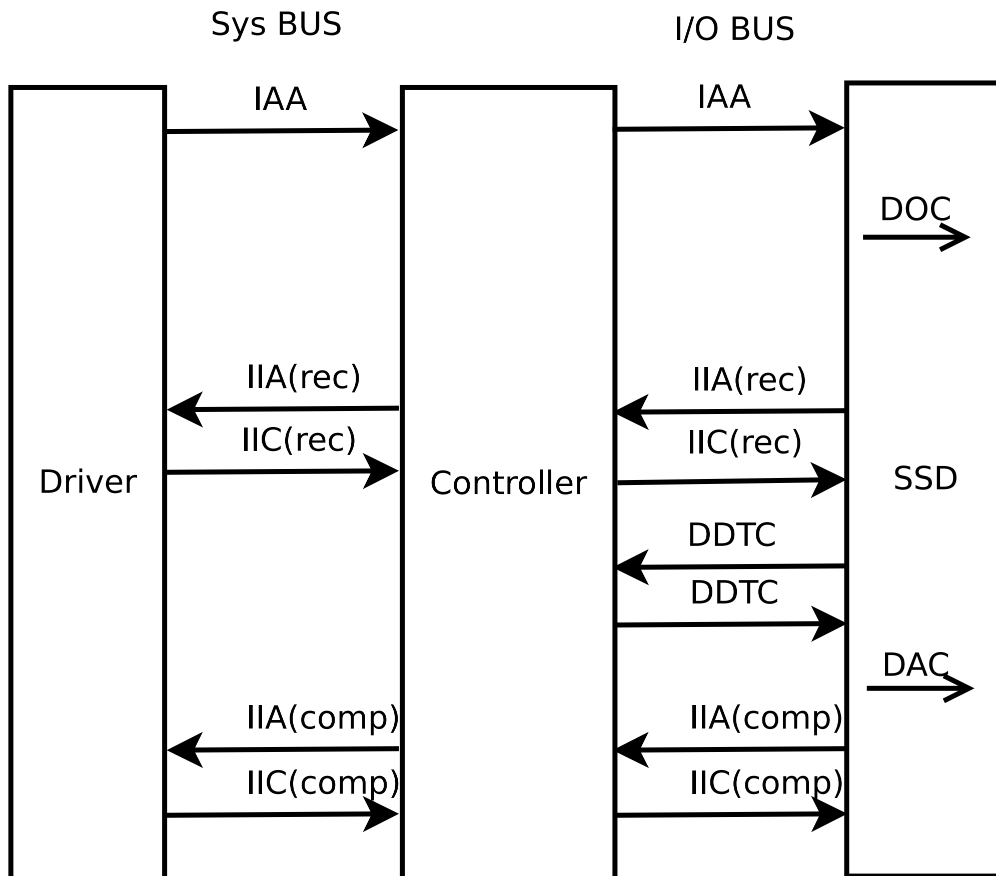
A.3.10 SSD writes with smart controller (type:3)

The flowchart for SSD writes is shown in Figure A.13. The flowchart is based on the trace to the run of DiskSim with parameter file ssd-postmark3.parv, which sets up smart controller (type:3). Like reads, the smart controller processes all messages upward or downward. A cycle is started while the device driver gets an IAA message. IAA is passed down to the controller. Next, data is transmitted between the device driver and the controller via CDTCH. Then, IAA reaches the device (SSD). Next, the SSD issues IIA(reconnect) to reconnect the I/O bus, which is completed upon receiving IIC(reconnect). Unlike dumb controller, smart controller processes IIA and returns IIC. Then, data is transfered via DDTC, which is done upon receiving DDTC. The SSD issues IIA(completion) to complete the cycle, which is done upon receiving IIC(completion). DOC and and DAC are internal events in the SSD. Finally, the controller issues an IIA(completion) to wrap up the cycle. The device driver gets the IIA(completion) and returns an IIC(completion).

A.4 Post Processing Scripts

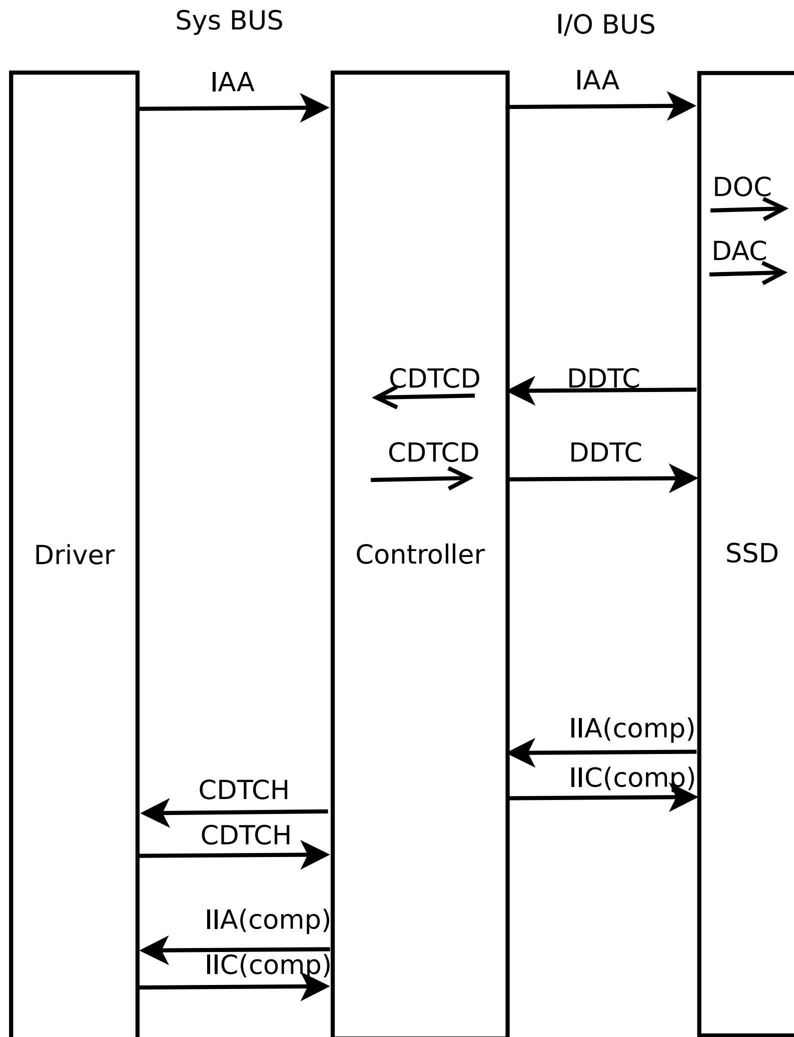
We wrote Bash scripts regarding the run of modified DiskSim 4.0. The scripts have the following features:

1. Run DiskSim 4.0 with different parameters



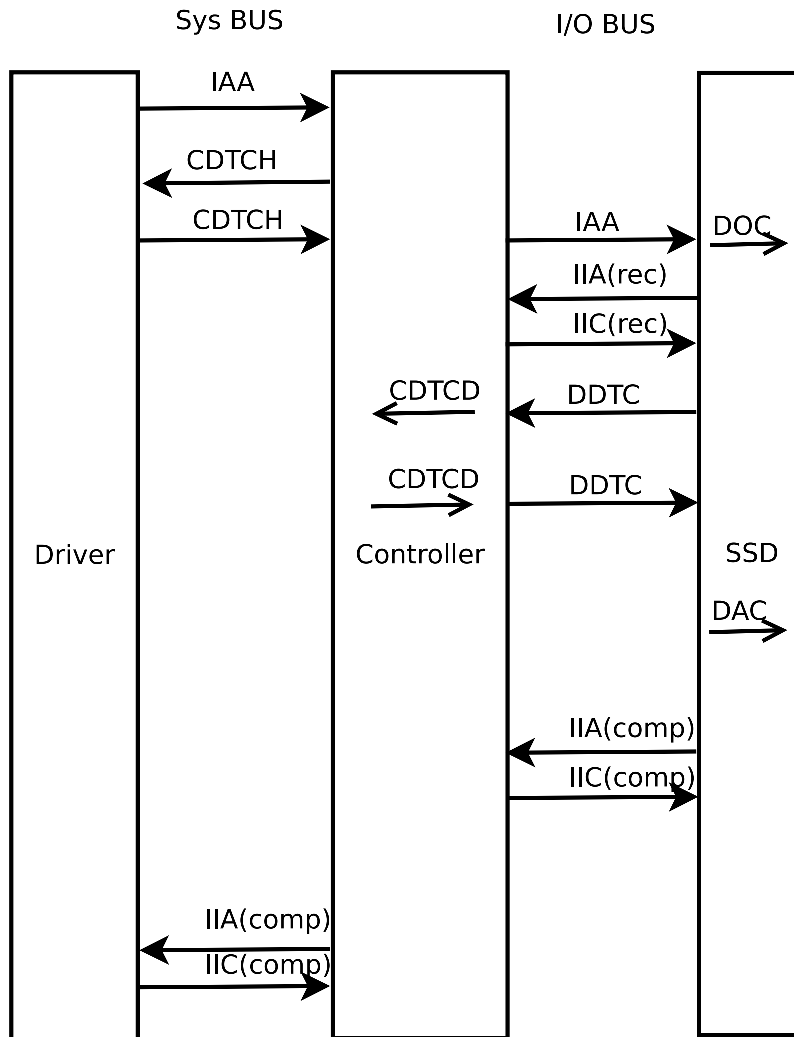
IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
DOC: DEVICE_OVERHEAD_COMPLETE
DAC: DEVICE_ACCESS_COMPLETE

Figure A.11: Flowchart for SSD writes (controller type:1)



IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
CDTCH: CONTROLLER_DATA_TRANSFER_COMPLETE(host)
CDTCD: DEVICE_DATA_TRANSFER_COMPLETE(device)
DOC: DEVICE_OVERHEAD_COMPLETE
DAC: DEVICE_ACCESS_COMPLETE

Figure A.12: Flowchart for SSD reads (controller type:3)



IAA: IO_ACCESS_ARRIVE
IIA: IO_ITERRUPT_ARRIVE
IIC: IO_INTERRUPT_COMPLETE
DDTC: DEVICE_DATA_TRANSFER_COMPLETE
CDTCH: CONTROLLER_DATA_TRANSFER_COMPLETE(host)
CDTCD: DEVICE_DATA_TRANSFER_COMPLETE(device)
DOC: DEVICE_OVERHEAD_COMPLETE
DAC: DEVICE_ACCESS_COMPLETE

Figure A.13: Flowchart for SSD writes (controller type:3)

2. Produce multiple output files
3. Extract corresponding outcomes into files
4. Plot figures based on the above outcomes

The scripts consist of:

- fig_hplajw.sh –main shell
- hplajw.sh –sub shell
- process.sh –outfile processing shell
- miss_ratio.p –gnuplot script for plotting Miss Ratio figure
- res_time.p –gnuplot script for plotting Response Time figure

To work with these scripts, copy all these script files into DiskSim 4.0/valid. Run:

sh fig_hplajw.sh

Figures for miss ratio and response time will be plotted like Figure A.14 and Figure A.15 respectively.

A.4.1 fig_hplajw.sh source code

```
#!/bin/bash
#------------------------------------------------------------
# C.J. Wang
# This script call DiskSim to produce data and plot the figures via gnuplot
# This script assumes that it is under DiskSim4.0/valid.
#------------------------------------------------------------
# variables that need to change based on the traces
#

FileBase=hplajw
SubShell=hplajw.sh
#------------------------------------------------------------
echo Producing "$FileBase.txt (it takes up to 30 seconds)"
echo
sh ./$SubShell > ./$FileBase.txt
echo $FileBase.txt is done
echo
```

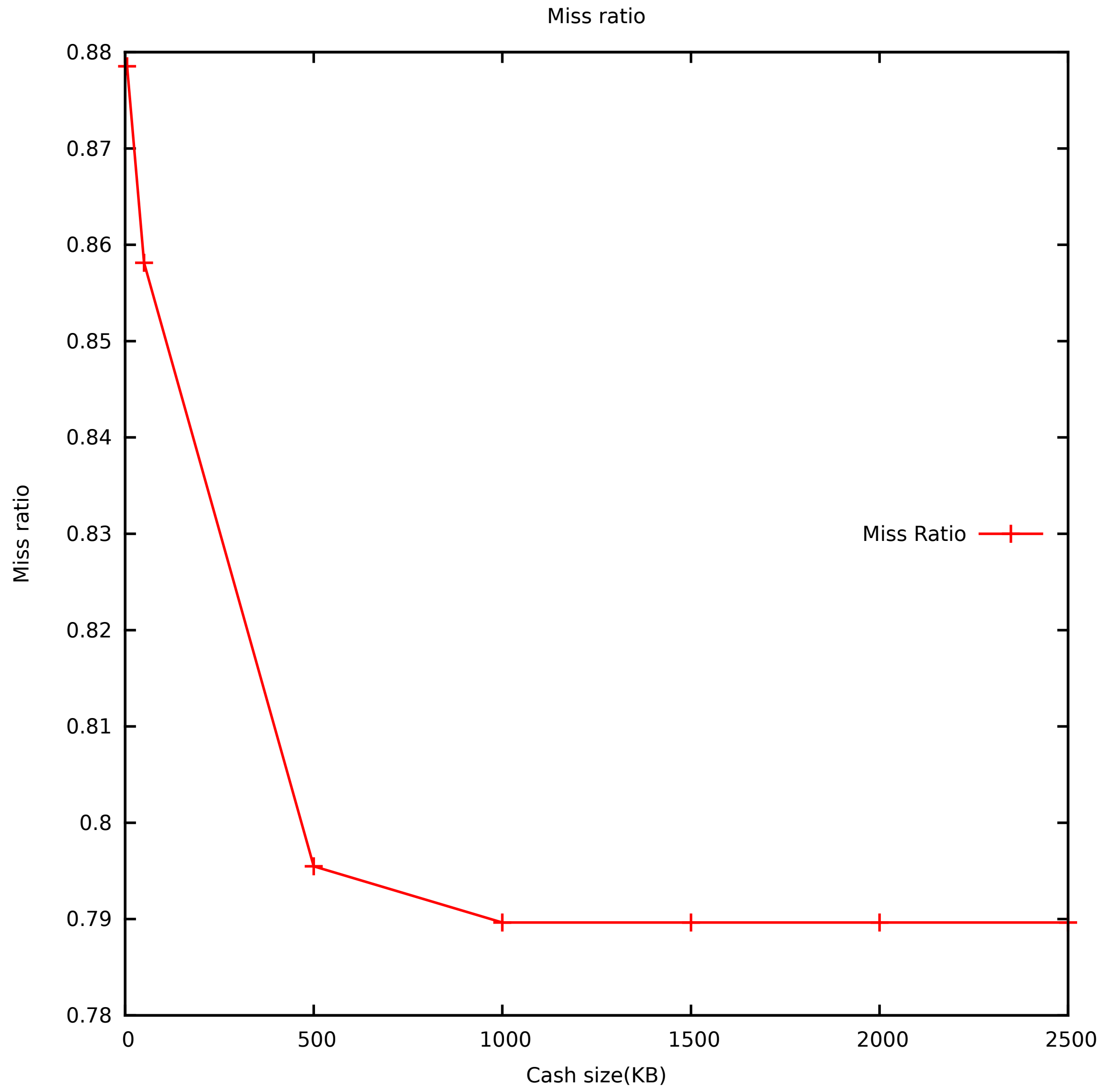


Figure A.14: Miss Ratio for Hplajw workload

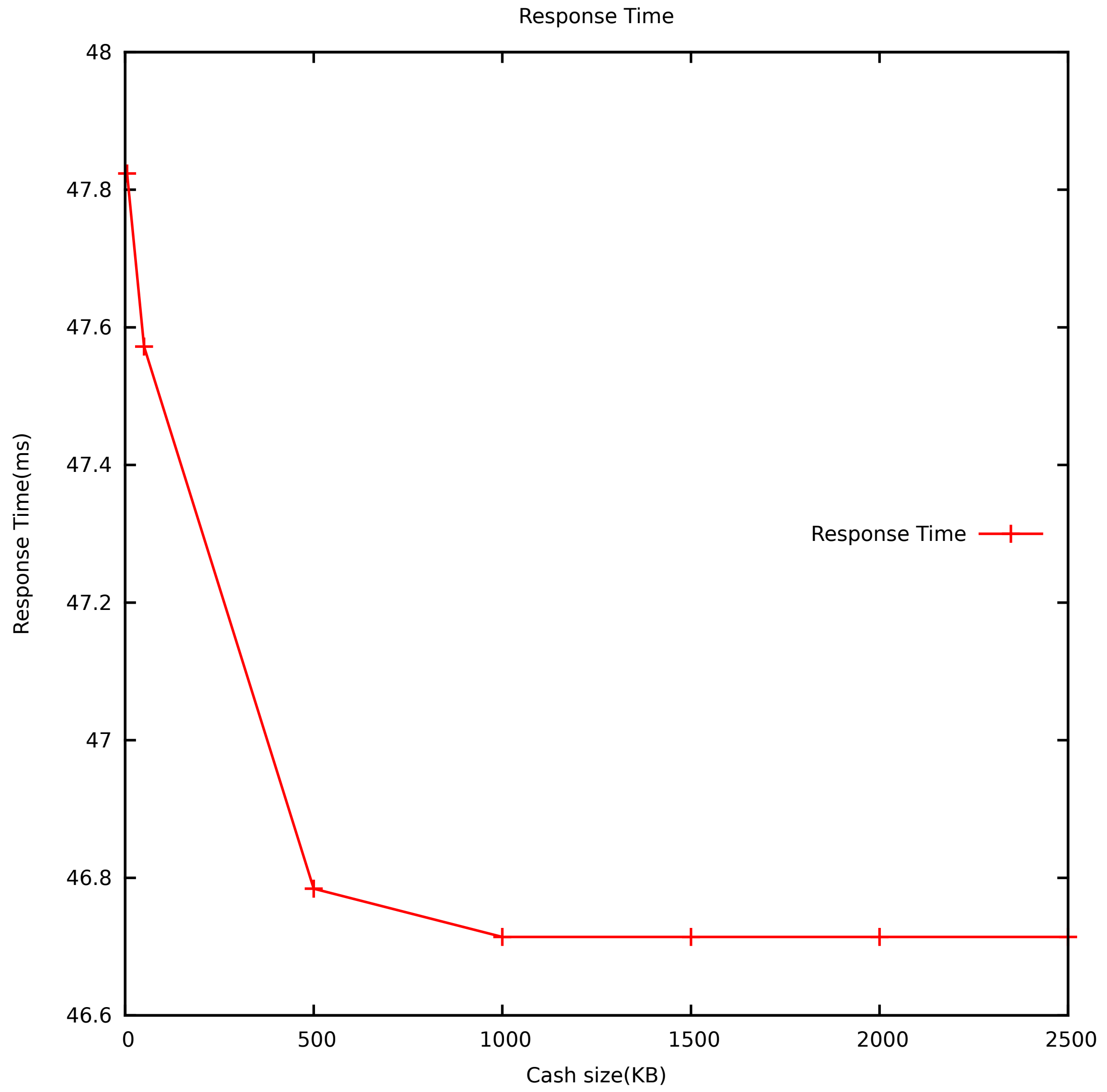


Figure A.15: Response Time for Hplajw workload

```
#gnuplot
echo "Producing figures (ftrfc, fmiss, frestime)"...
echo

Param="call './miss_ratio.p' $FileBase"
gnuplot -e "$Param"
Param="call './res_time.p' $FileBase"
gnuplot -e "$Param"

echo figures are done.
echo
```

A.4.2 hplajw.sh source code

```
#!/bin/bash
#——————————————————————————
# C.J. Wang
#——————————————————————————
# variables that need to change based on traces
#

BASE_OUTFILE=hplajw
PARFILE=hplajw.parv;
TRACEFILE=ajw.1week.srt
TRACEFORMAT=hpl;
CacheSizeRange='10 100 1000 2000 3000 4000 5000'
#——————————————————————————

export OUTFILE
export PARFILE
export TRACEFILE
export TRACEFORMAT
export com_1
export par_1
export val_1
export com_2
export par_2
export val_2
export CacheSize

printf "#Cache Size(KB), MissRatio, Response Time(ms)\n"
```

```
for CacheSize in $CacheSizeRange
do
        com_1="disk*"
        par_1="Number of buffer segments"
        val_1=$CacheSize
        com_2="disk*"
        par_2="Maximum number of write segments"
        val_2=$val_1
        OUTFILE=$BASE_OUTFILE-$CacheSize.outv;
        sh ./process.sh
done
```

A.4.3 process.sh source code

```
#!/bin/bash
#------------------------------------------------------------------
# C.J. Wang
# This script extracts Miss Ratio and Response Time from $OUTFILE
# and print into stdout
#------------------------------------------------------------------
PREFIX=../src
if ! [ -f $OUTFILE ]; then
        $PREFIX/disksim ${PARFILE} ${OUTFILE} ${TRACEFORMAT}\
                ${TRACEFILE-0} ${SYNTH-0}\
                "${com_1}" "${par_1}" ${val_1}\
                "${com_2}" "${par_2}" ${val_2}
fi

MissRatio=`grep "Buffer miss ratio" ${OUTFILE} \
        |head -1 | cut -d: -f2 | cut -f2`

ResTime=`grep "Overall I/O System Response time average" ${OUTFILE} \
        | cut -d: -f2`

CacheSizeInKB=`echo "scale=1; $CacheSize / 2.0" | bc`
printf "$CacheSizeInKB\t\t\t$MissRatio\t\t\t$ResTime\n"
```

A.4.4 miss_ratio.p source code

```
#————————————————————————————-
# C.J. Wang
# gnuplot script file for plotting data in file
# gnuplot 4.4 patch level 0
#————————————————————————————-

set autoscale # scale axes automatically
unset log # remove any log-scaling
unset label # remove any previous labels
set xtic auto # set xtics automatically
set ytic auto # set ytics automatically
set title "Miss ratio"
set xlabel "Cash size(KB)"
set ylabel "Miss ratio"
set key right center

set macros
FileBase="`echo $0`"
FileTxt = sprintf( "%s.txt", FileBase)
OutFile= sprintf( "%s-miss-ratio.pdf", FileBase )

plot FileTxt using 1:2 title 'Miss Ratio' with linespoints

#output to .pdf
set size 1, 1
set terminal pdf enhanced color dashed lw 4 size 6, 6
set output OutFile
replot
```

A.4.5 res_time.p source code

```
#————————————————————————————-
# C.J. Wang # Gnuplot script file for plotting data in file
# gnuplot 4.4 patch level 0
#————————————————————————————-
set autoscale # scale axes automatically
unset log # remove any log-scaling
unset label # remove any previous labels
set xtic auto # set xtics automatically
```

```
set ytic auto # set ytics automatically
set title “Response Time”
set xlabel “Cash size(KB)”
set ylabel “Response Time(ms)”
set key right center

set macros
FileBase=“‘echo $0‘”
FileTxt = sprintf( “%s.txt”, FileBase)
OutFile= sprintf( “%s-res-time.pdf”, FileBase )

plot FileTxt using 1:3 title ’Response Time’ with linespoints

#output to .pdf
set size 1.0, 1.0
set terminal pdf enhanced color dashed lw 4 size 6, 6
set output OutFile
replot
```