Informed Prefetching in Distributed Multi-Level Storage Systems

by

Maen Mahmoud Al Assaf

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
December 12, 2011

Keywords: informed prefetching, pipelining, parallel storage systems, multi-level storage system, distributed storage systems

Approved by

Xiao Qin, Chair, Associate Professor of Computer Science and Software Engineering
David Umphress, Associate Professor of Computer Science and Software Engineering
Wei-Shinn Ku, Assistant Professor of Computer Science and Software Engineering

Abstract

In this dissertation, we present pipelined prefetching mechanisms that use application-disclosed access patterns to prefetch hinted blocks in multi-level storage systems. The fundamental concept in our approach is to split an informed prefetching process into a set of independent prefetching steps among multiple storage levels (e.g., main memory, solid state disks, and hard disk drives). In the first part of this study, we show that a prefetching pipeline across multiple storage levels is an viable and effective technique for allocating file buffers at the multiple-level storage devices. Our approaches (a.k.a., iPipe and IPO) extends previous ideas of informed prefetching in two ways: (1) our approach reduces applications' I/O stalls by keeping hinted data in caches residing in the main memory, solid state disks, and hard drives; (2) we propose a pipelined prefetching scheme in which multiple informed prefetching mechanisms semi-dependently work to fetch blocks from low-level (slow) to high-level (fast) storage devices. Our iPipe and IPO strategies integrated with the pipelining mechanism significantly reduce overall I/O access time in multiple-level storage systems. Next, we propose a third prefetching scheme called IPODS that aims to maximize the benefit of informed prefetches as well as to hide network latencies in a distributed storage systems. Finally, we develop a simulator to evaluate the performance of the proposed informed prefetching schemes in the context of multiple-level storage systems. We implement a prototype to validate the accuracy of the simulator. Our results show that our iPipe improves system performance by 56% in most informed prefetching cases, IPO and IPODS improve system performance by 56% and 6% respectively in informed prefetching critical cases across a wide range of real-world I/O traces.

Acknowledgments

I owe my gratitude to all the people who have made this work possible and who supported me during my stay at Auburn University.

I would like to express my deepest gratitude to my advisor, Dr. Xiao Qin. I was indeed fortunate to be his student. He educated me and gave me support, experience, encouragement, and guidance. He taught me good research methodologies, team work, and writing skills. He also did his best to help me when I face obstacles. Without his help and support, this work would not have been possible. I really hope to be a good advisor like Dr. Qin for my future students.

I am grateful and thankful for Dr. David Umphress who reviewed my proposal and dissertation. He alerted me to many important points concerning good writing skills and research. I am pleased to be his student.

I would like to express my sincere thanks for Dr. Wei-shinn Ku for reviewing my proposal and dissertation. Dr. Ku indeed supported me in my research and guided me on the right track. I am really grateful and thankful for him.

I would also like to acknowledge Dr. Guofu Niu from the Department of Electrical and Computer Engineering at Auburn University for reviewing my dissertation and supporting my research. I really thank him very much.

This work was discussed with my colleagues in Dr. Xiao Qin's research group. I would like to mention in particular Xiaojun Ruan, Shu Yin, Yun Tian, Zhiyang Ding, James Majors , Jiong Xie, Yixian Yang, Ji Zhang, Joshua Lewis,and Jianguo Lu. I really thank them very much.

I am indebted to the University of Jordan who sponsored me during my stay in the United States. In particular, I would like to thank my colleagues in King Abdullah II School of IT for their support and advise.

Most importantly, I thank my family who encouraged me to continue my Ph.D education. Without their patience and love, this work would not have been possible. I dedicate this dissertation to them.

Table of Contents

Abstract ii

Acknowledgments iii

List of Figures x

List of Tables xiv

1 Introduction 1

1.1 Problem Statement 1

1.1.1 Informed Prefetching for Multiple-Level Storage Systems 2

1.1.2 Prefetching in Multi-Level Storage Systems with Limited I/O Bandwidth 3

1.1.3 Informed Prefetching for Distributed Multi-level Storage Systems . . 3

1.2 Motivations 4

1.2.1 Motivation 1: the growing needs of multi-level storage systems. . . . 4

1.2.2 Motivation 2: the I/O access hints offered by applications. 4

1.2.3 Motivation 3: multiple prefetching mechanisms perform in parallel. . 5

1.3 Contributions 6

1.4 Dissertation Organization 7

2 Literature Review & Current Work 8

2.1 Storage systems current work 8

2.1.1 Multi-level storage systems 8

2.1.2 Solid State Disks and Hard Drives 10

2.1.3 Parallel Storage Systems 11

2.1.4 Distributed Storage Systems 12

2.2 Prefetching 13

2.2.1 Two Types of Prefetching 13

2.2.2 Informed Prefetching 14
2.2.3 Predictive Prefetching 15
2.2.4 Prefetching in multi-level storage systems 16
2.2.5 Prefetching in Distributed and Parallel Storage Systems 17
2.3 Summery 18
3 Assumptions and Parameters Validations 20
3.1 System Assumptions 20
3.1.1 Demand Misses 20
3.1.2 Informed Caching 21
3.1.3 Bandwidth Limitations 22
3.1.4 I/O Bandwidth Sharing 22
3.1.5 Life Time of Solid State Disk (SSD) 23
3.2 Assumptions on Prefetching and Pipelining 23
3.2.1 Initial Data Allocation 23
3.2.2 A Pipeline for Data Transfers 24
3.2.3 Writes and Data Consistency 25
3.2.4 Pipelining Depth 26
3.2.5 Block Size 26
3.2.6 LASR Traces 26
3.3 Validations 27
3.3.1 Overview 27
3.3.2 System Setup 28
3.3.3 Proof of the Concept 29
3.3.4 Block Size 31
3.3.5 Model $T_{cpu} + T_{hit} + T_{driver}$ 31
3.3.6 System Parameters validation 35
3.3.7 Limited Parallel I/O Bandwidth 36

3.4 Summary . . . 39
4 iPipe: An Pipelined and Informed Prefetching for Multi-Level Storage Systems . 41
4.1 Overview . . . 41
4.2 Motivations and Objectives . . . 41
4.3 Design Issues in iPipe . . . 42
4.3.1 Architecture of iPipe . . . 43
4.3.2 Assumptions . . . 45
4.4 The iPipe Algorithm . . . 46
4.4.1 Stalls and Disk Read Latencies . . . 46
4.4.2 Prefetching Horizon . . . 48
4.4.3 The Pstart and Pdepth Algorithms . . . 48
4.4.4 Stalls, Elapsed Time, Prefetching Horizon, and Prefetching Benefit . 53
4.4.5 The iPipe Algorithm . . . 57
4.5 Performance Evaluation . . . 58
4.5.1 System Setup . . . 58
4.5.2 Preliminary Results . . . 59
4.5.3 Validated Performance Evaluation . . . 67
4.6 Summary . . . 73
5 IPO: Informed Prefetching Optimization in Multi-level Storage Systems . . . 74
5.1 Overview . . . 74
5.2 Motivations and Objectives . . . 74
5.3 Design Issues in IPO . . . 75
5.3.1 Architecture of IPO . . . 76
5.3.2 Assumptions . . . 79
5.4 The IPO Algorithm . . . 79
5.4.1 Definitions . . . 79
5.4.2 The Pstart and Pnext Algorithms . . . 81

5.4.3 The IPO Algorithm . . . 83
5.5 Performance Evaluation . . . 88
5.5.1 Simulation Environment . . . 88
5.5.2 Elapsed Time Improvement . . . 88
5.5.3 Bandwidth Utilization . . . 91
5.5.4 Increasing the Max_{BW} Value . . . 91
5.6 Summary . . . 92
6 IPODS: Informed Prefetching in Distributed Multi-level Storage Systems . . . 94
6.1 Overview . . . 94
6.2 Motivations and Objectives . . . 95
6.3 IPODS Design Issues . . . 96
6.3.1 The IPODS Architecture . . . 97
6.3.2 Assumptions . . . 99
6.4 The IPODS Algorithm . . . 99
6.4.1 Definitions . . . 99
6.4.2 The IPODS Algorithm . . . 101
6.5 Performance Evaluation . . . 102
6.5.1 Simulation Environment . . . 102
6.5.2 Improving Elapsed Time . . . 103
6.6 Summary . . . 104
7 Prototype Development . . . 105
7.0.1 Objectives . . . 105
7.0.2 System Setup . . . 106
7.0.3 Design Issues of the Prototypes . . . 106
7.0.4 Validation Process . . . 107
7.1 Prototypes . . . 108
7.1.1 The iPipe Prototype . . . 108

7.1.2 The IPO Prototype . . . 115
7.1.3 The IPODS Prototype . . . 119
7.2 Summary . . . 120
8 Conclusions & Future Work . . . 123
8.1 Main Contributions . . . 123
8.1.1 iPipe: An Pipelined and Informed Prefetching . . . 124
8.1.2 IPO: Informed Prefetching Optimization . . . 125
8.1.3 IPODS: Pipelined Prefetching in Distributed/Parallel Storage Systems 126
8.1.4 Prototypes for iPipe, IPO, and IPODS . . . 126
8.2 Future Work . . . 127
8.2.1 Data Migration . . . 127
8.2.2 The Cost-Benefit Model . . . 128
8.2.3 Write Performance . . . 128
8.2.4 Most Recently Used Policy . . . 129
8.2.5 Extending Storage Hierarchy . . . 129
8.2.6 Caching and Benchmarking . . . 130
8.2.7 Various Solid State Disks . . . 130
8.2.8 Block Sizes . . . 130
8.2.9 Fast Networks for Distributed Storage Systems . . . 131
Bibliography . . . 132

List of Figures

2.1 Multi-level storage system that consists of different storage devices with various speed performance. 9

3.1 Read Latency of HDDs and SDDs. When block size is 10 MB, SSD has better read performance than HDD. 30

3.2 Read Latency of accessing a remote HDD and SDD through the LAN network connection. In this distributed system setting, SSD has better read performance than HDD when block size is 200 MB. 30

3.3 (T_{cpu}) + (T_{hit}) + (T_{driver}) values range from 0.037- 0.104 seconds. We will consider the smallest value. 35

3.4 10 MB: Estimated T_{hdd-ss} is 0.122 seconds. 36

3.5 10 MB: Estimated $T_{hdd-cache}$ is 0.12 seconds. 37

3.6 10 MB: Estimated $T_{ss-cache}$ is 0.052 seconds. 37

3.7 200 MB: Estimated T_{hdd-ss} is 4.5 seconds. 38

3.8 200 MB: Estimated $T_{hdd-cache}$ is 2.3 seconds. 38

3.9 200 MB: Estimated $T_{ss-cache}$ is 1.5 seconds. 39

3.10 Limited Parallel I/O Bandwidth. Concurrent read requests are noticeably affected by the limited parallel I/O bandwidth. 40

4.1 iPipe system hardware architecture. Consists of an array of multi-level disks. . . 43

4.2 High-level design of the iPipe software architecture for a multi-level storage system. The multi-level storage system consists of two levels - SSDs and HDDs. . . 44

4.3 Detailed design of the iPipe software architecture for a three-level storage system. An application provides hints to both TIP and iPipe. The system performance parameters are passed to iPipe to calculate the pipelining starting block and depth. iPipe keeps fetching hinted data blocks to the highest level. Since the storage system's bandwidth is high enough, iPipe is able to fetch most of the hinted blocks. TIP uses a cost benefit model to determine the number of prefetch buffers. Hinted data blocks are fetched from the storage system to the buffer cache. 45

4.4 Average stall when using iPipe and a fixed number of buffers for pipelined prefetching. $T_{hdd-cache} = 5$, $T_{hdd-ss} = 8$, $T_{ss-cache} = 4$, $X_{cache} = 3$. The first stall is 5 time units. Before the first hinted block is fetched from HDD into SSD, the application stalls for $T_{stall-hdd}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver})$ = 2 time units every 3 accesses. When hinted blocks are retrieved in the SSD, the application stalls for $T_{stall-ss}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver})$ = 1 every 3 accesses. 55

4.5 Average stall when using a fixed number of buffers for pipelined prefetching. $T_{hdd-cache} = 5$, $X_{cache} = 3$. The first stall is for 5 time units, because all data blocks are read from the HDD. The application stalls for $T_{stall-hdd}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 2$ time units every 3 accesses. 56

4.6 Total elapsed time when the number of prefetching buffers is set from 1 to 11. iPipe reduces the elapsed time. 61

4.7 Total elapsed time when the number of prefetching buffers is set from 13 to 26. iPipe reduces the elapsed time. 62

4.8 Total informed prefetching read latency. iPipe reduces the read latency. 62

4.9 Total stall time when the number of prefetching buffers is set from 1 to 9. iPipe reduces the stall time. 64

4.10 Total stall time when the number of prefetching buffers is set from 11 to 19. iPipe reduces the stall time. 64

4.11 Total elapsed time when the number of prefetching buffers is set from 1 to 5. iPipe reduces the elapsed time in both Nova-V64 and Intel X25-E SSDs. Elapsed time is less when Intel X25-E SSD is tested because it is faster than Nova-V64 SSD. 65

4.12 Total elapsed time when the number of prefetching buffers is set from 7 to 11. iPipe reduces the elapsed time in both Nova-V64 and Intel X25-E SSDs. Elapsed time is less when Intel X25-E SSD is tested because it is faster than Nova-V64 SSD. 66

4.13 Total elapsed time when the number of prefetching buffers is set from 13 to 26. iPipe reduces the elapsed time in both Nova-V64 and Intel X25-E SSDs. Elapsed time is less when Intel X25-E SSD is tested because it is faster than Nova-V64 SSD. 66

4.14 Block size = 10 MB. Total elapsed time when the number of buffers is set from 1 to 9. iPipe reduces the elapsed time. 69

4.15 Block size = 10 MB. Total elapsed time when the number of buffers is set from 11 to 25. iPipe reduces the elapsed time. 69

4.16 Block size = 10 MB. Total elapsed time when the number of buffers is set from 35 to 63. iPipe reduces the elapsed time. 70

4.17 Block size = 200 MB. Total elapsed time when the number of buffers is set from 1 to 9. iPipe reduces the elapsed time. 70

4.18 Block size = 200 MB. Total elapsed time when the number of buffers is set from 11 to 25. iPipe reduces the elapsed time. 71

4.19 Block size = 200 MB. Total elapsed time when the number of buffers is set from 35 to 63. iPipe reduces the elapsed time. 71

5.1 High-level design of IPO software architecture. The multi-level storage system consists of an array of two levels of SSDs and HDDs with limited bandwidth and scalability. 78

5.2 Detailed design of IPO software architecture for a three-level storage system. The application provides hints on future I/O accesses. IPO determines the appropriate hinted blocks to be fetched. IPO keeps prefetching a particular number (depends on available bandwidth) of hinted blocks to the uppermost level. The upper-level prefetcher (i.e., TIP) uses the cost/benefit model to determine the number of prefetching buffers. 78

5.3 Average stalls when using IPO and a fixed number of buffers for parallel prefetching in the buffer cache. The maximum number (Max_{BW}) of read requests is 5. Informed prefetching buffers = 2, and the rest 3 spaces of the bandwidth are used for pipelined prefetching. $T_{hdd-cache} = 5$, $T_{hdd-ss} = 8$, $T_{ss-cache} = 4$, and $X_{cache} = 2$. The first accesses stall for 5 time units. Before IPO fetches hinted blocks from HDD to SSD, the application stalls for $T_{stall-hdd}(\mathrm{X}_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 2$ time units every 3 accesses. IPO continues to fetch 3 hinted blocks each time from HDD. When a prefetched block is consumed from SSD, a new pipelined prefetching request is initiated by IPO. When IPO is employed, stalls time becomes 40 and elapsed time is 76 time units. 84

5.4 Continue Figure 5.3 85

5.5 Average stalls when using a fixed number of buffers for parallel prefetching in buffer cache. $\mathrm{T}_{hdd-cache} = 5$, and $\mathrm{X}_{cache} = 2$. The first accesses stall for 5 time units. All data is read from HDD. The application stalls for $\mathrm{T}_{stall-hdd}(\mathrm{X}_{cache})$ = $\mathrm{T}_{hdd-cache}$ - $3(\mathrm{T}_{cpu} + \mathrm{T}_{hit} + \mathrm{T}_{driver}) = 2$ time units every 3 accesses. In the non-IPO case, stalls time is 45 and elapsed time is 81. 86

5.6 Continue Figure 5.5 87

5.7 IPO reduces application elapsed time. 10 MB block size. Total elapsed time when using 1 to 15 prefetching buffers. $\mathrm{Max}_{BW} = 15$. 89

5.8 IPO reduces application elapsed time. 200 MB block size. Total elapsed time when using 1 to 7 prefetching buffers. $\text{Max}_{BW} = 15$. 90

5.9 IPO reduces application elapsed time. 200 MB block size. Total elapsed time when using 9 to 15 prefetching buffers. $\text{Max}_{BW} = 15$. 90

5.10 Bandwidth utilization when X_{cahce} is varied from 1 to 15 in both IPO and non-IPO cases. IPO fully utilizes the bandwidth. 91

5.11 IPO reduces the elapsed time. Total elapsed time when the X_{cahce} value is increased from 1 to 63. Max_{BW} is set to 154. 92

6.1 The architecture of a distributed parallel storage system. Several distributed clients and storage nodes are connected by a network. T.Madhyastha; G. Gibson; C. Faloutsos: Informed prefetching of collective input/output requests, Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), Portland, Oregon, 1999. 95

6.2 [71] Another architecture of a distributed parallel storage system. Several distributed clients and storage nodes are connected by a network. Luis Cabrera , Darrell D.E. Long: SWIFT: USING DISTRIBUTED DISK STRIPING TO PROVIDE HIGH I/O DATA RATES, University of California at Santa Cruz, Santa Cruz, CA, 1991. 96

6.3 Distributed/Parallel Multi-level Storage System: The system shows several distributed storage nodes and clients connected by a network. Each I/O storage node consists of a two-level storage device containing both SSD and HDD. T.Madhyastha; G. Gibson; C. Faloutsos: Informed prefetching of collective input/output requests, Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), Portland, Oregon, 1999. 98

6.4 Total elapsed time when the number of prefetching buffers is varied from 1 to 15. $\text{Max}_{BW} = 15$. IPODS reduces the elapsed time. 104

List of Tables

3.1 I/O bandwidth measured using the Ramspeed benchmark. 33

4.1 The number of informed prefetching requests issued to HDDs when the LASR1 and LASR2 traces are evaluated. 60

4.2 Service time is reduced when one extra buffer is added for prefetching. 61

4.3 Prefetching Horizon P(T_{cpu}) equals 26 data blocks distance while iPipe is not used. P(T_{cpu}) drops to 20 data blocks distance when iPipe is used. 63

4.4 Total stall time when the number of prefetching buffers is set from 21 to 26. iPipe reduces the stall time. 65

4.5 Data block size = 10 MB. The position of the first data block to be prefetched. 72

4.6 Data block size = 10 MB. The depth of the pipelined prefetching when using different X_{cahce} values. Small depth is needed when few X_{cahce} buffers are used. The maximum depth = 91. iPipe needs to assign the maximum depth for pipelined prefetching starting from X_{cahce} = 35, because the reading stalls from SSD at that point = 0 72

4.7 Data block size = 200 MB. The position of the first data block to be prefetched. 73

4.8 Data block size = 200 MB. The depth of the pipelined prefetching when using different X_{cahce} values. Small depth is needed when few X_{cahce} buffers are used. The maximum depth = 163. iPipe needs to assign the maximum depth for pipelined prefetching starting from X_{cahce} = 45, because the reading stalls from SSD at that point = 0 73

7.1 Total elapsed time measured in seconds when the iPipe prototype is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 10 MB. 111

7.2 Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe prototype. The number of prefetching buffers is set to a range between 1 to 63. The block size is 10 MB. 111

7.3 Total elapsed time measured in seconds when the iPipe simulator is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 10 MB. 112

7.4 Comparison between iPipe's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 10 MB. 112

7.5 Total elapsed time measured in seconds when the iPipe prototype is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 200 MB. 113

7.6 Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe prototype. The number of prefetching buffers is set to a range between 1 to 63. The block size is 200 MB. 113

7.7 Total elapsed time measured in seconds when the iPipe simulator is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 200 MB. 114

7.8 Comparison between iPipe's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 200 MB. 114

7.9 Total elapsed time measured in seconds when the IPO prototype is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 10 MB. 116

7.10 Total elapsed time measured in seconds when the LASR traces are replayed by the IPO prototype. The number of prefetching buffers is set to a range between 1 to 15. The block size is 10 MB. 116

7.11 Total elapsed time measured in seconds when the IPO simulator is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 10 MB. 116

7.12 Comparison between IPO's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 10 MB. 117

7.13 Total elapsed time measured in seconds when the IPO prototype is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 117

7.14 Total elapsed time measured in seconds when the LASR traces are replayed by the IPO prototype. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 117

7.15 Total elapsed time measured in seconds when the IPO simulator is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 118

7.16 Comparison between IPO's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 200 MB. 118

7.17 Total elapsed time measured in seconds when the IPODS prototype is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 120

7.18 Total elapsed time measured in seconds when the LASR traces are replayed by the IPODS prototype. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 120

7.19 Total elapsed time measured in seconds when the LASR traces are replayed by the IPODS simulator. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 121

7.20 Total elapsed time measured in seconds when the IPODS simulator is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB. 121

Chapter 1

Introduction

To solve the I/O bottleneck problem (see [81]) in large-scale computing systems, researchers have proposed a wide range of prefetching techniques for preloading data from disks into the main memory prior to the data accesses. Existing prefetching techniques can be categorized into two camps - predictive prefetching and informed prefetching. Predictive prefetching schemes predict future I/O access patterns based on historical I/O accesses of applications [2], whereas informed prefetching techniques make preloading decisions based on applications' future access hints [1]. In this dissertation study, we will focus on informed prefetching schemes and investigate performance impact of informed prefetching on multiple level storage systems. Well-known predictive prefetching solutions are summarized in Section 2.2.

The chapter is organized as follows. The first section describes the problem statement of this dissertation study. In the second section, we illustrate the important motivations for our new proposed informed prefetching approaches. Next, the second section outlines the main contributions of this study. Finally, the last section specifies the organization of the dissertation.

1.1 Problem Statement

As the performance gap between processors and I/O subsystems increases rapidly, disk performance becomes a serious bottleneck for large-scale computing systems supporting data-intensive applications [32]. Recent studies show that informed prefetching can bridge the performance gap between the CPU and I/O; for example, an informed prefetching algorithm called TIP proposed by Patterson *et al.* aims to improve performance of I/O-intensive

applications by applying cost-benefit analysis to allocate buffers for both prefetching and caching [1]. TIP's cost-benefit analysis is possible, because TIP estimates the impact of alternative buffer allocations on application execution time.

This dissertation research is inspired by the TIP approach [1]. Follows are three main challenges to be addressed our study:

1. informed prefetching for multiple-level storage systems,

2. prefetching in multi-level storage systems with limited I/O bandwidth, and

3. informed prefetching for distributed multi-level storage systems.

1.1.1 Informed Prefetching for Multiple-Level Storage Systems

When disk arrays are employed, the TIP algorithm improves the quality of parallel prefetching through accurately eliminating I/O stalls. In Patterson's approach, the informed caching and prefetching algorithm assigns a portion of buffer space used for demand caching (LRU) and the rest of the buffer to store hinted blocks prefetched from disks [1]. This buffer allocation process is guided by a cost-benefit model. When data accessing time increases due to high disk load, I/O stall time goes up and creates an increasing benefit of assigning additional buffers for hinted blocks. When parallel disk subsystems are extended into multiple-level storage systems [52] [23], a hierarchy of multiple storage devices increases data access latency if the data are residing in a lower level of the systems. To shorten long data transfer latency, popular data or future accessed data may be stored in the upper level of the storage systems.

Traditional informed prefetching schemes can hide the latency of accessing storage systems by invoking disk I/O parallelisms and fetching data based on application-disclosed hints. We will show that building an informed prefetching pipeline can significantly improve the I/O performance of multi-level storage systems. We will illustrate how to use application hints to initiate prefetching among multiple storage levels like main memory, solid

state disks, and hard disk drives. The centerpiece of our approach is a pipeline in which we split the informed prefetching process into a set of independent prefetching steps among the multiple storage levels. In particular, we will demonstrate how to integrate this pipeline with informed prefetching and caching to manage file buffers at various storage levels.

1.1.2 Prefetching in Multi-Level Storage Systems with Limited I/O Bandwidth

I/O-intensive applications can disclose hints about their future I/O accesses and these hints can be used to guide prefetching mechanisms in making accurate prefetching decisions. Existing informed prefetching algorithms rely on the assumption that parallel disks offer enough I/O bandwidth for prefetching without encountering I/O congestion. Under such an assumption, an informed prefetching mechanism can prefetch a large number of data blocks in parallel; Unfortunately, our preliminary results show that real-world storage systems may not have unlimited I/O bandwidth; this observation is especially true for small-scale storage systems. We addressed this issue by developing an informed prefetching algorithm (see Chapter 5) in a multiple-level storage system where disk devices offer limited I/O bandwidth. Our informed prefetching solution is practical, because it does not rely on the assumption that storage systems provide unlimited I/O bandwidth.

1.1.3 Informed Prefetching for Distributed Multi-level Storage Systems

To further extend our prefetching approaches (see Chapters 4 and 5), we developed an informed prefetching algorithm (see Chapter 6) tailored for distributed multi-level storage systems, each of which consists of a group of multi-level storage servers. In a distributed storage system, large disk access latency due to network delays can be hidden by informed prefetching. We will demonstrate that a pipeline mechanism can be used to efficiently prefetch data blocks from a low-level storage device to a up-level storage device before moving the data blocks to the clients.

1.2 Motivations

The following key factors motivated us to investigate pipelined informed prefetching:

1. the growing needs of multi-level storage systems,
2. the I/O access hints offered by applications, and
3. the possibility of multiple prefetching mechanisms working in parallel.

1.2.1 Motivation 1: the growing needs of multi-level storage systems.

Multi-level storage systems have been widely employed in data centers supporting service-based applications such as multimedia streaming and scientific computing. For example, popular data are fetched and cached in an upper-level server while massive amounts of unpopular data are placed in lower-level storage servers. Overall performance of I/O-intensive applications can be improved by increasing the I/O performance of multi-level storage systems. Existing studies (see, for example, [19] and [9]) suggest that new prefetching and caching techniques are needed to boost the I/O performance of multi-level storage systems. Since prefetching must be performed at each storage level to hide I/O latencies, prefetching mechanisms at multi-level need to coordinate in order to achieve high prefetching efficiency.

1.2.2 Motivation 2: the I/O access hints offered by applications.

The second factor motivating our informed prefetching for multi-level storage systems is that largely predictable I/O access patterns can be disclosed by applications as hints. I/O access hints are used by prefetching mechanisms to invoke asynchronous I/O accesses to fetch data to upper-level storage like the main memory and solid state disks.

Informed prefetching in multi-level storage systems is challenging for two main reasons. First, hints must be processed at different storage levels in different manners. For example, an

upper-level prefetching mechanism takes hints and makes conservative prefetching decisions for upper-level storage with small capacity; a lower-level prefetching mechanism is aggressive in order to effectively use large lower-level storage as a staging area. Second, at each storage level, an informed prefetching manager must balance cache space against prefetching space. Informed prefetching mechanisms at multiple levels should coordinate to manage caches across multiple levels of storage devices.

Prefetching and caching issues in multi-level storage systems have been investigated in existing studies [19] [9]. However, to our best knowledge no research has incorporated informed prefetching mechanisms into a multi-level storage system.

1.2.3 Motivation 3: multiple prefetching mechanisms perform in parallel.

Multiple informed prefetching mechanisms can independently fetch blocks from lower-level to upper-level storage devices. These prefetching mechanisms can work in parallel, because multiple prefetching operations can be simultaneously processed by the upper-level and lower-level prefetching mechanisms. Such storage parallelisms make it possible to implement a prefetching pipeline, where the informed prefetching processes are separated into a set of distinct prefetching steps among multiple storage levels.

When data blocks have to be cached in all of the storage levels, significant I/O delays are incurred by prefetching data from the lowest-level to the highest-level of storage. We show that a pipelined prefetching approach can increase prefetching throughput, which is defined as the number of blocks prefetched from the lowest storage level to the highest storage level.

We will focus on read-intensive applications, because read performance of applications is poor when the data to be accessed are residing in the lower-level storage subsystems. Write performance in multi-level storage systems is not as critical as read performance, since applications can write data to upper-level storage devices (e.g., write-behind buffer) before moving data from the upper-level to the lower-level storage.

1.3 Contributions

The following list summarizes the major research contributions made in this dissertation study:

- To reduce I/O delays in multi-level storage systems, we propose new informed prefetching approaches to coordinating multiple prefetching mechanisms in the form of a pipeline. The three informed prefetching algorithms developed in this study are called iPipe (see Chapter 4), IPO (see Chapter 5), and IPODS (see Chapter 6). We show that with our prefetching pipeline in place, multiple prefetching operations can be processed in parallel by both upper-level and lower-level prefetching mechanisms.

- We apply a novel cost-benefit model to estimate the value of prefetching or caching a block at a specific storage level in a multi-level storage system. The cost-benefit model is used by prefetching mechanisms deployed in multiple storage levels to improve buffer usage at all storage levels. We describe how we employed the model in our prefetching algorithms in Chapters 4-6.

- We developed a simulated multiple-level storage system, in which the three prefetching algorithms are implemented. Simulation results show that our prefetching mechanism powered by a data fetching pipeline reduces applications' stall and execution times. Simulation results also indicate that our approaches can reduce prefetching distance (a.k.a., prefetching horizon).

- We implemented a prototype to validate our simulator for multi-level storage systems. Like the simulator, the prototype contains the implementation of all the three proposed informed prefetching algorithms. We conducted an experiment using the prototype, showing that our prefetching solutions can reduce disk read latency in distributed multi-level storage system, an environment in which data are stored in different nodes that are connected by a network. Prefetching time will increase because of network

and server latencies. Pipelining can only reduce the disk read latency as a part of the total prefetching time.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows:

Chapter 2 describes the cutting-edge-research in multi-level storage systems as well as informed and predictive prefetching techniques.

Chapter 3 explains the assumptions made in this dissertation, shows the proof of concept, and validates system parameters used throughout this dissertation.

Chapter 4 , outlines the system architecture, algorithm Design, and performance evaluation for our iPipe solution that assumes enough storage system's bandwidth.

Chapter 5, outlines our IPO solution that performs pipelining in a limited bandwidth storage system. The chapter discusses the system architecture, algorithm Design, and the performance evaluation.

Chapter 6 outlines IPODS solution which implements IPO in distributed multi-levels storage systems.

Chapter 7 discuss our solutions' prototyping results.

Finally, Chapter 8 provides a summary of this dissertation study with a list of directions for future research.

Chapter 2

Literature Review & Current Work

Previous researchers have suggested that application-disclosed hints can be used by prefetching mechanisms to dramatically improve I/O performance. To our best knowledge, however, ours is the first study to focus on informed prefetching in multi-level storage systems, the first to consider how to apply the cost-benefit model in a multi-level prefetching system, the first to construct a pipeline to coordinate multiple prefetching mechanisms in a multi-level storage system, and the first to offer a systematic performance evaluation of informed prefetching in multi-level hybrid storage systems.

In this Chapter, we will discuss closely related work in both storage systems in general and prefetching in particular.

2.1 Storage systems current work

We will begin this subsection by describing multiple-level storage systems (see Section 2.1.1). Then, we will compare solid state disks with traditional hard disk drives (see Section 2.1.2). Next, we introduce new techniques in parallel storage systems (see Section 2.1.3). Finally, we will discuss related work in distributed storage systems (see Section 2.2).

2.1.1 Multi-level storage systems

A multi-level storage system consists of a hierarchy of heterogeneous storage devices that differ in their hardware, speed, size, and other specifications [87]. Multilevel storage systems provides cost-effective solutions for large-scale data centers without significantly affecting I/O response times. The I/O performance of a multi-level storage system depends on data placement of the system. Ideally, a high-level storage device should store two types of data:

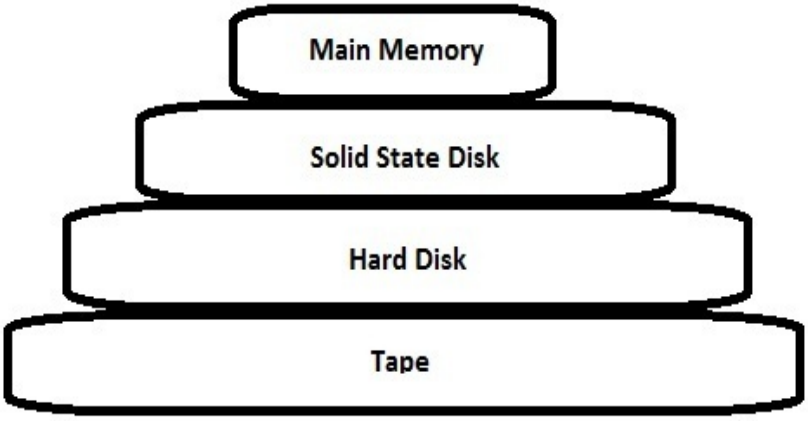

Figure 2.1: Multi-level storage system that consists of different storage devices with various speed performance.

(1) popular data that are frequently accessed and (2) data that are likely to be access in the not-too-distant future.

Typical storage devices in a modern multi-level storage system include main memory, solid state disks, hard disks, and magnetic tape subsystems (see, for example, [23]). Figure 2.1 shows a 4-level storage system, which is a straightforward extension of a traditional 2-level storage system with main memory and hard drives.

Multi-level storage systems and caches share similar advantages. A multi-level cache system contains a hierarchy of more cache levels [33]. Previous studies show that there is a maximum benefit of having a single level cache [21] [22]. Making tradeoff between cache latency and hit rate is a challenge, because large caches inevitably have high hit rates with long latency. To achieve a good tradeoff between hit rate and latency, researchers proposed multiple-level caches, where small, fast caches (i.e., upper level caches) are backed up by large, slow caches (lower level caches).

Similar to multi-level caches, multi-level storage systems [18] [19] [20] first check their upper-level storage devices. If data items are not in the upper level storage, the next storage level is checked. This process is repeated until the required information is retrieved.

In a multi-level storage system, data can be moved from one particular level to another by data migration processes. The decision of migrating data blocks from one level to another

is based on the data's metadata (e.g., popularity and pass access patterns). Data migration based on popularity or predictions is a good example of the important metadata used by data migration modules [38], [34], [36]. When a multi-level storage system migrates data to upper levels based on popularity or predicted access patterns, the migrated data will be accessed by clients in a fast way thanks to reduced access delays [37].

2.1.2 Solid State Disks and Hard Drives

Compared to traditional hard drives (HDD), solid state disks (SSD) show better data read performance. Tuma provides comparisons between a wide range of SSDs and HDDs; detailed comparisons can be found in [85]. Tuma's results show that SSDs have better random read-performance than HDDs. For example, reading 4- KB blocks from a seagate HDD achieves a throughput of 0.70 MB/s; whereas reading the same data blocks from M-systems Model FDD 3.5" Flash SSD can obtain a throughput of 4.3 MB/s. Rizvi and Chung compares SLC with MLC-flash SSDs with HDDs; the findings show that SSDs offer better I/O throughput than HDDs [86]. Therefore, solid state disks are becoming increasingly popular in the support of large-scale multimedia systems requiring high I/O bandwidth. Researchers can obtain the specifications of various HDD and SDD products from vendors (e.g., [94] [93] [92] [90]). The product data sheets clearly indicate that SDDs outperform HDDs in term of reads. For example, WD Caviar Green has approximately a maximum read bandwidth of 190 MB/s, assuming that the WD's SSDs are connected with SATA 1.5 Gb/s (1.5 Gb/s is approximately equal to 190 MB/s). On the other hand, Nova Series V64 solid-state drive can achieve a high read bandwidth of 215 MB/s, which is higher than that of WD's SSDs. Different SSD products vary in their I/O performance. For example, Intel's X25-E Extreme SATA SSDs - with a read bandwidth of 250 MB/s - is better than Nova Series V64 SSDs. Many SSD and HDD products distinguish themselves from each other in terms of I/O performance, size, cost, and other aspects. In general, SDDs have good I/O performance and a high price whereas HDDs offer high storage capacity at low cost. As such,

J. No built a hybrid file system using both SSDs and HDDs, thereby taking the advantages of two different types of storage devices [56].

2.1.3 Parallel Storage Systems

Parallel storage systems are commonly deployed in supercomputers [1] [23] [63]. A parallel storage system consist of redundant storage devices offering high I/O performance and bandwidth [51] [60]. Disk arrays are important components in parallel storage systems.

Scalable parallel storage systems (see, for example, [23] [63] [49]) provide I/O parallelisms through the stripping technique, in which each large data block is stripped among an entire disk array [64]. When a particular data block is requested, the request will be directed to the entire disk array, where multiple disk controlled by a disk controller can coordinate and respond to the request in parallel. Generally speaking, it is very expensive to build large-scale parallel storage systems; therefore, researchers have proposed various ways to reduce the cost of scalable parallel storage systems (see, for example, [57]). Cluster storage systems - a new high-performance computing storage resource, is one of a few effective ways to develop cost-effective parallel storage systems, because cluster storage systems use entirely off-the-shelf components from data storage vendors.

Parallel storage systems can be very reliable, thanks to the fault tolerance features offered by data redundancy [59] [58]. In the event of a disk or a storage device fails, users are not aware of such a failure. Thus, all requests issued to a fault-tolerant storage system can be processed, even in the presence of disk failures. Data stored on failed disks can be recovered or reconstructed from data stored in other disks in the system.

A recent study shows that informed prefetching mechanisms can rely on disk parallelisms to quickly prefetch data in parallel [1]. Applications with parallel I/O access patterns benefit greatly from parallel storage systems that make parallel informed prefetching possible [50] [53] [54]. In this dissertation study, we focus on the issue of informed prefetching in the context of storage system in general and multiple-level storage systems in particular.

2.1.4 Distributed Storage Systems

A distributed storage system consists of multiple storage servers or sites that communicate through a network. Data files stored in distributed storage systems may be stripped or replicated among the storage servers [72] [83] [68] [82]. Clients access data on a distributed system through computer networks connecting all the storage servers in the system. Modern distributed systems achieve high I/O performance, bandwidth, and scalability [67] [69] [66]. A previous study addressed mismatches among applications, storage devices, and interconnection medium requirements in distributed storage systems [71]. Tierney **et al.** designed an architecture of a distributed storage system that uses a high speed ATM network to improve system performance [84]. I/O workload characteristics in the context of distributed systems were investigated by researchers [70], because understanding the I/O workloads of applications in distributed systems is the first step toward improving the applications' I/O performance.

Numerous distributed storage systems exhibit high reliability through data replication mechanisms, where each data item in a storage server has one or two backup copies in other servers. Because there are three copies of a set of data, If a particular storage server fails, the data on it can still be accessed from the backup servers [61] [62] [80].

Distributed file systems are file systems that allow file accesses from multiple storage servers or nodes through a computer network. Good examples of these systems include Google file system (GFS), Andrew file system (AFS), Hadoop distributed file system (HDFS), Ceph, Frangipani, Deceit,ITC, Coda, wide area AEF, and Sprite, to name just a few [73] [74] [76] [77] [78] [79] [39] [40] [42] [45]. Access transparency - an important feature of modern distributed file systems - allows users and applications to store and retrieve files over the network as if the files were on local disks [75].

Recently, Hadoop distributed file system (HDFS) emerged as an efficient distributed file system supporting the Hadoop software framework for data-intensive distributed applications [43] [44] [45] [46] [47] [48]. Hadoop is an open-source implementation of Google's MapReduce

programming model designed to deal with thousands of computing nodes and Petabytes of data. HDFS in the Hadoop framework is a portable and scalable file system widely deployed in numerous large-scale cluster computing systems. Hadoop, with the support from HDFS, relies on location awareness to run Hadoop applications on computing nodes in which processed data is located. To improve reliability of Hadoop, HDFS maintains multiple copies of each file and keeps the different copies on different computing nodes. Like MapReduce applications, Hadoop programs comprise two types of operations: 1) a Map operation that outputs a key-value pair used for generating a set of intermediate key-value pairs and 2) a Reduce operation that merges all the intermediate values associated with the same intermediate key.

Scalability is a critical issue in the development of large-scale distributed file systems. Scalability issues must be addressed along with consistency, availability, and synchronization. For example, Srinivas and Janakiram built a generic scalability model for distributed file systems. Their model characterizes scalability as a factor related to workload and faultload [41]. This model was used to compare several well-known distributed file systems in terms of availability, synchronization, and consistency.

2.2 Prefetching

In this section, we focus on various issues related to prefetching. We will start this section by classifying prefetching approaches into two group. Then, we will outline the state-of-the-art research in informed prefetching. Next, research progress on predictive prefetching will be reported. Finally, we will summarize a few studies on prefetching mechanisms in multiple-level storage systems and distributed systems.

2.2.1 Two Types of Prefetching

Due to the large performance gap between CPU and disks, I/O becomes a significant performance bottleneck in large-scale computing systems [81]. To hide I/O latencies, prefetching

can preload data from disks into the main memory prior to data accesses. Prefetching techniques can be classified into two categories: informed and predictive approaches. Informed prefetching relies on applications to provide future access hints [1]; predictive prefetching involves algorithms that predict future access patterns of applications based on their historical access patterns [2]. In the next two subsections, we will review previous studies on informed prefetching and follow with an overview of existing predictive prefetching strategies.

2.2.2 Informed Prefetching

A study conducted by Patterson *et al.* [1] [12] [11] [16] [26] inspired us to concentrate on informed prefetching issues. Patterson *et. al* suggested that informed prefetching algorithms - invoking storage parallelisms - can take advantage of application-disclosed I/O access hints to eliminate I/O stalls through aggressive prefetching [26][1][12]. Performance benefits of informed prefetching were confirmed by other studies undertaken by Huizinga *et al.* [10] and Chen *et al.* [30].

When it comes to disk arrays, parallel informed prefetching aims to leverage parallel I/O to improve prefetching performance. Parallel informed prefetching is made possible, because data blocks are striped across an array of disks [25][51]. Parallel informed prefetching eliminates I/O stalls by placing hinted data in the cache before the data are requested by applications. To improve cache usage for both prefetching and caching, Patterson *et al.* proposed a cost-benefit model, which assists their prefetching mechanism to balance cache/buffer space shared between the LRU (least-recently-used) cache and the prefetching buffer [1]. The model decides the benefit of using more buffers for prefetching and the cost of ejecting a LRU block or a prefetched block.

A informed prefetching mechanism makes prefetching decisions based on access hints and access patterns being given a priority. Researchers have investigated various ways of collecting information to offer accurate access hints for informed prefetching mechanisms.

Without appropriate hints, the prefetching mechanisms are unable to make accurate decisions. Chang and Gibson proposed an application speculative execution scheme that pre-executes applications to record the applications future accesses [13]. Future access patterns recorded by Chang and Gibson's scheme can be used to guide informed prefetching algorithms to preload data that are likely to be accessed. Byna *et al.* studied a solution that combines post-execution and runtime analysis to reduce future I/O reads prediction overhead [35]. Byna's study is one step toward improving the performance of existing informed prefetching mechanisms. Our informed prefetching algorithms presented in this dissertation are orthogonal to the above research in the sense that integrating our solutions with these techniques of collecting hints can significantly improve the overall performance of informed prefetching mechanisms for multiple-level storage systems.

Prefetch horizon is an important concept in informed prefetching. A prefetch horizon is an upper bound on prefetch depth; evidence shows that there is no performance benefit from prefetching further ahead than the prefetch horizon. Much attention has been paid to the impact of prefetch horizon on prefetching performance (see, for example, [1][12]. In this dissertation study, we will also evaluate the performance impact of prefetch horizon on our informed prefetching mechanisms in parallel and distributed storage systems.

2.2.3 Predictive Prefetching

Predictive Prefetching (a.k.a., automatic prefetching) relies on past I/O accesses to predict future access and to prefetch predicted future required data [2] [31]. Griffioen and Appleton developed a model for encoding past access patterns and future access probabilities [2]. In Griffioen and Appleton's model, directed weighted graphs are employed to estimate access probabilities [2]. Please refer to [2] for details on the predictive prefetching model where directed weighted graphs are used.

The Markov predictor is another model used by existing predictive prefetching algorithms [5] [88] [4] [6] [55]. It uses prediction by partial match to find recurring sequences of

I/O events. The Markov predictor has been widely applied in web prefetching, because web accesses of the Internet have a hypertextual nature [7] [8][88].

Recently, we developed an automatic prefetching and caching system called APACS [31]. Three unique techniques integrated into APACS include (1) dynamic cache partitioning, (2) prefetch pipelining, and (3) prefetch buffer management. APACS dynamically partitions the buffer cache memory, used for prefetched and cached blocks, by automatically changing buffer/cache sizes in accordance to global I/O performance.

The aforementioned studies focused on predictive prefetching, whereas our solutions performs informed prefetching. Since informed-prefetching and predictive-prefetching algorithms supplement one another, our solutions can seamlessly work together with a variety of predictive prefetching algorithms.

2.2.4 Prefetching in multi-level storage systems

Prefetching in multi-level storage systems is technically challenging due to the following dilemma - aggressive prefetching is required to efficiently reduce I/O latency [3], whereas overaggressive prefetching may waste I/O bandwidth by transferring useless data from lower-level to upper-level storage devices.

Previous studies showed that a single-level cache offers I/O performance improvements [21][22]. Empirical evidence indicates that a multi-level storage system can extend the benefits of a single-level cache by augmenting storage systems with multi-level caches [19][18][20]. Recently, Nijim proposed a multi-layer prefetching algorithm that can speculatively prefetch data from tapes to hard drives and preload data from hard drives to solid state disks [23] [24]. The experimental results show that one can leverage prefetching techniques to enhance I/O performance of multiple-level storage systems.

A recent study conducted by Zhang *et al.* suggests that prefetching algorithms designed for single-level storage systems are inadequate for multi-level systems [9]. Rather than proposing a new prefetching algorithm for multi-level storage systems, Zhang *et al.*

developed a hierarchy-aware optimization scheme called PFC [9]. Their PFC coordinator is applicable to any existing prefetching algorithm, because PFC's aim is to coordinate prefetching aggressiveness across multiple levels of caches.

Kraiss and Weikum developed an integrated approach to migrating data in multiple-level storage systems [34]. In particular, Kraiss and Weikum focused on prefetching data between the tertiary, secondary, and primary storage, thereby masking high latency of the tertiary storage [34]. In their study, replacement policies and the interaction of prefetching/caching policies were also considered. It is worth noting that Kraiss and Weikum's prefetching mechanism relies on Markov-chain predictions to make prefetching decisions [34].

Our solutions differ from the existing prefetching and caching techniques for multi-level storage systems in the sense that they manage multi-level caches based on application-disclosed access patterns.

2.2.5 Prefetching in Distributed and Parallel Storage Systems

Apart from multiple-level storage systems, parallel and distributed storage systems can benefit from prefetching schemes. For example, Patterson *et al.* proposed and implemented an informed prefetching model in a distributed storage system where data are allocated in remote nodes [1]. Unlike multi-level storage systems, distributed storage systems have to face the challenge of reducing latencies of accessing remote storage nodes through networks. To address this challenge, Rochberg and Gibson extended a network file system by integrating an informed prefetching mechanism [17]. Rochberg and Gibson's approach hides disk latency while exposing storage parallelism. Their experimental results show that informed prefetching over network reduces application's execution time by anywhere from 17% to 69% [17].

Performance of parallel storage systems can be improved through collective input/output (I/O). A collective application programming interface (API) is necessary to obtain good I/O performance. Collective APIs, however, require applications to disclose their entire access

patterns to fully reorder I/O requests. To solve this problem, Gibson and Faloutsos investigated an approach to optimizing collective I/O access patterns through informed prefetching [65]. Gibson and Faloutsos's idea was to make use of informed prefetching to exploit any amount of available memory to overlap I/O with computation [65]. After comparing their approach to disk directed I/O, Gibson and Faloutsos showed that under certain conditions, a per-processor prefetch depth equal to the number of drives is able to guarantee sequential disk accesses for collectively accessed files in a parallel storage system. Gibson and Faloutsos also conducted experiments to indicate that a prefetch horizon of one to two times the number of disks per processor is good enough to match disk-directed I/O's performance for sequentially allocated files.

Distributed and parallel storage systems are widely adopted to support high-performance Web servers; therefore, in the context of the Web, it is critical to improve I/O performance of distributed and parallel systems. Nanopoulos *et al.* investigated interesting predictive prefetching issues in the Web, where a predictive Web prefetching mechanism aims at deducing forthcoming page accesses of a client based on its past accesses [88]. The prefetching algorithms developed by Nanopoulos and his team are referred to as Markov predictors. Nanopoulos *et al.* studied factors affecting the performance of their Web prefetching algorithms that are based on data mining schemes [88]. The experimental results show that the new data-mining-based prefetching algorithm can provide performance improvements over previously proposed Web prefetching algorithms.

Our informed prefetching strategies are quite different from the aforementioned approaches, because ours leverage a prefetching pipeline to efficiently migrate hinted data to upper-level storage devices, thereby significantly reducing data access times.

2.3 Summery

In this Chapter we discussed related work in both storage systems and prefetching techniques. First, we described recent studies on multiple-level storage systems, solid state

disks, and parallel and distributed storage systems. Then, we summarized research progress on both informed and predictive prefetching schemes.

In particular, we conducted a literature review to investigate novel prefetching algorithms and mechanisms designed for multiple-level storage systems and distributed/parallel storage systems. We also described the major differences between our proposed informed prefetching approaches and existing prefetching solutions reported in the literature.

Chapter 3

Assumptions and Parameters Validations

In this chapter, let us first discuss assumptions used in this dissertation research. Sections 3.1-3.2 explain and justify research assumptions related to caching, initial data allocations, bandwidth limitations, and I/O writes. Then, we describe in Section 3.3 system parameters validated by empirical studies. These validated system parameters will be used in the subsequent chapters.

3.1 System Assumptions

In this subsection, we illustrate assumptions on data storage systems as well as software environments.

3.1.1 Demand Misses

In this study, we intend to efficiently manage I/O buffers to balance caching against prefetching. A caching algorithm manages all demand misses. For example, when a demand miss of an application occurs, the application stalls until the miss is satisfied for a complete disk read latency period. The application is not able to consume the prefetched data and to invoke a new prefetching request. When a demand miss takes place, caching buffer size should be increased at the cost of reduced prefetching buffer size. In doing so, storage systems can boost I/O bandwidth to quickly respond demand miss requests.

In addition, we assume that only one demand miss request can take place at a given time period. Our pipelining mechanism is synchronized with an informed prefetching scheme. Thus, when a new informed prefetching request takes place, a new pipelining request is

initiated. This synchronization means that the pipelining mechanism also stalls with demand misses.

3.1.2 Informed Caching

Caching prefetches is an effective technique for improving the performance of I/O intensive applications [14] [15]. In addition to the prefetching process, informed caching removes the need to fetch data again, which in turn reduces the application's stall time. The most recently used policy (MRU) can be used in an informed caching mechanism, thereby improving I/O performance of the system when there are many repeated access patterns.

In our dissertation study, we do not investigate informed caching due to the following two reasons:

- In the worst case, informed caching does not show any performance optimization. The preliminary results show that several benchmarks are unable to benefit from informed caching.
- Informed caching is a technique for optimizing the informed prefetching process. The performance optimization gained by employing informed caching does not necessarily represent the advantages of our solutions.

Rather than informed caching, the pipelining mechanism is at the heart of our system design. The pipelining aims to optimize informed caching by reducing the overhead of prefetching. With the pipelining mechanism in place, storage systems need less caching space.

Unfortunately, it is difficult, if not impossible, to predict whether a particular hinted data block will be cached. In one of our future studies, we will integrate the informed caching algorithm with our pipelining algorithms in order to determine if a hinted access needs to be pipelined. Please refer to Chapter 8 for details on this future research direction.

3.1.3 Bandwidth Limitations

We assume that the maximum I/O bandwidth offered by a parallel storage system equals the maximum number of concurrent read requests that may take place in the storage system without causing any congestion. Thus, the maximum I/O bandwidth depends on the scalability of the storage system. A similar assumption can be find in the literature (see, for example, [1]). In the first part of our research (see Chapter 4), we assume that parallel storage systems have enough I/O bandwidth with no congestion.

This assumption, of course, is relaxed in our solution described in Chapter 5, which proposes a pipelined prefetching solution for parallel storage systems with limited bandwidth.

3.1.4 I/O Bandwidth Sharing

I/O bandwidth can be shared among solid state disks (i.e., SSDs) and hard drive disks (i.e., HDDs). In other words, a multiple-level storage system can access its SSDs and HDDs in parallel. For example, let us consider a multi-level disk array consists of 5 disks. When there are two concurrent read requests taking place, at the same time three concurrent data blocks can be written from HDDs to the SSDs. We conducted empirical experiments to validate the correctness of this assumption, showing that fetching data from SSDs to main memory and moving data from HDDs to SDDs can be carried out in parallel. This observation indicates that it is feasible for us to implement a pipelined prefetching mechanism.

We only focus on reads in this dissertation study; our goal is to improve I/O performance of read-intensive applications. Thus, write-intensive applications can not benefits a whole lot from our solution, because writes cause informed prefetching latency and the prefetching pipeline can be delayed by demand misses. Other studies show that writes are less critical than reads because applications handles writes in an asynchronous way - applications do not wait until new data blocks are written. Writes are governed by the write-behind policy; log-structured file systems can significantly improve I/O performance for write-intensive

applications. Please refer to [27] [28] [29] for details on the write-behind policy and the log-structured file systems.

In the event that a storage system encounters bandwidth shortage (because of pipelining, writes behind, power availability, or other reasons), informed prefetching requests may not be honored in a timely manner. This problem can be solved by shrinking the pipelining depth, meaning that the pipelined prefetching mechanism does not aggressively load data blocks from lower-levels of the storage devices.

3.1.5 Life Time of Solid State Disk (SSD)

Because the life time of SSD's storage cells is limited. The pipelining process may have negative impact on the life time of the SSDs. The purpose of our research is to build a generic solution for informed and pipelined prefetching that optimizes prefetching performance in multi-level storage systems. Although our solutions inevitably affect SSD's life time, other types of storage devices (e.g. HDD and Tapes) used in a multi-levels storage system will not be affected by our solution at all.

Many techniques are proposed to improve reliability of SSDs. Quantifying the impacts of our pipelined prefetching on reliability of SSDs is out the scope of our research. In the future, we plan to implement a scheme that distributes the pipelining writes among the SSD's cells. Hence, this future solution will provide load balancing and improve the SSD's life time.

3.2 Assumptions on Prefetching and Pipelining

In this section, we illustrate the assumptions on our pipelined prefetching algorithms.

3.2.1 Initial Data Allocation

Let us consider a simple multi-level storage system, where there are two-level disk arrays composed of an SSD (top) and an HDD (bottom). All of the data can initially be distributed

between the two levels. In our simulation studies, we assume that all the informed prefetched blocks are initially located in the HDDs (bottom level). This assumption is reasonable because of the following five reasons.

- Normally, HDDs have larger storage capacity than SSDs and; therefore, the probably of finding data in the HDDs is greater than that of finding it in the SSDs.
- Research of prefetching in multi-level storage systems (see previous studies conducted by Nijim and Zhang *et al.*) assumes that the bottom level contains less important data [23] [9]. Consequently, prefetching techniques move data likely to be accessed in the future to the upper-level storage devices. Thus, the probability of finding hinted data blocks in the HDDs is increased.
- The worse case scenario is that all data are initially allocated in the lowest level (e.g., HDDs in our study). If some data were initially allocated in the upmost level (e.g., SSDs)of the system, the number of pipelined prefetching requests would be reduced.
- Having data partially or totally allocated in SSDs does not properly show the performance improvement offered by our solutions. For this reason, we consider a conservative case - the worst-case scenario - in which all the data are initially allocated to the lowest level.
- If a hinted data block is already available in an SSD, the corresponding informed prefetching request will be discarded.

3.2.2 A Pipeline for Data Transfers

We build a pipeline to transfer data among multiple storage levels in parallel. For example, we design two prefetching modules: the first one fetches data from HDDs and stores the data to SSDs; the second one prefetches data from SSDs to I/O buffers in main memory. These two prefetching modules can perform in parallel to form a prefetching pipeline. In

our design, the lowest-level storage devices always achieve original copies of prefetched data. Our motivations behind this design are:

- Keeping original copies in lower-level storage devices can save storage space in higher-level storage devices, which are more expensive than the lower-level counterparts.
- In case we have to migrate entire data blocks from HDDs to SSDs (see Chapter 8 for details on data migrations), we need to keep moving the data back and forth among multiple-level storage, which consumes I/O bandwidth.
- Keeping the original copies at the lowest level can save I/O bandwidth, which in turn improve the overall I/O performance of multi-level storage systems.

3.2.3 Writes and Data Consistency

Writes may create a updated version of a data block that has been prefetched or is currently being prefetched. This leads to inconsistency between the new version stored in an upper-level storage and the original copy in the lower-level storage. To counter this problem, we adopt the write-back policy. Thus, our pipelined prefetching mechanism keeps track of writes. When data blocks are evicted from upper-levels, updated data will be forced to written down to the lowest level. In doing so, we can guarantee that data copies in upper levels are consistent with copies stored in lower levels.

If the writing and updating process cause any I/O latency for informed prefetching, the prefetching pipeline will have to wait until the updating process is done. This is because the prefetching process must be synchronized with the writing and updating process.

Recall that our focus is read-intensive applications where writes are considered rare. Applications with few writes do not experience any significant I/O delays introduced by our mechanism.

3.2.4 Pipelining Depth

Our pipelined prefetching algorithms reserve space (called pipelining depth) in the upper-level storage devices for the concurrent prefetching processes at multiple levels. This space depends on (1) the number of concurrent prefetching requests that our schemes can handle and (2) the available I/O bandwidth. As we will discuss later, the pipelining depth is small and the storage consumed by hinted blocks is insignificant compared with the entire storage system's capacity.

In Chapters 4 and 5, we will provide a mathematical model that calculates the pipelining depth, which is used to indicate reserved buffer space for pipelined prefetching.

3.2.5 Block Size

Using SSDs and HDDs installed in the servers in our laboratory, we observed that an SSD does not provide better performance than an HDD for small data blocks; SSDs are faster than HDDs when the block size is at least 10MB. In our distributed system implementation, our empirical results show that in order to benefit from high speed SSDs, a data block size must be at least 200MB.

Many file systems pack data into large blocks to improve I/O performance. For example, the data block size in HDFS (Hadoop Distributed File System) is 64 MB [43] [44]. In reference [45], HDFS block size is increased to improve system performance. Hadoop achieves (HAR) tool used to pack several small files into a large one [89], can be used to create large data blocks out of small ones.

3.2.6 LASR Traces

Our develop trace-driven simulators to evaluate performance of our prefetching mechanisms under I/O-intensive workload. Traces used in our experiments represent applications that have small CPU processing time between each two subsequent I/O reading requests. All data blocks in the tested traces have identical block size. More specifically, we use the

machine01 trace (called LASR1) [97] and the machine06 trace (called LASR2) [98] in our simulation studies.

We assume that all the requested data blocks have the same blocks size. For example, block size is set to 10MB in single-node systems and 200MB in the distributed system tested in our studies. We also assume that time between each two sequential I/O reading requests (i.e. arrival rate or processing time) is the smallest possible value (later: (T_{cpu} + T_{hit} + T_{driver})). This setting allow us to evaluate I/O-intensive cases. Simulation results of the two traces share the same trend but differ in elapsed time due to the variation between the number of I/O reading requests that appears in both of them.

In our future work (see Chapter 8), we will collect traces from real-world benchmarks. Then, we will use new traces to further evaluate our solutions. Our major obstacle is that we need to implement these benchmarks to represent a wide range of real-world I/O-intensive applications.

3.3 Validations

3.3.1 Overview

In this section, we provide the proof of concept that lays a solid foundation for proposed solutions described in the subsequent chapters. Next, we validate system parameters of our simulators using data collected from real-world storage systems. Finally, we discuss the I/O bandwidth limitation issue of parallel storage systems.

We need to find the block size for which SSD's read performance is better than that of HDD. We also validate SSD and HDD read latencies (i.e., $T_{ss-cache}$, $T_{hdd-cache}$) for a single data block. We also validate the T_{hdd-ss} latency which is the time spent in reading a data block from an SSD and write the block back to the SSD. Next, we validate SSD and HDD read latencies of accessing a remote storage node through a network (i.e., $T_{ss-network-cache}$, $T_{hdd-network-cache}$ respectively). Finally, we validate the time needed for the application to consume a single prefetched data block (i.e., (T_{cpu} + T_{hit} + T_{driver})).

The following list summarizes the validated system parameters:

- Data block size.
- T_{cpu}: CPU time to process a data block.
- T_{hit}: Time to read a single data block from an I/O buffer.
- [31] T_{driver} Time to allocate a single buffer in the buffer cache.
- T_{hdd-ss}: Time to fetch a single data block from HDD and to store the block in SSD.
- $T_{hdd-cache}$: Time to fetch a single data block from HDD and to store the block in the buffer cache.
- $T_{ss-cache}$: Time to fetch a single data block from SSD and to store the block the buffer cache.
- $T_{hdd-network-cache}$: Time to fetch a single data block over a network from a remote node's HDD.
- $T_{ss-network-cache}$: Time to fetch a single data block over a network from a remote node's SSD.

3.3.2 System Setup

The following are storage and network devices tested in our laboratory:

- Memory: Samsung 3GB RAM Main Memory.
- HDD : Western Digital 500GB SATA 16 MB Cache WD5000AAKS.
- SSD: Intel 2Gb/s SATA SSD 80G sV 1A.
- Network Switch: [96] Network Dell Power Connect 2824.

3.3.3 Proof of the Concept

As discussed in our research motivations, multi-level storage systems consist of several levels that vary in disk read performance (e.g., upper levels are faster than lower levels). Our proposed solutions aim to optimize informed prefetching when it is implemented in a parallel multi-levels storage system by prefetching multiple blocks in a pipelining manner to reduce disk read latency. Pipelined prefetching aims to reduce applications' stalls and elapsed time.

Hiding disk read latencies is the main approach used in our prefetching designs. Multiple prefetching modules are deployed at different storage levels. For example, one module handles the prefetching process between HDDs and SSDs; another module takes care of the preteching process between SSDs and main memory. These two prefetching modules can fetch hinted blocks in parallel to form a two-stage prefetching pipeline.

The goal of the pipelining approach is to hide long read latencies by moving data from HDDs to SSDs at the background. With our pipeline in place, many hinted blocks are residing in SSDs rather than in HDDs. Fetching hinted data from SSDs, of course, is faster than fetching the data from HDDs.

When it comes to a distributed multi-level storage system, the long read latency problem is even worse - read latencies include network and server latencies. Our pipelining mechanism can hide the long read latencies by fetching hinted block from remote storage nodes to local SSDs. time that we cannot change (their optimization is beyond our research's scope). Hiding disk read latencies improves the I/O throughput of both local multi-level storage systems and distributed multi-level storage systems.

Our preliminary results show that SSDs have better performance than HDDs when block size exceeds a particular threshold. Figure 3.1 and 3.2 validate this statement where read latency of SSD is less than that of HDD. We performed this experiment using read requests. We conduct experiments on real-world storage devices in our laboratory, showing that the performance difference between HDDs and SSDs begins at 10 MB per data block in a local storage system and 200 MB per block in a distributed system.

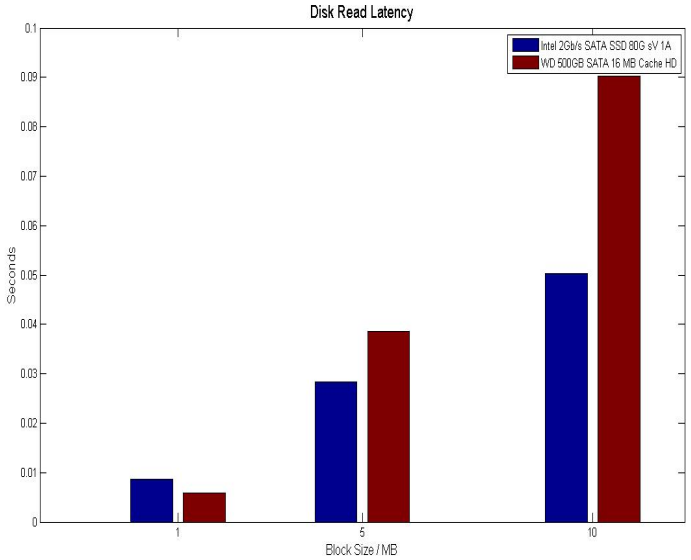

Figure 3.1: Read Latency of HDDs and SDDs. When block size is 10 MB, SSD has better read performance than HDD.

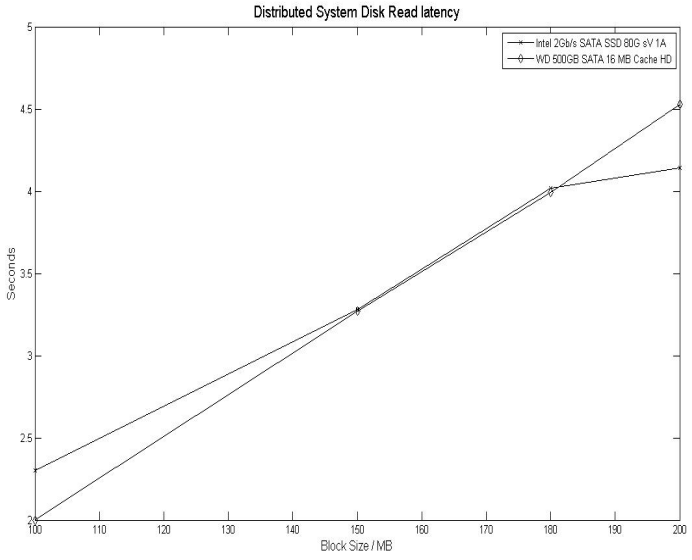

Figure 3.2: Read Latency of accessing a remote HDD and SDD through the LAN network connection. In this distributed system setting, SSD has better read performance than HDD when block size is 200 MB.

3.3.4 Block Size

Our preliminary results based on our storage devices indicate that an SSD is guaranteed to exhibit better performance than HDD when the data block size is 10 MB for a local system and 200 MB for a distributed system. Figures 3.1 and 3.2 validate this argument. We perform a comparison between the highest read latency of SSD with the lowest of HDD, observing that for small data blocks (e.g., 1 MB), SSD is not superior to HDD. When block size reaches 5 MB, SSD begins to improve performance over HDD. We choose to use 10 MB as the next point where SSD shows noticeably better performance than HDD. For the case of distributed systems, we observe that an SDD starts to exhibit better performance than an HDD when the block size is 200 MB. Because of these preliminary results, in the subsequent chapters we consider the block size equal to 10 MB for local systems (see our iPipe and IPO schemes) and 200 MB for distributed system (see the IPODS scheme).

3.3.5 Model $\mathbf{T}_{cpu} + \mathbf{T}_{hit} + \mathbf{T}_{driver}$

We denote (T_{cpu} + T_{hit} + T_{driver}) as the time that an application takes to consume a prefetched data block from I/O buffers. This is considered the single step time to which the entire prefetching system is synchronized. It is also the time interval between two sequential I/O reading requests. We discover from our experiments that this single step time has three components: T_{cpu}, T_{hit}, and T_{driver}, which have a positive relation with data block size.

While calculating (T_{cpu} + T_{hit} + T_{driver}), values of the three components are varying. Therefore, it is difficult to assign a constant value to each of these three factors. A recent study (see the TIP model) considers (T_{cpu} + T_{hit} + T_{driver}) to be a constant value. We take a similar approach by modeling this single-step time as a constant. We consider the worst case scenario. Please note that when (T_{cpu} + T_{hit} + T_{driver}) is small, the application consumes prefetching buffers faster.

Estimate $\mathbf{T}_{cpu} + \mathbf{T}_{hit} + \mathbf{T}_{driver}$ when block size is 1KB.. In our first pipelined informed prefetching scheme or iPipe, we assume that the data block size equals to 1KB. In

other prefetching research (e.g., TIP), the data block size is 8KB. Thus, we decide to use a small data block to make fair comparison with the existing solution. Now, let us estimate $T_{cpu} + T_{hit} + T_{driver}$ for when data size is 1KB. In this small-data-block case, T_{cpu} equals 4 nanosecond, T_{hit} equals 0.15 microsecond, and T_{driver} equals 0.06 microsecond.

The (T_{cpu}) value was derived from a trivial application that measures the time needed to perform simple computations (i.e., a loop on each byte without doing any computing) on 1KB data. We consider that the CPU always does computations on the 1KB data. The (T_{hit}) value is derived from the 1KB main memory read latency specification (see [90] for details). T_{driver} is derived from a trivial application that measures the time needed to allocate 2000 unique 8 KB data blocks (see similar application in the TIP research). In the worst-case case, $T_{cpu} + T_{hit} + T_{driver}$) becomes at least 0.2 microsecond. We use this worst-case-value in our subsequent simulation studies.

Estimate $T_{cpu} + T_{hit} + T_{driver}$ when block size is 200MB. In order to validate the value of $T_{cpu} + T_{hit} + T_{driver}$ when block size is 200MB in our simulation studies, we need to individually validate each factor in this expression. As we mentioned earlier, 200MB is a large block size. T_{cpu} in particular varies each time depending on applications' data processing time. For the value of $T_{cpu} + T_{hit} + T_{driver}$, we must first determine the best and worst cases. Then, in our simulations, we may randomly choose a value between the best-case and the worst-case values.

• **Validate T_{hit}:** We used RamSpeed cache and memory benchmarking tool [99] to validate the T_{hit} value when block size is set to 200MB. Table 3.1 shows the I/O bandwidth measured when we run the RamSpeed benchmark in our testbed.

It is clear from the benchmark testing that the I/O bandwidth achieved for 200MB block size is at least 5304.49 MB/s. We take the lowest possible approximation for the T_{hit} value to make applications more I/O than CPU intensive. We also observe that I/O bandwidth for block size larger than 4096KB share similar trend. Using this testing information, we estimate that the T_{hit} value of a 200MB data block is approximately equal to 0.037 second.

Table 3.1: I/O bandwidth measured using the Ramspeed benchmark.

INTEGER & READING	1 Kb block: 10586.42 MB/s
INTEGER & READING	2 Kb block: 10624.48 MB/s
INTEGER & READING	4 Kb block: 10626.43 MB/s
INTEGER & READING	8 Kb block: 10627.82 MB/s
INTEGER & READING	16 Kb block: 10618.71 MB/s
INTEGER & READING	32 Kb block: 10386.05 MB/s
INTEGER & READING	64 Kb block: 9010.96 MB/s
INTEGER & READING	128 Kb block: 9016.90 MB/s
INTEGER & READING	256 Kb block: 9020.86 MB/s
INTEGER & READING	512 Kb block: 9021.48 MB/s
INTEGER & READING	1024 Kb block: 9012.64 MB/s
INTEGER & READING	2048 Kb block: 8110.29 MB/s
INTEGER & READING	4096 Kb block: 5870.99 MB/s
INTEGER & READING	8192 Kb block: 5193.41 MB/s
INTEGER & READING	16384 Kb block: 5221.85 MB/s
INTEGER & READING	32768 Kb block: 5222.84 MB/s
INTEGER & READING	65536 Kb block: 5359.69 MB/s
INTEGER & READING	131072 Kb block: 5272.53 MB/s
INTEGER & READING	262144 Kb block: 5354.08 MB/s
INTEGER & READING	524288 Kb block: 5342.72 MB/s
INTEGER & READING	1048576 Kb block: 5304.49 MB/s

• **Validate $\mathbf{T}_{cpu}$:** In order to calculate T_{cpu}, we implement a trivial application that measures the time spent in performing simple computations (i.e., a loop on each byte without any computation) on a 200MB data block residing in the main memory. We measure that the application takes at least 0.067000 seconds to complete the loop. This T_{cpu} value, of course, can increase when massive numerical computations are involved. The (T_{cpu}) value is zero if no computations take place while its maximum value is at least 0.067000 seconds. The measured T_{cpu} is relatively noticeable when comparable with the measured I/O latencies; this seemly small T_{cpu} makes the application be more CPU bounded than I/O. Because informed prefetching aims to improve I/O performance of I/O-intensive applications, we intend to use small T_{cpu} values to make our benchmarks I/O intensive. We also consider the worst case where T_{cpu} is set to zero.

• **Validate T_{driver}):** In the 1KB case, we use dynamic memory allocation to test the time needed to allocate 1KB in the memory. We tried using the same method for a 200MB

block size, but we were unable to find a reasonable value for (T_{driver}) - all the measured results are too large to represent any real-world case. Consequently, we choose to use static memory allocation for 200MB blocks; we implement a trivial application to statically allocate 200MB blocks in the main memory. Our measurements show that T_{driver} in the static memory allocation cases is negligible.

• **Put it all together - Validate $T_{cpu} + T_{hit} + T_{driver}$:** As mentioned previously, the worst case value of $T_{cpu} + T_{hit} + T_{driver}$ is very small. The purpose of this validation process is to find the lowest possible approximation for this small value, which causes our benchmarks to be more I/O bounded and enables us to investigate performance improvement offered by our prefetching solutions. Applying the (T_{cpu}) + (T_{hit}) + (T_{driver}) value in our simulation studies described in the subsequent Chapters is to estimate reasonable and practical CPU and I/O processing overheads. In reality, of course, this estimated value is application dependent and may be large.

Figure 3.3 shows the range of the (T_{cpu}) + (T_{hit}) + (T_{driver}) value. In our experiments, we use the minimum value 0.037 seconds, indicating that T_{hit} is the most critical factor that must be recorded each time. As seen in the TIP research, the value of $T_{cpu} + T_{hit}) + T_{driver}$ is always constant and; therefore, we also make a similar assumption that this value used in our simulation studies is a constant.

Estimate $T_{cpu} + T_{hit} + T_{driver}$ when block size is 10MB. When the block size is large, the worst-case value of $T_{cpu}) + T_{hit} + T_{driver}$ is the minimum value of T_{hit}, which dominate performance for I/O-intensive applications. Table 3.1 allows us to derive T_{hit} when block size is 10MB. T_{hit} for the 10MB-block-size case approximately equals to 0.00192 seconds; the value of $T_{cpu} + T_{hit} + T_{driver}$ when the block size is 10MB is around 0.00192 seconds.

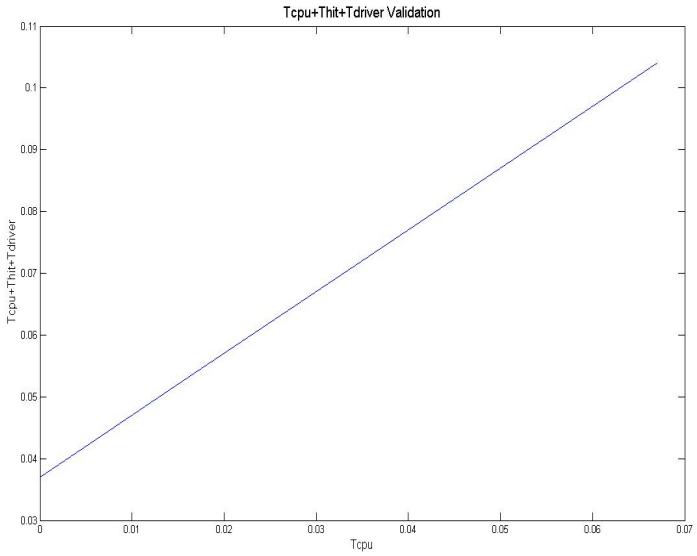

Figure 3.3: (T_{cpu}) + (T_{hit}) + (T_{driver}) values range from 0.037- 0.104 seconds. We will consider the smallest value.

3.3.6 System Parameters validation

Our goal is to validate important system parameters used in simulation studies presented in the subsequent chapters. These critical system parameters include I/O access latencies like T_{hdd-ss}, $T_{hdd-cache}$, and $T_{ss-cache}$ when block size is 10MB and 200 MB, respectively. The other parameter validated in this part of study are $T_{hdd-network-cache}$ and $T_{ss-network-cache}$ for the 200MB-block-size cases.

Figures 3.4, 3.5, and 3.6 validate that T_{hdd-ss} approximately equals to 0.122 seconds, $T_{hdd-cache}$ is about 0.12 seconds , and $T_{ss-cache}$ is in the neighbourhood of 0.052 seconds when block size is 10 MB.

Figures 3.7, 3.8, and 3.9 show that T_{hdd-ss} is around 4.5 seconds, $T_{hdd-cache}$ is estimated to be 2.3 seconds, and $T_{ss-cache}$ is measured to be 1.5 seconds when data block is set to 200 MB. Please note that in Chapters 4 and 5, we will evaluate two of our prefetching solutions - iPipe and IPO - using 200MB-block-size cases.

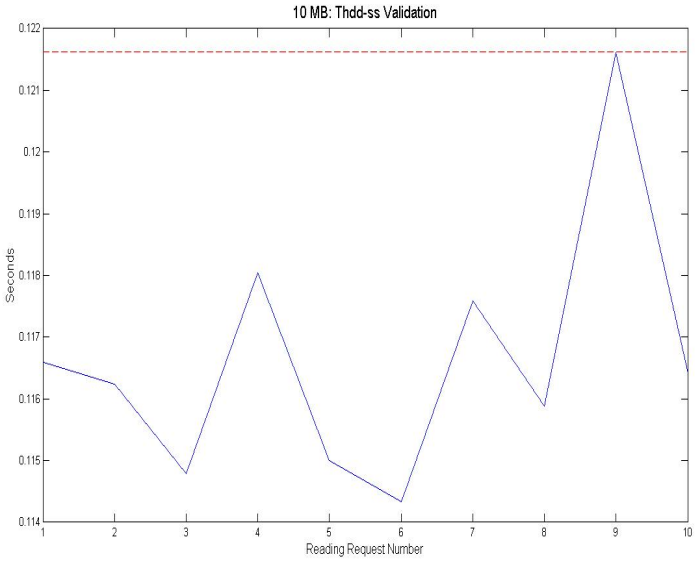

Figure 3.4: 10 MB: Estimated T_{hdd-ss} is 0.122 seconds.

To validate $T_{hdd-network-cache}$ and $T_{ss-network-cache}$, we calculate average read latencies using a network storage system in our laboratory at Auburn. In our testbed, we use SCP (i.e., Secure copy) to read data from a remote node's HDD and SSD. The read latencies of accessing a 200MB block from a remote HDD ($T_{hdd-network-cache}$) and from a remote SSD ($T_{ss-network-cache}$) are estimated as 4.43 and 4.158 seconds, respectively. For distributed storage systems (see Chapter 6), we will use these validated system parameters in addition to parameter T_{hdd-ss} that equals approximately 4.5 seconds.

For our iPipe solution (see Chapter 4), we test 1KB-block-size cases and provide preliminary experimental results. We validate various I/O bandwidth of HDDs and SSDs by using their specifications data released by the vendors.

3.3.7 Limited Parallel I/O Bandwidth

One may assume that parallel storage systems have enough I/O parallelism and bandwidth (see, for example, [1]). This assumption implies that there is no I/O congestion during

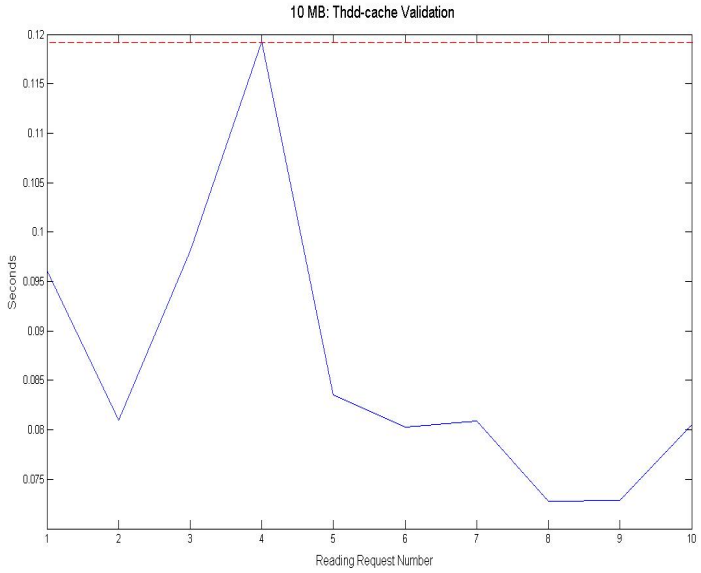

Figure 3.5: 10 MB: Estimated $T_{hdd-cache}$ is 0.12 seconds.

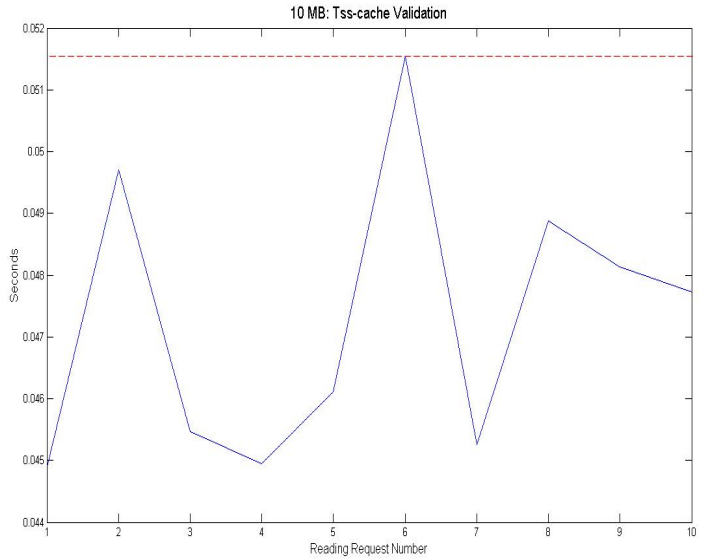

Figure 3.6: 10 MB: Estimated $T_{ss-cache}$ is 0.052 seconds.

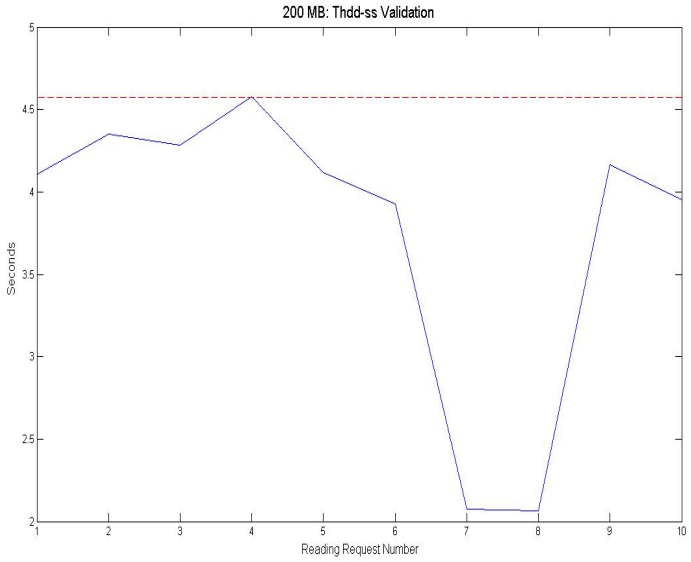

Figure 3.7: 200 MB: Estimated T_{hdd-ss} is 4.5 seconds.

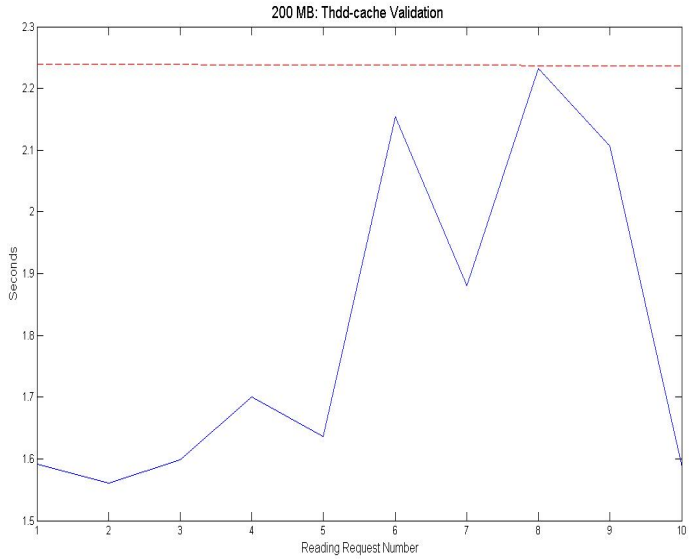

Figure 3.8: 200 MB: Estimated $T_{hdd-cache}$ is 2.3 seconds.

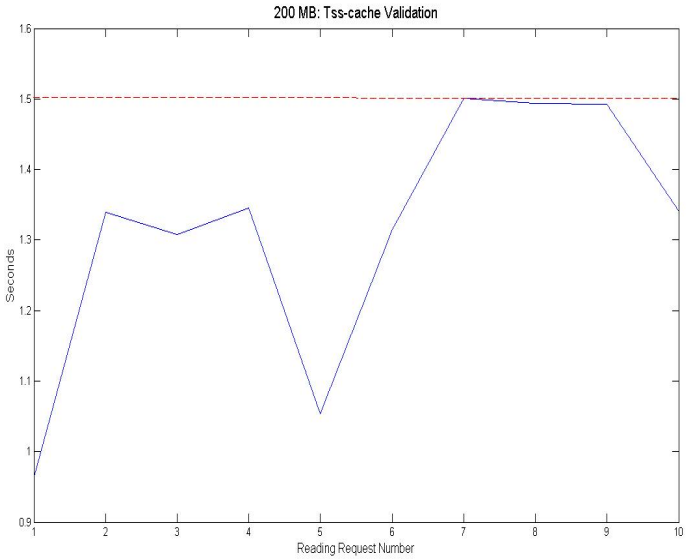

Figure 3.9: 200 MB: Estimated $T_{ss-cache}$ is 1.5 seconds.

data prefetching processes when a parallel storage system scales up. When the parallel storage system is not scalable due to limited parallel I/O bandwidth, the above assumption may become invalid. In other words, limited I/O bandwidth make the parallel storage system liable to I/O congestions (e.g. I/O queuing). This problem is more pronounced if the concurrent number of read requests exceeds a threshold. While designing the iPipe scheme, we relies on the above assumption, which is relaxed in the development of our IPO and IPODS prefetching mechanisms. Figure 3.10 shows concurrent read requests to the same disk are noticeably affected by the limited parallel I/O bandwidth.

3.4 Summary

We started this Chapter by justifying a number of important research assumptions related to caching, initial data allocations, bandwidth limitations, and I/O writes. Before present our pipelined prefetching solutions in the subsequent Chapters, we validated critical system parameters used in our simulation studies. The validate processes described in this Chapter ensure that parameters used in our simulations represent real-world storage systems,

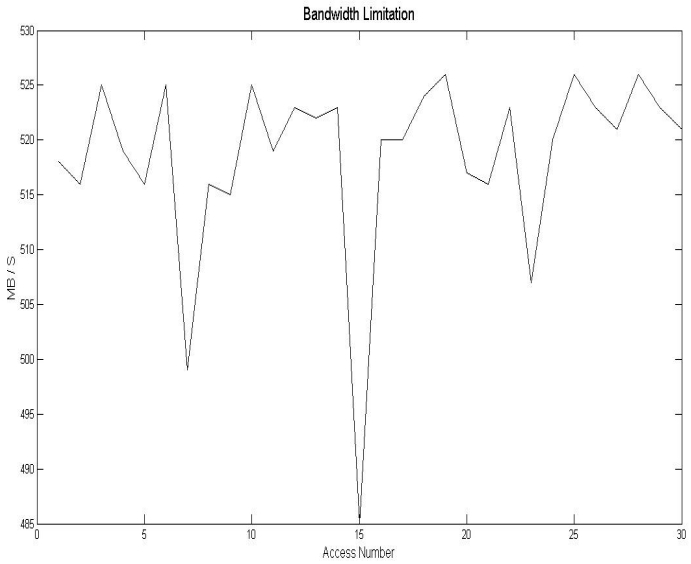

Figure 3.10: Limited Parallel I/O Bandwidth. Concurrent read requests are noticeably affected by the limited parallel I/O bandwidth.

because the parameters showed in this Chapter were estimated using the storage systems in our laboratory at Auburn.

Chapter 4

iPipe: An Pipelined and Informed Prefetching for Multi-Level Storage Systems

4.1 Overview

The informed prefetching algorithm - TIP (see [1] for details) - reduces applications' stalls and elapsed time. TIP makes use of applications' hints of future I/O accesses. In this Chapter, we extend the informed prefetching technique by creating a pipeline in which we split the informed prefetching process into a set of independent prefetching steps among the multiple storage levels. In Chapter, we show that how multiple prefetching mechanisms can coordinate to prefetch hinted blocks in parallel.

We call our new informed prefetching technique powered by a pipeline as iPipe. Because iPipe is the first step toward building a prefetching pipeline, we assume that parallel storage systems have scalable I/O bandwidth and parallelism. This assumption implies that there is no I/O congestion. This assumption of scalable I/O enables a prefetching mechanism to issue a large number of concurrent read requests without introducing long I/O queues. In the first Section of this Chapter, we explain the motivations and objective of iPipe. Then, we describe the design and implementation issues of the iPipe prefetching mechanism. Finally, the performance of iPipe is quantitatively evaluated.

4.2 Motivations and Objectives

The iPipe prefetching technique is motivated by the following three observation:

- There are growing needs of applying multi-level storage systems in data centers. Multi-level storage systems consist of multiple-level storage devices, where the uppermost level - the most expensive device in the storage systems - has the best I/O performance.

- The informed prefetching technique can utilize I/O access hints offered by applications to boost I/O performance. However, informed prefetcing has not yet been incorporated in multi-level storage systems.

- Multiple-level storage devices can be accessed in parallel. Such I/O parallelisms among multiple storage levels allows hinted blocks to be simultaneously fetches from the multiple storage levels to reduce prefetching times.

The overall objectives of the iPipe scheme are summarized as follows:

- To minimized time spent in prefetching data from storage systems by moving hinted blocks to upper-levels of storage devices in multi-level storage systems.

- To reduce application stalls and execution times.

- To decrease prefetching horizon (i.e., prefetching distance that leads to zero stalls).

- To balance prefetching and caching and to distribute I/O buffers between the prefetching and caching mechanisms.

4.3 Design Issues in iPipe

A study [1] shows that long disk latency and low I/O bandwidth increase applications' stalls. Aggressive prefetching data blocks may inevitably pollute cache buffers. To address this problem, in this section we describe design issues of a pipelined prefetching mechanism - iPipe - for hinted blocks in multi-level storage systems.

Compared with existing prefetching schemes, iPipe introduces a set of salient features: support application-disclosed I/O access hints, multiple informed prefetching mechanisms, incorporate the cost-benefit model in multi-level storage systems, and support a prefetching pipeline across multiple storage devices. Before presenting the iPipe implementation details, we first outline a high-level overview of iPipe's hardware and software architectures.

4.3.1 Architecture of iPipe

Hardware Architecture of iPipe

The system consists of an application (user) that can provide hints of its future accesses. The hierarchy from top to bottom consists of a prefetching buffer cache and an array of two levels of storage devices (Solid State Drive (SSD) and Hard Disk Drive (HDD)) as shown in figure 4.1. As we descend in the hierarchy, the disk read latency increases.

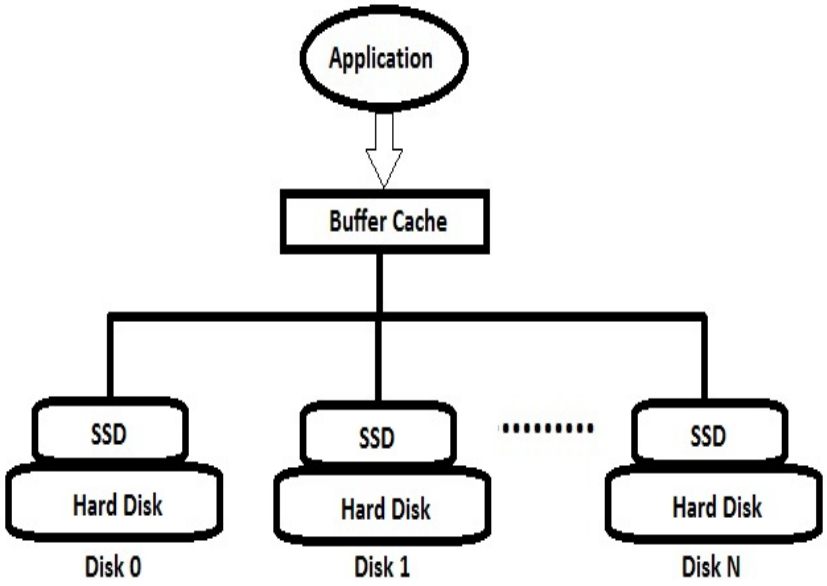

Figure 4.1: iPipe system hardware architecture. Consists of an array of multi-level disks.

Software Architecture of iPipe

Figure 4.2 depicts a high-level iPipe software architecture. The software architecture contains software modules that (1) implement the informed prefetching algorithm (see, for example, [1]), (2) control buffer cache, and (3) perform informed prefetching. In a multi-level storage system, an informed prefetching request is first issued to a solid state disk (SSD). If the hinted block is not residing in SSD, the block will be fetched from a hard disk (HDD). In the software architecture, there is a module that manages the SSD and HDD. The

iPipe mechanism take applications' future accesses hints as input and performs prefetching pipelining in the multiple levels of storage.

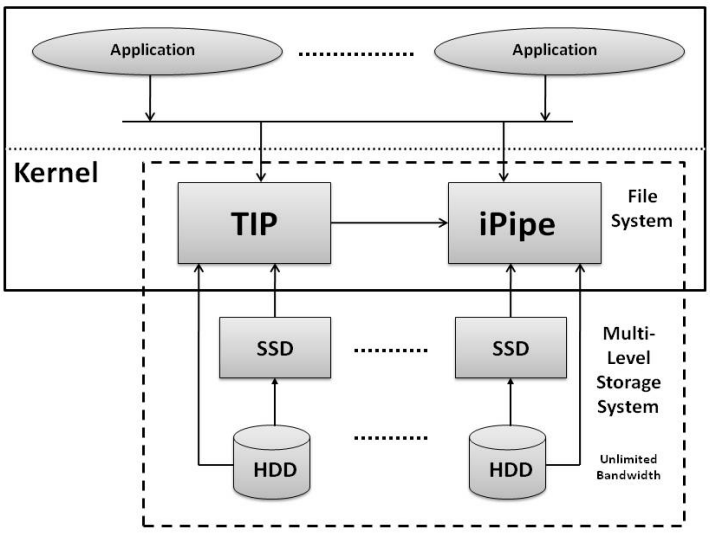

Figure 4.2: High-level design of the iPipe software architecture for a multi-level storage system. The multi-level storage system consists of two levels - SSDs and HDDs.

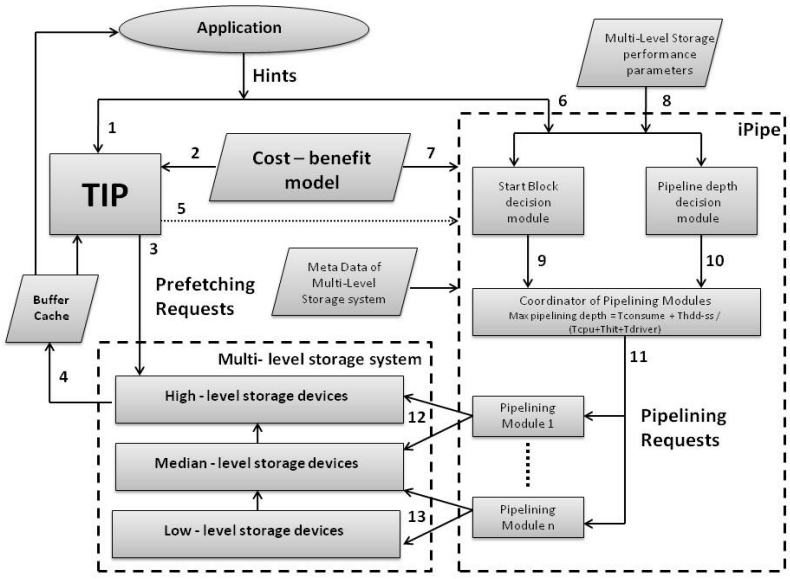

Figure 4.3: Detailed design of the iPipe software architecture for a three-level storage system. An application provides hints to both TIP and iPipe. The system performance parameters are passed to iPipe to calculate the pipelining starting block and depth. iPipe keeps fetching hinted data blocks to the highest level. Since the storage system's bandwidth is high enough, iPipe is able to fetch most of the hinted blocks. TIP uses a cost benefit model to determine the number of prefetch buffers. Hinted data blocks are fetched from the storage system to the buffer cache.

Figure 4.3 details the design of the iPipe software architecture for a three-level storage system. Figure 4.3 shows that an application provides hints to both TIP and iPipe. The system performance parameters are passed to iPipe to calculate the pipelining starting block and depth. iPipe keeps fetching hinted data blocks to the highest level. Since the storage system's bandwidth is high enough, iPipe is able to fetch most of the hinted blocks. TIP uses a cost benefit model to determine the number of prefetch buffers. Hinted data blocks are fetched from the storage system to the buffer cache.

4.3.2 Assumptions

In chapter 3, we discussed our research's assumptions in details. In this subsection, let us highlight the most important assumptions related to iPipe.

We assume that prefetching I/O requests can be processed in parallel. Further, storage devices have sufficient I/O bandwidth and enough I/O parallelism, meaning that there is no I/O congestions. Similar assumptions can be found in the TIP study [1]. We assume that there is storage systems can be well scale up to accommodate a large number of I/O requests being fetched in parallel [23]. Therefore, we conclude that if a list of hinted data blocks are requested at the same time, the blocks can be fetched without incurring excessive I/O congestion.

As a conservative assumption, hinted data blocks are initially allocated to HDDs thanks to the large capacity of the HDDs. It is noteworthy that I/O performance of multi-level storage systems can be improved if hinted blocks are initially placed in SSDs rather than HDDs.

In a multiple level storage system, a small portion of SSD space is reserved for retaining copies of the prefetched data. Instead of migrating data from HDDs to SSDs, iPipe keeps original copies at the SSD level while fetching duplicated copies to SSDs. In addition, iPipe adjust SSD space reserved for prefetching.

4.4 The iPipe Algorithm

In this section, we first present a mathematical model for iPipe. Then, we describe the concept of prefetching horizon. Next, the iPipe algorithm is proposed. Finally, we demonstrate the usage of iPipe through a concrete example.

4.4.1 Stalls and Disk Read Latencies

When an application starts, the informed prefetching module (i.e., TIP [1]) in iPipe assigns a number of buffers for prefetching hinted blocks based on the cost-benefit model (see [1] for details on the cost-benefit model).

Let (X_{cache}) be the number of prefetching buffers in the buffer cache, TIP initiates (x_{cache}) prefetching requests. Assuming the storage system provides sufficient I/O bandwidth and

parallelism, multiple hinted data blocks can be fetched in parallel without incurring I/O congestion. Let T_{disk} denote disk read latency (i.e., time spent in fetching a block from a disk to a buffer cache). The application initially stalls for T_{disk} until the hinted blocks arrive at the buffer cache.

After the hinted blocks are ready in the buffer, the application begins consuming these blocks. When each hinted blocks is consumed in the buffer, a new prefetching request is issued. In the case of few buffers being used for prefetching to be consumed in T_{disk} time period, newly requested blocks will cause the application to stall every X_{cache} accesses for the time calculated by equation 4.1. Patterson *et al.* an example [1] where a system has three buffers for prefetching, the disk read latency is five, and the time that the application needs to consume a single buffer was one. This example shows that a stall of two time units will arise for every three accesses.

$$T_{stall}(X_{cache}) = \frac{T_{disk} - X_{cache}(T_{cpu} + T_{hit} + T_{driver})}{X_{cache}} \quad (4.1)$$

where;

$T_{stall}(X_{cache})$ [1]: average stall of prefetching

T_{cpu}: computational time

T_{hit}: time to read a data block from the buffer cache

T_{driver}: time to allocate a buffer in the buffer cache

We denote $T_{cpu} + T_{hit} + T_{driver}$ as a single time unit spent in consuming one buffer in the buffer cache. This single time unit is also the time between each two subsequent I/O read requests.

[1] R. Patterson, Hugo, G. Gibson, D. Stodolsky, and J. Zelenka: Informed prefetching and caching, *In Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79-95, CO, USA, 1995.

In a multi-level storage system, we consider two types of T_{disk} - $T_{hdd-cache}$ and $T_{ss-cache}$. $T_{hdd-cache}$ is the time spent fetching a block from HDD; $T_{ss-cache}$ is the time interval for loading a block from SSD. We define T_{hdd-ss} as the time spent copying a block from an HDD to an SSD. Our empirical results show i.e., $T_{ss-cache}$ is smaller than $T_{hdd-cache}$, indicating that loading data from SSD is much faster than from HDD.

4.4.2 Prefetching Horizon

Prefetching Horizon is a distance by which informed prefetching leads an application to experience zero stall time [1]. Equation 4.2 represents a way of calculating prefetching horizon. In the previous example, $P(T_{cpu})$ equals five time units because T_{disk} equals five and $T_{cpu} + T_{hit} + T_{driver}$ equals one. Consequently, if the prefetching mechanism assigns five buffers for prefetching, stalls time will drop down to zero.

$$P(T_{cpu}) = \frac{T_{disk}}{(T_{cpu}+T_{hit}+T_{driver})} \quad (4.2)$$

where;

$P(T_{cpu})$ [1] : prefetching horizon

4.4.3 The Pstart and Pdepth Algorithms

Two important modules in the iPipe architecture are (1) a module to determine the starting block to be prefetched and (2) a module to compute the prefetching depth. The what follows, we describe the algorithms implemented in these two modules.

[1]R. Patterson, Hugo, G. Gibson, D. Stodolsky, and J. Zelenka: Informed prefetching and caching, *In Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79-95, CO, USA, 1995.

When the first informed prefetching request is initiated, iPipe begins fetching hinted blocks from HDD to SSD. iPipe relies on the following algorithm called Pstart - to decide which hinted blocks in the first block to be loaded.

The Pstart Algorithm:

```
xcachecounter = 0
accesstime = - (T_cpu + T_hit + T_driver)
for blockcounter = 1 to T_hdd−ss / (T_cpu + T_hit + T_driver) do
  xcachecounter ++
  accesstime += (T_cpu + T_hit + T_driver)
  if accesstime ≥ T_hdd−ss then
    Pstart = blockcounter
    return Pstart
  end if
  if xcachecounter = X_cache then
    if T_stall−hdd(Xcache) > 0 then
      accesstime += T_stall−hdd(Xcache)
    end if
    if accesstime ≥ T_hdd−ss then
      Pstart = blockcounter + 1
      return Pstart
    end if
    xcachecounter = 0
  end if
end for
```

Algorithm 1. returns the future hinted access position that iPipe starts pipelining from. It depends on the T_{hdd-ss} value and ($T_{stall-hdd}(X_{cache})$) stalls that takes place from the HDD to the buffer cache during the beginning of the informed prefetching process . The algorithm calculates the position of the future hinted access that will be accessed far enough in the future for it to be read from the HDD to the SSD (see equation 4.3).

The above Pstart algorithm is very important, because loading hinted blocks so earlier can pollute I/O buffers and fetching hinted blocks at a late stage can increase applications stall time.

The *Pstart* algorithm determines the first hinted block to be prefetched given a future access pattern. Pstart depends on the T_{hdd-ss} value and ($T_{stall-hdd}(X_{cache})$) stalls that takes place from HDD to buffer cache during the beginning of the informed prefetching process. The Pstart algorithm calculates the position in the future access list. Thus, Pstart needs

to decide which hinted blocks in the list should be fetched in advance from HDD to SSD to reduce application stall times. The Pstart algorithm is guided by Equation 4.3 to pick the most appropriate hinted blocks to fetch from HDD.

While the iPipe mechanism is utilizing the TIP technique [1] to prefetch hinted blocks from SSDs to main memory, iPipe manges to fetch hinted blocks from HDDs to SDDs. Prefetching hinted blocks by two parallel prefetchers in a pipelining manner can increase chances of retrieving hinted blocks in SSDs rather than in HDDs.

The application begins consuming the arrived data and new prefetching requests may take place before the iPipe brings hinted blocks from HDDs to SSDs. The new informed prefetching requests may also cause stalls until the requests are honored, especially when X_{cache} is smaller than the prefetching horizon. If $T_{stall-hdd(Xcache)}$ is the average stall time of fetching blocks from HDDs, iPipe starts loading hinted blocks from HDDs within or after time T_{hdd-ss} minus the stall time.

$$iPipe_{start} = \frac{T_{hdd-ss}}{(T_{cpu} + T_{hit} + T_{driver})} - (\frac{\frac{\frac{T_{hdd-ss}}{(T_{cpu}+T_{hit}+T_{driver})}}{X_{cache}} \times T_{stall-hdd}(X_{cache})}{(T_{cpu} + T_{hit} + T_{driver})}) \quad (4.3)$$

where;

$iPipe_{start}$: iPipe pipelining start

Algorithm 2: Prefetching pipelining depth

totaltime = T_{hdd-ss} + $T_{consume-ss}$
xcachecounter = 0
accesstime = 0
for blockcounter = 1 to totaltime / (T_{cpu} + T_{hit} + T_{driver}) **do**
  xcachecounter ++
  accesstime += (T_{cpu} + T_{hit} + T_{driver})
  if accesstime ≥ totaltime **then**
    Pdepth = Pstart + blockcounter - 1
    return Pdepth

```
    end if
    if xcachecounter = X_cache then
      if T_stall−ss(Xcache) > 0 then
        accesstime += T_stall−ss(Xcache)
      end if
      if accesstime ≥ totaltime then
        Pdepth = Pstart + blockcounter -1
        return Pdepth
      end if
      xcachecounter = 0
    end if
  end for
```

Algorithm 2. returns the depth of iPipe's pipeline that is needed to maintain the pipelining process working smoothly. Starting from Pstart, iPipe initially pipelines a number of hinted future accesses that equals the prefetching horizon of the disk read latency from HDD to SSD plus additional blocks to cover a single SSD block's consuming time (see equation 4.5). $T_{consume-ss}$ is the time needed to consume a block in the SSD. $T_{stall-ss}(X_{cache})$ represents the application's stalls when the the data is prefetched from the SSD to the buffer cache. These stalls reduce the pipelining depth.

The above *Pdepth* algorithm chooses pipelined prefteching depth, which is defined as the number of hinted block to be fetched from HDDs to SSDs. The goal of the *Pdepth* algorithm is to prefetch as many hinted blocks as possible without polluting buffer caches in SSDs.

After performing the *Pstart* and *Pdepth* algorithms, the iPipe mechanism have a full list of the most appropriate blocks to be prefetched from HDDs to SSDs. The number of hinted blocks to be fetched largely depends on two factors: (1) prefetching horizon of the disk read latency from HDD to SSD, and (2) the time spent consuming a single SSD block in the buffer cache (see equation 4.5). Let $T_{consume-ss}$ be the time needed to consume a block in SSD and $T_{stall-ss}(X_{cache})$ be an application's stalls when the the data is prefetched from the SSD to the buffer cache. These stalls reduce the number of hinted blocks to be prefetched from HDDs to SSDs.

When it comes to the decision of how many hinted blocks to be loaded from HDDs to SSDs, iPipe initially fetches the number of hinted blocks that equal the prefetching horizon plus additional blocks to cover the time needed to consume a single block in the SSD. We

assume that ($T_{consume-ss}$) equals the latency of a SSD read to the buffer cache ($T_{ss-cache}$). The hinted blocks residing in the SSDs may cause stalls until the blocks become available in buffer caches in main memory. This takes place when X_{cache} is smaller than the prefetching horizon of reading data from SSDs to buffers in the main memory. $T_{stall-ss(Xcache)}$ is the average stall time of prefetching from the SSD, and these stalls reduce the pipelined prefetching depth. Equation 4.4 calculates the maximum number of concurrent prefetching requests that may take place when using iPipe. Equation 4.5 shows the calculation of the maximum iPipe's pipelined prefetching depth. This occurs when TIP is assigning X_{cache} buffers that equal the prefetching horizon of reading data from HDD to buffer cache; no stalls can take place during the pipelined prefetching process.

Our experimental results show that pipelined prefetching depth is a small value, indicating that iPiple does not need to reserve a large buffers in SSDs for pipelined prefetching requests.

$$iPipe_{maxreads} = \frac{T_{hdd-cache} + T_{hdd-ss} + T_{consume-ss}}{(T_{cpu} + T_{hit} + T_{driver})} (4.4)$$

where;

$iPipe_{maxreads}$: iPipe's maximum number of concurrent reading requests

$T_{hdd-cache}$: Latency of reading a single data block from HDD to the cache

T_{hdd-ss}: Latency of reading a single data block from HDD to SSD

$T_{consume-ss}$: The time that TIP takes to consume a single data block from the SSD

$$iPipe_{depth} = \frac{T_{hdd-ss} + T_{consume-ss}}{(T_{cpu} + T_{hit} + T_{driver})} (4.5)$$

where;

$iPipe_{depth}$: iPipe's pipelining depth

T_{hdd-ss}: Latency of reading a single data block from HDD to SSD

$T_{consume-ss}$: The time that TIP takes to consume a single data block from the SSD

4.4.4 Stalls, Elapsed Time, Prefetching Horizon, and Prefetching Benefit

In the example shown earlier, we assume that T_{hdd-ss}, $T_{consume-ss}$, and $T_{hdd-cache}$ equal eight, four, and five time units, respectively. iPipe initially issue a pipelined prefetching request for a hinted data block with position number seven through position number fifteen (inclusive). Giving sufficient I/O bandwidth and parallelism, informed prefetching requests as well as pipelined prefetching requests will be completed (i.e., arrive at buffer caches in the main memory and SSDs, respectively) within $T_{hdd-cache}$) and $T_hdd - ss$, respectively. The application stalls at the beginning of $T_{hdd-cache}$, which is five time units.

$$T_{pf}(X_{cahce}) = \begin{cases} X_{cache}=0 & -(T_{cpu} + T_{hit} + T_{driver}) \\ X_{cache} < P(T_{cpu}) & \frac{-T_{disk}}{X_{cache}(X_{cache}+1)} \\ X_{cache} \geq P(T_{cpu}) & 0 \end{cases} \quad (4.6)$$

Where;

$T_{pf}(X_{cahce})$ [1]: reduction in service time

The application then begins to consume the prefetched blocks in the buffer cache residing in the main memory. Every time when a buffer is consumed, an informed prefetching request is issued for the next hinted block. When a hinted block has been preloaded from HDDs to SDDs, iPipe can quickly fetches the hinted block from SSDs into the main memory without

[1]R. Patterson, Hugo, G. Gibson, D. Stodolsky, and J. Zelenka: Informed prefetching and caching, *In Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79-95, CO, USA, 1995.

making a trip to HDDs. After a hinted block is consumed in the SSDs in $T_{consume-ss}$ time, iPipe triggers the next pipelined prefetching request.

Disk accessing time - T_{disk} - from the perspective of informed prefetching becomes $T_{ss-cache}$, which is less than $T_{hdd-cache}$. Hinted blocks are fetched from HDDs to SSDs prior to being accessed by the prefetching module fetching the blocks from SSDs into main memory. Prefetching hinted blocks from HDDs and keeping the hinted blocks in SSDs reduce the number of application stalls (see Equation 4.1).

Figure 4.4 builds on the previous example and shows the average stall time when a fixed number of buffers are used with pipelined prefetching. In contrast, Figure 4.5 reveals average stall when using a fixed number of buffers in the buffer cache for pipelined prefetching (the regular case in which no pipelining is used). When iPipe is employed, the stall time and elapsed time are 9 and 39 time units, respectively. When it comes to the same storage system without iPipe, the stall time and elapsed time are 16 and 46 time units, respectively. In this example, iPiple reduce the application's elapsed time by 15.2%.

The TIP study [1] offers a cost-benefit model to calculating benefits (e.g., reduction in I/O service time) of assigning one additional buffer for prefetching. The benefit is that the stall time is reduced from $T_{stall(Xcache)}$ to $T_{stall(Xcache+1)}$. The benefit calculation makes use of the prefetching horizon, which changes with the T_{disk} value. In the case that X_{cache} is smaller than the prefetching horizon P(T_{cpu}), the benefit of adding an extra buffer is calculated by Equation 4.6.

It takes $T_{hdd-cache}$ time to fetch the first hinted block from an HDD to a buffer cache in the main memory. When subsequent hinted blocks are fetched from HDDs to SSDs in a pipelining manner, the prefetching time drops from $T_{hdd-cache}$ to $T_{ss-cache}$). Pipelined prefetching reduces the prefetching horizon and stalls. As a result, iPipe increase the benefits of using an additional buffer for informed prefetching.

Access Number	Time (One Time Step = ($Tcpu$ + $Thit$ + $Tdriver$))																			
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
1	I	O	O	O	O	AX														
2	I	-	-	-	-	A	X													
3	I	-	-	-	-	A		X												
4						I	-	-	O	O	AX									
5							I	-	-	-	-	AX								
6								I	-	-	-	-	AX							
7	II	-	-	-	-	-	-	-	a		I	-	-	O	AXx					
8	II	-	-	-	-	-	-	-	a			I	-	-	-	AXx				
9	II	-	-	-	-	-	-	-	a				I	-	-	-	AXx			
10	II	-	-	-	-	-	-	-	a						I	-	-	O	AXx	
11	II	-	-	-	-	-	-	-	a							I	-	-	-	AXx
12	II	-	-	-	-	-	-	-	a								I	-	-	-
13	II	-	-	-	-	-	-	-	a										I	-
14	II	-	-	-	-	-	-	-	a											I
15	II	-	-	-	-	-	-	-	a											
16															II	-	-	-	-	-
17																II	-	-	-	-
18																	II	-	-	-
19																			II	-
20																				II

I: Initiate prefetch to buffer cache II: Initiate prefetch to SSD -: Prefetch in progress

A: Arrived in buffer cache a: Arrived in SSD X: Consumed from buffer cache

x: Consumed from SSD O: Stall

Access Number	Time (One Time Step = ($Tcpu$ + $Thit$ + $Tdriver$))																			
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
12	AXx																			
13	-	O	AXx																	
14	-	-	-	AXx																
15	I	-	-	-	AXx															
16	-	-	al	-	-	O	AXx													
17	-	-	-	al	-	-	-	AXx												
18	-	-	-	-	al	-	-	-	AXx											
19	-	-	-	-	-	-	al	-	-	O	AXx									
20	-	-	-	-	-	-	-	al	-	-	-	AXx								
21	II	-	-	-	-	-	-	-	al	-	-	-	AXx							
22			II	-	-	-	-	-	-	-	al	-	-	O	AXx					
23				II	-	-	-	-	-	-	-	al	-	-	-	AXx				
24					II	-	-	-	-	-	-	-	al	-	-	-	AXx			
25							II	-	-	-	-	-	-	-	al	-	-	O	AXx	
26								II	-	-	-	-	-	-	-	al	-	-	-	AXx

Figure 4.4: Average stall when using iPipe and a fixed number of buffers for pipelined prefetching. $T_{hdd-cache} = 5$, $T_{hdd-ss} = 8$, $T_{ss-cache} = 4$, $X_{cache} = 3$. The first stall is 5 time units. Before the first hinted block is fetched from HDD into SSD, the application stalls for $T_{stall-hdd}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 2$ time units every 3 accesses. When hinted blocks are retrieved in the SSD, the application stalls for $T_{stall-ss}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 1$ every 3 accesses.

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
1	I	O	O	O	O	AX														
2	I	-	-	-	-	A	X													
3	I	-	-	-	-	A		X												
4						I	-	-	O	O	AX									
5							I	-	-	-	-	AX								
6								I	-	-	-	-	AX							
7											I	-	-	O	O	AX				
8												I	-	-	-	-	AX			
9													I	-	-	-	-	AX		
10																I	-	-	O	O
11																	I	-	-	-
12																		I	-	-

I: Initiate prefetch to buffer cache
II: Initiate prefetch to SSD
-: Prefetch in progress
A: Arrived in buffer cache
a: Arrived in SSD
X: Consumed from buffer cache
x: Consumed from SSD
O: Stall

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
10	AX																			
11	-	AX																		
12	-	-	AX																	
13	I	-	-	O	O	AX														
14		I	-	-	-	-	AX													
15			I	-	-	-	-	AX												
16						I	-	-	O	O	AX									
17							I	-	-	-	-	AX								
18								I	-	-	-	-	AX							
19											I	-	-	O	O	AX				
20												I	-	-	-	-	AX			
21													I	-	-	-	-	AX		
22																I	-	-	O	O
23																	I	-	-	-
24																		I	-	-

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
22	AX																			
23	-	AX																		
24	-	-	AX																	
25	I	-	-	O	O	AX														
26		I	-	-	-	-	AX													

Figure 4.5: Average stall when using a fixed number of buffers for pipelined prefetching. $T_{hdd-cache} = 5$, $X_{cache} = 3$. The first stall is for 5 time units, because all data blocks are read from the HDD. The application stalls for $T_{stall-hdd}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 2$ time units every 3 accesses.

4.4.5 The iPipe Algorithm

The pseudo code of the iPipe algorithm is presented below. First, the iPipe algorithm invokes the *Pstart* and *Pdepth* algorithms (see the earlier sections of this Chapter) to determines the most appropriate hinted blocks to be fetched from HDDs to SDDs. Second, iPiple initiates prefetching requests to HDDs. When a hinted block is consumed from SSD (i.e., the hinted block is fetched from SSD to main memory), iPipe starts fetching subsequent blocks from HDDs to SSDs. Due to limited I/O bandwidth, hinted blocks may not be fetched to the main memory before the blocks are consumed. In this case, iPipe shrinks its pipelined prefetching depth each time by 1 (i.e., reducing the number of hinted blocks fetched from HDDs to SSDs). Before prefetching any hinted block directly from HDD, the iPipe algorithm checks if a hinted block has been fetched to SDD or is being fetched to SDD. If the block is residing in SSD, the pipelined prefetching request will be discarded.

The iPipe Algorithm

```
Pstart = call prefetching start block
Pdepth = call prefetching depth
while informed prefetching do
  if bandwidth shortage then
    shrink the pipelining depth by 1
  end if
  if pipelined data block is altered then
    discard the pipelined data block
  end if
  if is the first prefetching then
    for block = Pstart To block = Pdepth do
      pipeline block to SSD
    end for
  else if SSD buffer is consumed then
    if the following data block is not already in the SSD then
      pipeline the following data block to SSD
    end if
  end if
end while
```

Algorithm 3. iPipe algorithm. It calls prefetching start and depth algorithms to calculate Pstart and Pdepth values. Then, it initiates a pipelining request from the HDD to the SSD for the data blocks from Pstart to Pdepth. Every time a pipelined block is consumed from the SSD by the informed prefetching, iPipe pipelines the next one to the SSD. Bandwidth

shortage means that TIP is not able to receive its prefetching requests on time (from any level). This may be caused by the pipelining process, writes, power problems, or many other reasons. In that case, iPipe shrinks its pipelining depth each time by 1 until the shortage is released. The algorithm checks if a data block that has been pipelined or is in the pipelining process is changed (written) before it is requested by TIP. In this case, the pipelined copy will be discarded to maintain consistency. The algorithm also checks if the data is already in the SSD. In this case, its pipelining request is discarded.

4.5 Performance Evaluation

We implement the iPipe algorithm in a trace-driven simulator written in C++. The metrics evaluated in the simulated two-level (i.e., SSD and HDD) storage system (see Figure 4.1) include elapsed time, prefetching stall time, prefetching horizon, as well as the benefit of adding an additional buffer for informed prefetching. These metrics are evaluated by varying the numbers of prefetching buffers in the buffer cache. We compare an iPipe-enabled prefetching mechanism against the same system without deploying iPipe. Data blocks are initially placed in HDD stores the data. The iPipe mechanism coordinates two prefeching modules: the first one fetches hinted blocks from HDDs to SSDs, the second one uses TIP [1] to further fetch hinted blocks from SSDs to main memory.

4.5.1 System Setup

In our simulation studies, we use two LASR traces: machine01 (LASR1) and machine06 (LASR2) [97][98], which consist of 11686 and 51206 I/O read system calls, respectively. Without loss of generality, we assume that each I/O read system call requests an entire data block. As a conservative assumption, the average I/O arrival interval is $T_{cpu} + T_{hit} + T_{driver}$, representing the worst case. If the interval is larger than $T_{cpu} + T_{hit} + T_{driver}$, iPipe can achieve even better I/O improvement.

We choose to focus on a two-level storage system containing a set of SSDs and HDDs. Our preliminary results prove that reading 1-KB block from SSD is faster than from HDD. We configure the simulated two-level storage system using the devices' specifications provided

by the disk vendors. To fetching blocks in an efficient way, we use a log file system to store hinted blocks sequentially. In this way, a list of hinted blocks can be retrieved from the log file system without incurring any disk rotational delay. This is practical to apply the log file system to support the iPipe mechanism, because target I/O-intensive applications of iPipe are read-intensive applications where most data blocks are read only. It is natural and straightforward to place all the read-only blocks sequentially using the log file system.

All the system parameters used in our simulator are validated by the testbed in our laboratory at Auburn. Please refer to Chapter 3 for details on the validation process.

4.5.2 Preliminary Results

The simulated two-level storage system is connect to a host computer with a 2GHz core 2 duo processor. The simulated host computer uses Kingston's RAM (800MHz) [90] as buffer caches. The data bus is 64- bit; the buffer caches offer I/O throughput at the rate of 6400 MB/s. The two-level storage system is comprised of Intel's X25-E Extreme SATA solid-state drive [91], which has a read throughput of 250 MB/s and WD Caviar Green HDD offering a read throughput of 190 MB/s assuming [94]. The validation of the above parameters can be found in Chapter 3.

The system parameters T_{cpu}, T_{hit}, T_{driver} are set to 4 nanosecond, 0.15 ms, and 0.06 us, respectively. The T_{cpu} value is derived from a trivial application that measures the time needed to perform simple computations on (1 KB) of data. T_{hit} is derived from the memory read latency specification for 1KB data [90]. T_{driver} - derived from a trivial application - is the time needed to allocate 2000 unique 8 KB data blocks in the main memory. Combining these values, it takes $T_{cpu} + T_{hit} + T_{driver}$ time (i.e., 0.2 ms) for the application to consume a buffer in the main memory.

Data is initially stripped in the HDD layer. The read latencies of 1KB data blocks from SSDs and HDDs are 4 and 5.2 microseconds, respectively. The write latency of 1KB blocks to SSDs is 5.8 microseconds. When reading blocks from HDDs to SDDs, reading from HDDs

Table 4.1: The number of informed prefetching requests issued to HDDs when the LASR1 and LASR2 traces are evaluated.

$\mathbf{X}_{cahce}$	**1**	**3**	**5**	**7**	**9**	**11**	**13**
LASR1 With iPipe	2	6	9	11	13	15	17
LASR1 Without iPipe	11676	11676	11676	11676	11676	11676	11676
LASR2 With iPipe	2	6	9	11	13	15	17
LASR2 Without iPipe	51206	51206	51206	51206	51206	51206	51206
$\mathbf{X}_{cahce}$	**15**	**17**	**19**	**21**	**23**	**26**	
LASR1 With iPipe	19	21	23	25	27	30	
LASR1 Without iPipe	11676	11676	11676	11676	11676	11676	
LASR2 With iPipe	19	21	23	25	27	30	
LASR2 Without iPipe	51206	51206	51206	51206	51206	51206	

and writing to SSDs are interleaved. Thus, the total time latency T_{hdd-ss} of fetching 1KB block from HDD and caching the block in SSD equals to 5.8 microseconds. There is no write latency to the buffer cache in main memory (see, for example, [1]).

Because hinted blocks are fetched from both SSDs and HDDs concurrently, Equation 4.4 calculates the maximum number of prefetching requests issued to HDDs. Equation 4.4 indicates that the largest possible number of prefetching requests for both SSDs and HDDs at a given time is 75 data blocks. To guarantee serving this maximum number of requests from the HDD to the buffer cache and the SSD, we assume there is sufficient I/O bandwidth and parallelism [1]. Equation 4.2 determines the prefetching horizon of reading data from HDDs to the buffer cache is 26 if all data blocks are prefetched from HDD without employing iPipe. Using the above system information, we evaluate the performance of iPipe by varying the number of buffers for informed prefetching from 1 to 26.

Our simulation results show the impact of number of buffers on the total elapsed time (i.e., application's execution time) and read latency time. Figures 4.6 and 4.7 show the total elapsed time of the two traces. The experimental results reveals that iPipe reduces elapsed time by up to 23% compared with the non-iPipe case. Figure 4.8 shows the average informed prefetching read latency of the two evaluated I/O traces. iPipe significantly improve I/O performance of the two-level storage system, because only a few hinted blocks are fetched directly from HDDs whereas most hinted blocks are prefetched and cached in SSDs.

Table 4.2: Service time is reduced when one extra buffer is added for prefetching.

$\mathbf{X}_{cahce}$	**1**	**3**	**5**	**7**	**9**	**11**	**13**
LASR1 With iPipe	21041	3895.92	1559.44	835.51	519.56	354.83	256.4
LASR1 Without iPipe	28044.8	5063.6	2027.63	1086.04	675.58	461.57	333.19
LASR2 With iPipe	102292.3	17064.3	6826.5	3656.7	2276.5	1551.14	1124.65
LASR2 Without iPipe	133165	22208.3	8871.8	4760.3	2959	2017.9	1462.04
$\mathbf{X}_{cahce}$	**15**	**17**	**19**	**21**	**23**	**26**	
LASR1 With iPipe	195.004	152.605	122.999	0.2	0.2	0	
LASR1 Without iPipe	253.79	198.21	160.201	131.202	110.403	0	
LASR2 With iPipe	852.92	669.52	539.015	0.2	0.2	0	
LASR2 Without iPipe	1108.84	871	700.98	576.36	482.355	0	

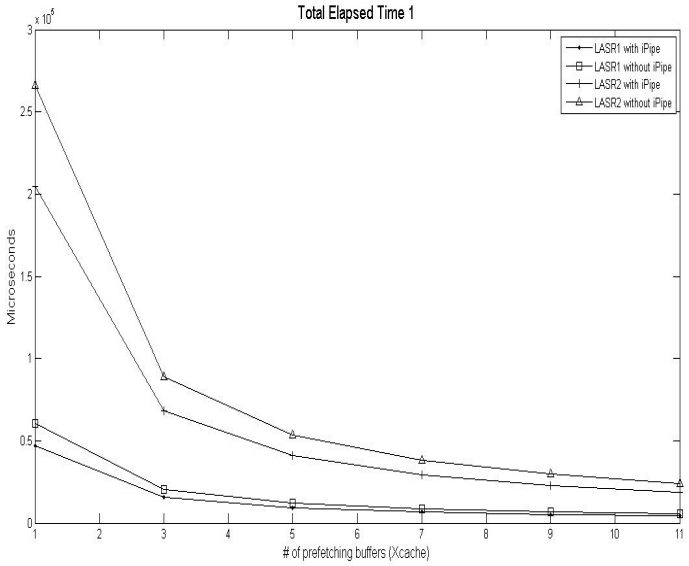

Figure 4.6: Total elapsed time when the number of prefetching buffers is set from 1 to 11. iPipe reduces the elapsed time.

Table 4.1 shows the number of data blocks prefetched from HDDs at the beginning of the simulation. When iPipe is deployed, we observe that very few hinted blocks are directly fetched from HDDs because most hinted blocks are prefetched and cached in SSDs for fast accesses. The prefetching horizon in our simulations is initially 26 data blocks, but drops to 20 data blocks. In the non-iPipe case, all informed prefetching requests must bring data directly from HDDs; the prefetching horizon always equals 26 data blocks.

When using iPipe, the first few informed prefetching requests are read directly from HDDs due to a cold start (i.e., hinted blocks have not been preloaded in SSDs). The rest

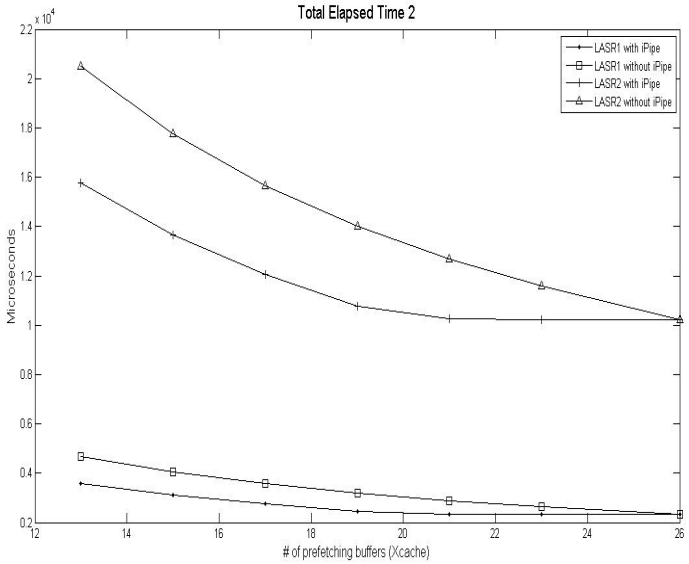

Figure 4.7: Total elapsed time when the number of prefetching buffers is set from 13 to 26. iPipe reduces the elapsed time.

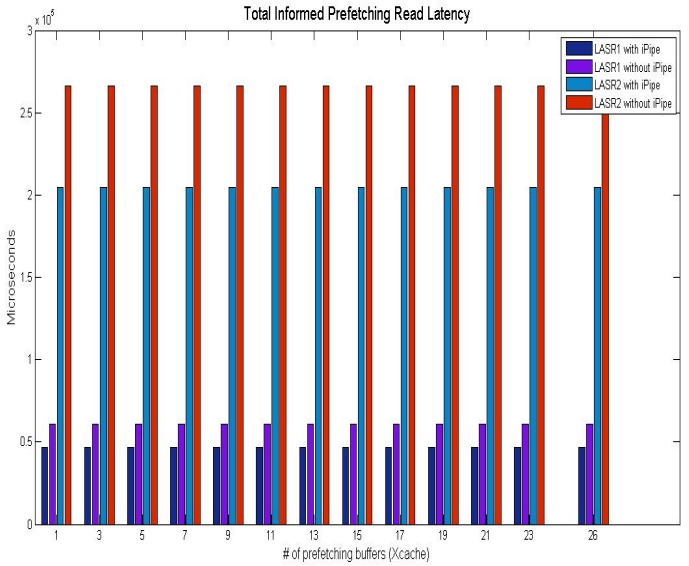

Figure 4.8: Total informed prefetching read latency. iPipe reduces the read latency.

Table 4.3: Prefetching Horizon P(T_{cpu}) equals 26 data blocks distance while iPipe is not used. P(T_{cpu}) drops to 20 data blocks distance when iPipe is used.

$\mathbf{X}_{cahce}$	**P(Tcpu)**
iPipe	20
non - iPipe	26

hinted blocks are retrieved from SSDs thanks to the pipelined prefetching mechanism that preloads the blocks from HDDs to SSDs. Since ($T_{ss-cache}$) is smaller than ($T_{hdd-cache}$), the prefetching horizon P(T_{cpu}) drops. Our simulation studies (see Table 4.3) show that when reading from the HDDs and the SSDs, P(T_{cpu}) equals 26 and 20 data blocks distance respectively. Table 4.1 shows the number of hinted blocks that are prefetched directly from the HDDs.

iPipe reduces informed prefetching disk read latency, which in turn leads to fewer application stalls during the prefetching processes. iPipe also reduces the benefit of adding an additional buffer for prefetching. Figures 4.9 and 4.10 show that the total stall time measured in microseconds is approximately the same for both traces. In all cases, the stalls decrease as the (X_{cache}) increases. However, iPipe shows fewer stalls when using a certain number of (X_{cache}). If the value of (X_{cache}) is larger than 20 data blocks, the stall time become zero because the number of informed prefetching requests exceed P(T_{cpu}) (as Table 4.4).

As the disk read latency, prefetching horizon, and stalls decrease, the reduction in service time (benefit) from using additional buffers for prefetching also decreases. Table 4.2 illustrates the decrease in service time for each (X_{cache}) value, that is when adding additional buffer used for prefetching. In all cases, the prefetching benefit decreases with the increase of (X_{cache}). iPipe shows less need for more buffers, however, when (X_{cache}) is larger than 20, which makes the reduction in service time become zero. In other words, there is no benefit from using more buffers than P(T_{cpu}). This conclusion is consistent with that reported in the TIP study [1].

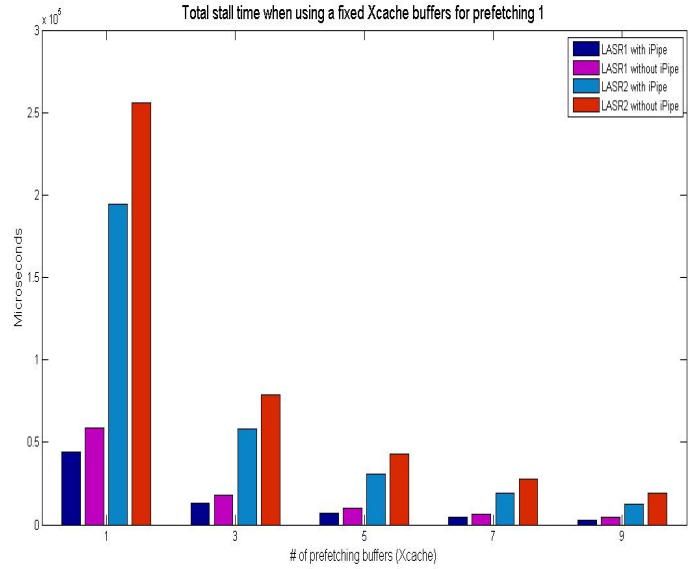

Figure 4.9: Total stall time when the number of prefetching buffers is set from 1 to 9. iPipe reduces the stall time.

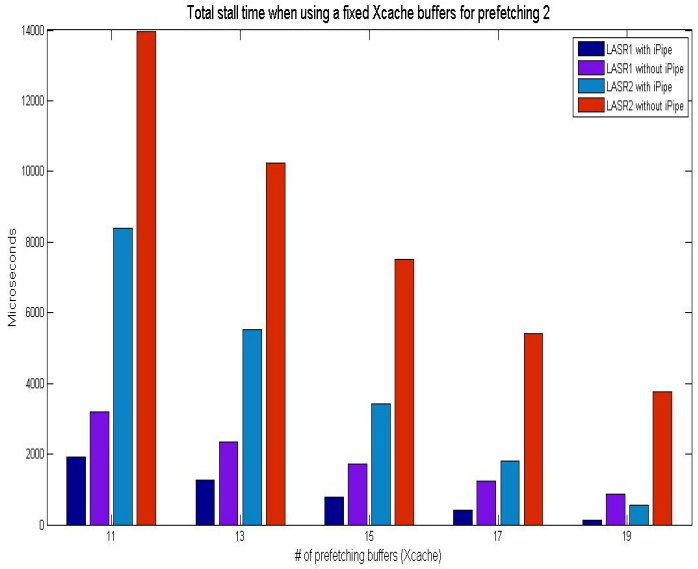

Figure 4.10: Total stall time when the number of prefetching buffers is set from 11 to 19. iPipe reduces the stall time.

Table 4.4: Total stall time when the number of prefetching buffers is set from 21 to 26. iPipe reduces the stall time.

$\mathbf{X}_{cahce}$	**21**	**23**	**26**
LASR1 With iPipe	1	0.6	0
LASR1 Without iPipe	556	304.802	0
LASR2 with iPipe	1	0.6	0
LASR2 Without iPipe	2438	1335.57	0

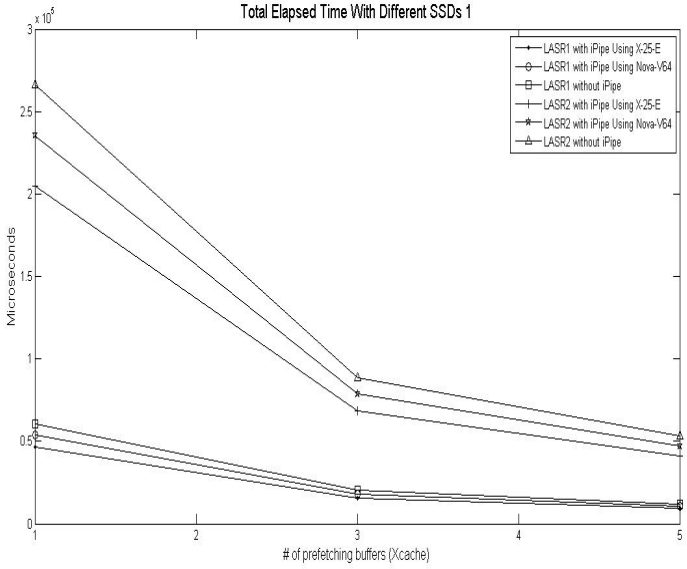

Figure 4.11: Total elapsed time when the number of prefetching buffers is set from 1 to 5. iPipe reduces the elapsed time in both Nova-V64 and Intel X25-E SSDs. Elapsed time is less when Intel X25-E SSD is tested because it is faster than Nova-V64 SSD.

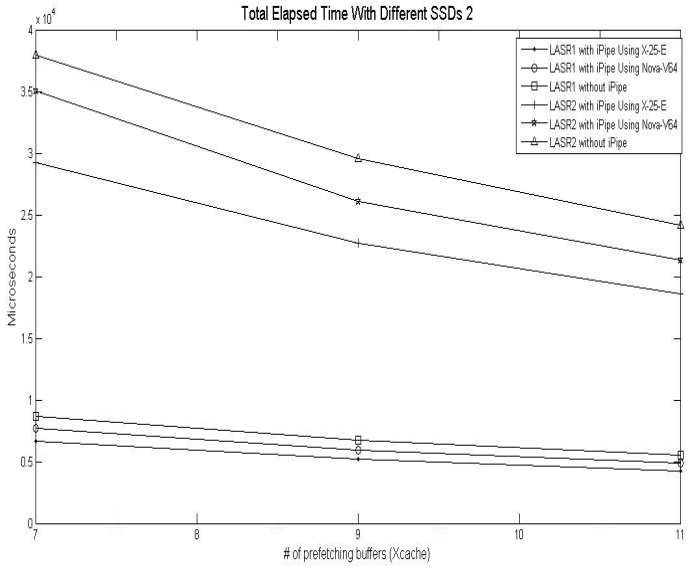

Figure 4.12: Total elapsed time when the number of prefetching buffers is set from 7 to 11. iPipe reduces the elapsed time in both Nova-V64 and Intel X25-E SSDs. Elapsed time is less when Intel X25-E SSD is tested because it is faster than Nova-V64 SSD.

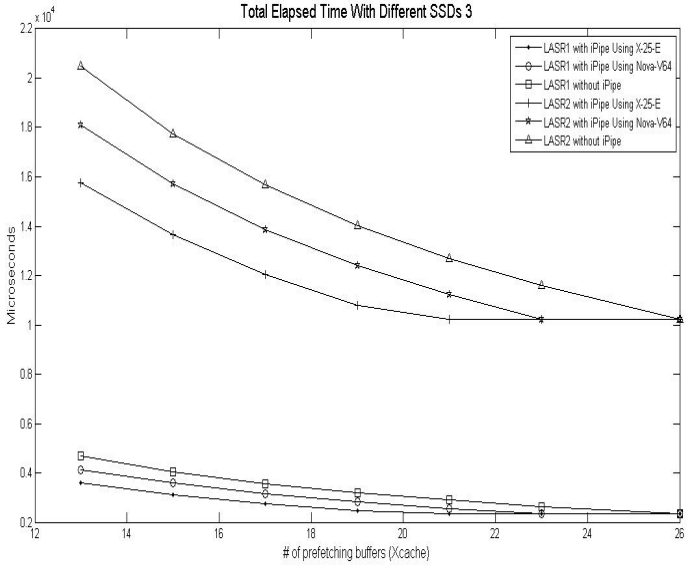

Figure 4.13: Total elapsed time when the number of prefetching buffers is set from 13 to 26. iPipe reduces the elapsed time in both Nova-V64 and Intel X25-E SSDs. Elapsed time is less when Intel X25-E SSD is tested because it is faster than Nova-V64 SSD.

Previous results indicate that iPipe's performance depends on both read and write performances of SSDs. Reading performance reduces the disk read latency for informed prefetching while writing performance effects the efficiency of the pipelined prefetching process. Because of this, performance varies when using different SSDs with different read performances. We simulate iPipe using various SSDs and compared the performance in terms of stalls and elapsed times. Figures 4.11, 4.12, and 4.13 compare of both two LASR traces when using [92] the Nova Series V64 Solid-State Hard Drive and Intel X25-E Extreme SATA Solid-State Drive. The Nova V64 SSD has a reading throughput of 215 MB/s and writing throughput of 130 MB/s. The latency of reading a (1KB) data block from it is 4.6 microseconds; the latency of writing a (1KB) block to the SSD equals 7.6 microseconds. When reading from the HDD to the SDD, we assumed that both reading from the HDD and writing to the SSD are interleaved making the total time latency for reading a (1KB) block from the HDD and having it available in the SSD (T_{hdd-ss}) equal to 7.6 microseconds.

4.5.3 Validated Performance Evaluation

We configure our simulator using the validated system parameters reported in Chapter 3. For the LASR 1 and LASR 2 traces, we assume that each exclusive access system call record represents a single 10 MB data block stored in the simulated storage system. In addition to 10-MB blocks, 200- MB data blocks are also test. In doing so, we can evaluate the impact of block size on iPipe performance. We assume that the sequence of the I/O reading system calls represents parallel I/O prefetching requests for the data. We also assume that each buffer can accommodate one data block.

When block sizes equals to 10 MB and 200 MB, (T_{cpu} + T_{hit} + T_{driver}) is at least 0.00192 and 0.037 seconds, respectively (see also Chapter 3). Data is initially stripped in the HDD layer of the two-level storage system. Reading a 10 MB data block from the SSD and the HDD takes 0.052 and 0.12 seconds, respectively. The total time spent reading a 10 MB block from the HDD and placing it in the SSD (T_{hdd-ss}) equals to 0.122 seconds. When it

comes to a 200 MB block, it takes 1.5 and 2.3 seconds to read the block from the SSD and the HDD, respectively. The total time spent reading a 200 MB block from the HDD and having it available in the SSD (T_{hdd-ss}) is 4.5 seconds. Time spent accessing buffer cache in the main memory is negligible (see [1] for a similar assumption).

Since two prefetching modules in iPipe work concurrently at two storage levels, the maximum number of read I/O requests issued to the HDD is calculated by Equation 4.4. According to this equation, the largest possible number of concurrent read requests from both prefetching modules in iPipe is 154 in the case of 10 MB data blocks and 225 in the case of 200 MB ones. This is because 200 MB blocks increase I/O latencies. Our results show that system parameters are the main factor affecting iPipe's behaviour and requirements.

To guarantee maintaining this maximum number of reading requests, we assume enough I/O bandwidth and parallelism (i.e., no I/O congestion). A similar assumption is justified in [1]. According to Equation 4.2, the prefetching horizon of reading data from HDD to the buffer cache equals 63 for both 10 MB and 200 MB data blocks; therefore, we vary the number of buffers from 1 to 63 for both the 10 MB and the 200 MB cases.

The simulation results show that the total elapsed time (a.k.a., application execution time) when using fixed numbers (X_{cahce}) of prefetching buffers is reduced by up to 56% in the 10 MB case and 34% in the 200 MB case. Our results show that SSD exhibits better performance compared to the non-iPipe system when the block size is 10 MB. Figures 4.14, 4.15, and 4.16 show the total elapsed time when the LASR traces are tested for the 10-MB-block-size case. Figures 4.17, 4.18, and 4.19 show the same test for the 200 MB block size. As the number of X_{cahce} buffers increases, the performance gap between iPipe and non-iPipe cases diminishes because the prefetching module (i.e., TIP) in the upper level becomes more efficient and able to reduce the application stalls. iPipe shows the most benefit when the upper-level prefetching module is using small X_{cahce}; (e.g. 1) this is considered the most critical situation for informed prefetching. iPipe allows the upper-level prefetcher find only a few of its reads in HDDs and the rest in SSDs.

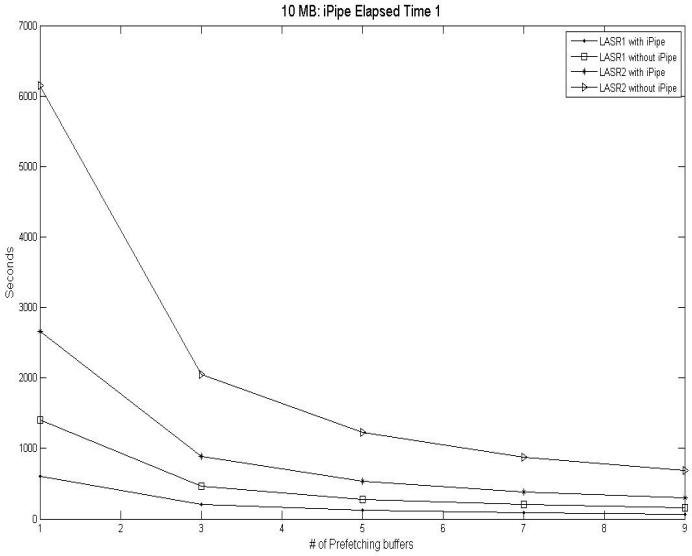

Figure 4.14: Block size = 10 MB. Total elapsed time when the number of buffers is set from 1 to 9. iPipe reduces the elapsed time.

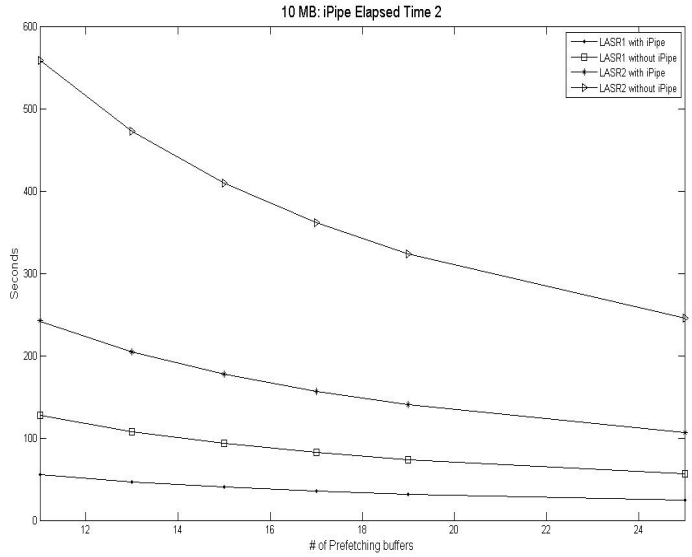

Figure 4.15: Block size = 10 MB. Total elapsed time when the number of buffers is set from 11 to 25. iPipe reduces the elapsed time.

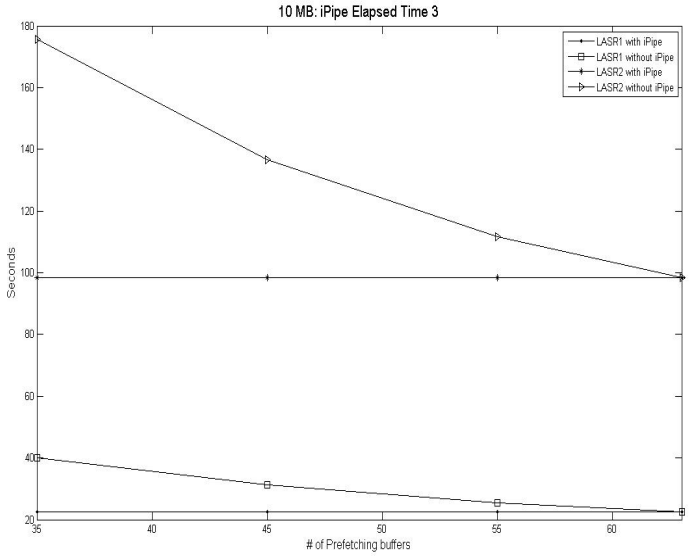

Figure 4.16: Block size = 10 MB. Total elapsed time when the number of buffers is set from 35 to 63. iPipe reduces the elapsed time.

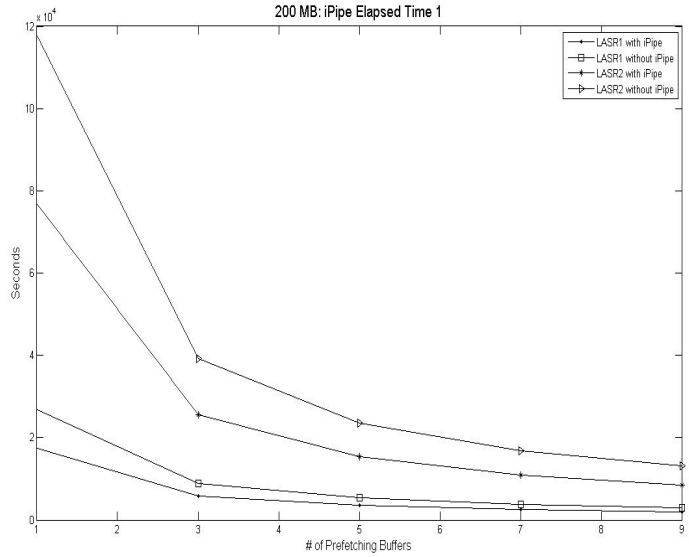

Figure 4.17: Block size = 200 MB. Total elapsed time when the number of buffers is set from 1 to 9. iPipe reduces the elapsed time.

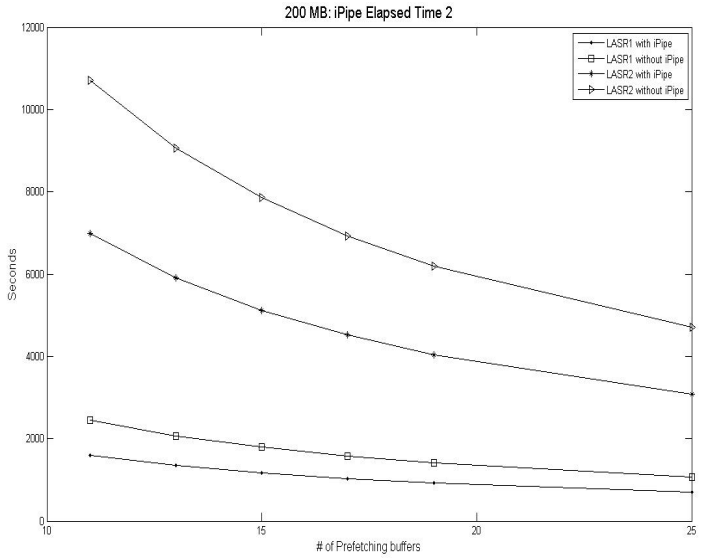

Figure 4.18: Block size = 200 MB. Total elapsed time when the number of buffers is set from 11 to 25. iPipe reduces the elapsed time.

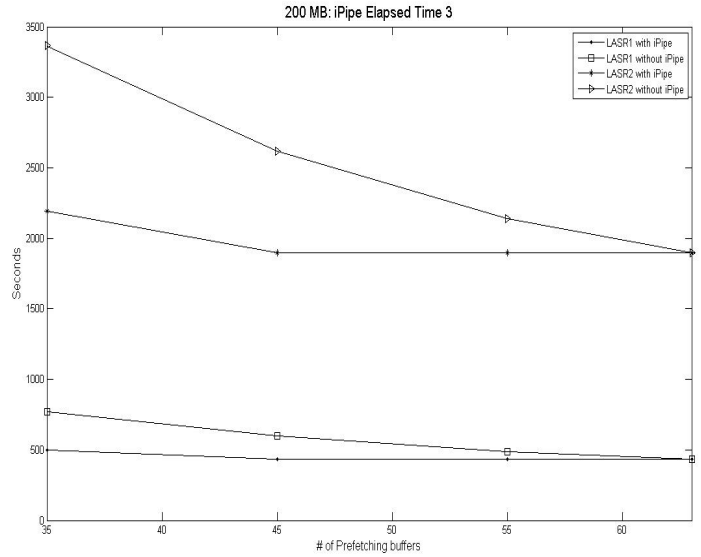

Figure 4.19: Block size = 200 MB. Total elapsed time when the number of buffers is set from 35 to 63. iPipe reduces the elapsed time.

Table 4.5: Data block size = 10 MB. The position of the first data block to be prefetched.

$\mathbf{X}_{cahce}$	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**
Depth	3	7	11	15	19	23	27	31
$\mathbf{X}_{cahce}$	**17**	**19**	**25**	**35**	**45**	**55**	**63**	
Depth	35	39	51	64	65	65	65	

Table 4.6: Data block size = 10 MB. The depth of the pipelined prefetching when using different X_{cahce} values. Small depth is needed when few X_{cahce} buffers are used. The maximum depth = 91. iPipe needs to assign the maximum depth for pipelined prefetching starting from X_{cahce} = 35, because the reading stalls from SSD at that point = 0

$\mathbf{X}_{cahce}$	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**
Depth	4	12	20	28	36	43	49	55
$\mathbf{X}_{cahce}$	**17**	**19**	**25**	**35**	**45**	**55**	**63**	
Depth	61	66	85	91	91	91	91	

It is clear that iPipe offers better performance improvement to the 10 MB case than to the 200 MB case. Small blocks benefit better from iPipe, because SSD's I/O performance is notably better than HDD when the block size is small. The time needed to read a single data block from the HDD to the SSD becomes very large in the 200 MB case.

iPipe can utilize multiple prefetching modules at different storage levels to load hinted blocks without needing a large pipelining depth, especially when few prefetching buffers X_{cahce} are assigned (e.g. 1). If X_{cahce} is closer to the prefetching horizon, iPipe needs a larger depth in the SSD for its pipelined prefetching process. If that depth is not available, iPipe should only be used when the X_{cahce} value is small (the case where TIP shows very poor performance). Tables 4.5 and 4.6 show the the first data block to be pipelined and the pipeline depth when using different X_{cahce} values for a block size of 10 MB. Tables 4.7 and 4.8 show the same performance trend for 200 MB data blocks. iPipe assumes enough bandwidth, implying that storage systems are sufficiently scaled and have redundant storage resources. Because of this assumption, the pipelined prefetching depth does not need to be large. Maximum pipelined prefetching space is 910 MB in the 10-MB-block-size case and 32600 MB (32.6 GB) in the 200-MB-block-size case.

Table 4.7: Data block size = 200 MB. The position of the first data block to be prefetched.

$\mathbf{X}_{cahce}$	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**
Depth	4	10	16	22	28	34	40	46
$\mathbf{X}_{cahce}$	**17**	**19**	**25**	**35**	**45**	**55**	**63**	
Depth	52	58	76	106	123	123	123	

Table 4.8: Data block size = 200 MB. The depth of the pipelined prefetching when using different X_{cahce} values. Small depth is needed when few X_{cahce} buffers are used. The maximum depth = 163. iPipe needs to assign the maximum depth for pipelined prefetching starting from X_{cahce} = 45, because the reading stalls from SSD at that point = 0

$\mathbf{X}_{cahce}$	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**
Depth	4	12	20	28	36	44	52	60
$\mathbf{X}_{cahce}$	**17**	**19**	**25**	**35**	**45**	**55**	**63**	
Depth	68	76	100	140	163	163	163	

4.6 Summary

Informed prefetching reduces application I/O stalls and elapsed times. Disk read I/O latencies effect the performance of iPipe. For example, when the I/O latencies increase, applications encounter more stalls. Multi-level storage systems' uppermost level show better performance than the lower levels. iPipe has multiple informed prefetching modules running in parallel in a pipeline manner. The prefetchers in lower-level storage devices help in reducing I/O latencies of the prefetchers in upper-levels. Consequently, iPipe reduces stall times, application elapsed times, the prefetching horizon. iPipe also improves the benefit of using more buffers for prefetching. One weakness of iPipe proposed in this Chapter is the assumption that storage systems have sufficient I/O bandwidth and scalability. This assumption will be relaxed in the proposed prefetching solutions described in the next two Chapters.

Chapter 5

IPO: Informed Prefetching Optimization in Multi-level Storage Systems

5.1 Overview

In the previous chapter, we discussed the importance of informed prefetching for reducing an application's stalls and elapsed time. We also pointed out the motivation of a pipelined prefetching mechanism aiming to reduce the overhead of prefetching.

The study presented in the previous chapter assumes that multi-level storage systems have sufficient I/O bandwidth; however, our empirical experiments indicate that parallel storage systems may have I/O congestion (see Figure 3.10 in Chapter 3). Evidence shows that there is a maximum number of read requests being concurrently processed in a parallel storage system. In the event that an upper-level prefetching mechanism does not fully utilize available I/O bandwidth, unused I/O bandwidth can be allocated for lower-level prefetching mechanisms to bring hinted blocks from lower-level to upper-level storage in a pipeline manner. This observation suggests that the pipelining process largely depends on available I/O bandwidth for lower-level prefetchers.

The remaining parts of this chapter is organized as follows. First, we highlight the motivation and objective of a pipelined informed prefetching scheme called IPO. Second, we describe the design issues of the IPO mechanism. Third, we present a core algorithm in IPO. Last, the performance of IPO is systematically evaluated.

5.2 Motivations and Objectives

The motivations behind the design of IPO are very similar to those of iPipe (See Chapter 4.2 for details on the motivations of the iPipe study). In addition to the motivations listed

in Chapter 4.2, limited I/O bandwidth of parallel storage systems motivate of to propose the IPO solution.

I/O bandwidth of a storage system, of course, is limited. This reality means that when I/O load is high, the storage system can exhibits I/O congestions (e.g. long I/O queuing times). In particular, I/O congestion takes place if the storage resources are limited and not scalable. When it comes to a small-scale parallel storage system, only a few I/O requests can be processed in parallel. This limitation motivates us to extend the iPipe design into IPO by considering the impact of limited I/O bandwidth on our pipelined prefetching mechanisms.

The objective of this part of the study is to address the issue of limited I/O bandwidth and investigate the negative impacts of low I/O bandwidth on pipelined informed prefetching.

5.3 Design Issues in IPO

Our preliminary results (see Chapter 3) show that the bandwidth of a storage system is limited based on the available resources. Our overall goal in this part of the study is to optimize informed prefetching in a multi-level storage system with limited I/O bandwidth. The iPipe study (see the previous Chapter) offers initial evidence that a prefetching pipeline is a good approach to utilizing multiple prefetchers to preload hinted blocks in parallel among multiple storage levels. The prefetching pipeline substantially reduces time spent loading hinted blocks from the up-most storage level.

Before the designing phase of IPO, we realize that the number of pipelined prefetching requests that may be handled concurrently is limited depending on available I/O bandwidth and the size of prefetching buffers managed by the upper-level prefetchers (e.g., TIP is the upper-level prefetcher in our studies). Recall that multi-level storage systems considered in this dissertation study consist of two levels: the uppermost one uses solid state drives (SSD); the lowest one uses Hard Drives (HDD). The IPO mechanism gets hints of future I/O accesses from applications and aggressively prefetches hinted blocks from HDDs up to SDDs in a pipeline manner.

The most challenging design issue is how to efficiently utilized limited available I/O bandwidth for the prefetching module that preloads hinted blocks from HDDs to SDDs. A second design challenge is how to allow two prefetching modules to share I/O bandwidth of SSDs, which are accessed by both upper-level and lower-level prefetchers.

In our design, we address the above challenges as follows. Recall that lower-level prefetcher and upper-level prefetchers handles informed prefetching requests in parallel. The lower-level prefetcher trigger a informed prefetching request when the following conditions are met. First, the upper level SDDs have sufficient available I/O bandwidth. The lower-level prefetcher aims to improve the utilization of SSDs by preloading data from HDDs to SSDs when the I/O load of SSD is median or low. This design policy is important because a very aggressive lower-level prefetcher can slow down the performance of its upper-level prefetching counterpart. Second, fetching hinted blocks from HDDs to SSDs will not saturate the I/O bandwidth of SSDs and HDDs. We believe that the main concern is SSDs' bandwidth, because the SSDs are accessed by two prefetching modules. Compared with HDDs, SSDs are more likely to have their I/O bandwidth saturated.

5.3.1 Architecture of IPO

Before presenting the implementation details of IPO, let us first outline a high-level overview of IPO's hardware and software architectures.

Hardware Architecture of IPO

IPO shares iPipe's hardware architecture as shown in Figure 4.1. As we discussed in Chapter 4, an application accessing a storage system provides hints of future I/O accesses for IPO, which manages two prefetching modules - the upper module (e.g., TIP) prefetches hinted blocks from SSDs to buffers in main memory; the lower module prefetches hinted blocks from HDDs to SSDs. Both prefetching modules work concurrently in a pipelines manner.

Parallel storage systems investigated in our studies contain prefetching buffers as well as an array of two-level storage devices - SSDs and HDDs. Recall that the storage systems have limited I/O bandwidth and scalability.

Software Architecture of IPO

Figures 5.1 and 5.2 show IPO's software architecture. One of the core components in the IPO mechanism is the upper-level informed prefetching module or TIP that receives hints from data-intensive applications, controls buffers in the main memory, and performs informed prefetching from SDDs. When the IPO system receives an informed prefetching request, IPO uses the meta-data to decide if the hinted block is residing in an SSD before accessing a low-level HDD. If the hinted block is not available in one of the SSDs, the block must be fetched directly from a HDD. In case that the hinted block has been fetched by the low-level prefetcher from the HDD to SSD, the hinted block will be quickly loaded from SSD into the main-memory buffer.

At the heart of the IPO mechanism is a lower-level informed prefetching module that fetches hinted blocks from HDDs to SSDs. When the lower-level prefetcher triggers an informed prefetching request, the prefetcher checks if the hinted block has been cached in one of the SSDs. If the hinted block is not available in SSDs, the block is fetched from the HDD and placed in an SSD.

In addition, the IPO system gains control of both SSDs and HDDs in order to perform pipelined prefetching based on the hints offered by the applications.

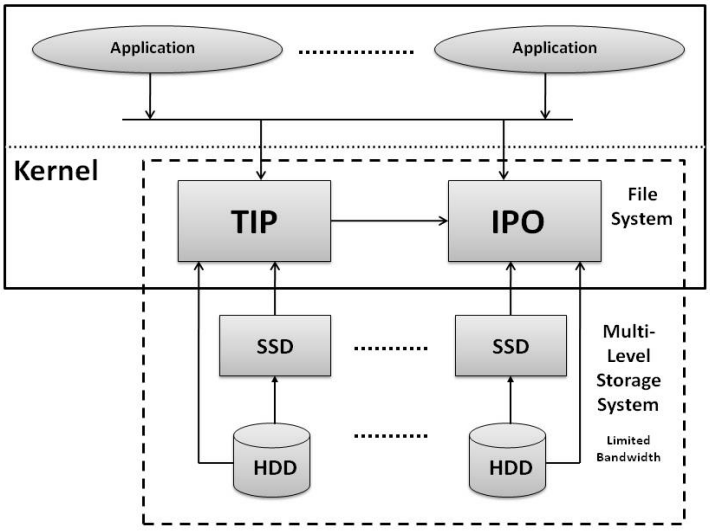

Figure 5.1: High-level design of IPO software architecture. The multi-level storage system consists of an array of two levels of SSDs and HDDs with limited bandwidth and scalability.

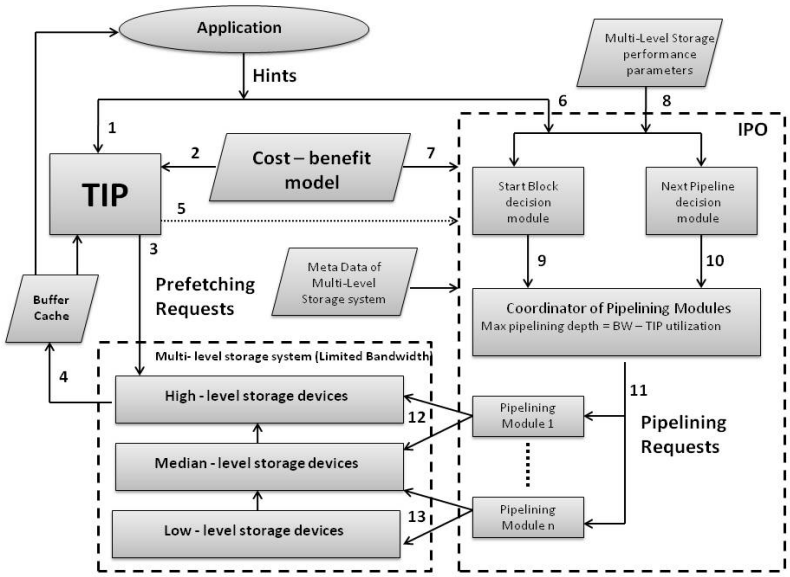

Figure 5.2: Detailed design of IPO software architecture for a three-level storage system. The application provides hints on future I/O accesses. IPO determines the appropriate hinted blocks to be fetched. IPO keeps prefetching a particular number (depends on available bandwidth) of hinted blocks to the uppermost level. The upper-level prefetcher (i.e., TIP) uses the cost/benefit model to determine the number of prefetching buffers.

5.3.2 Assumptions

In Chapter 3, we discussed our research's assumptions in details; in this section, we simply highlight the most important assumptions related to IPO.

We assume that data blocks are initially placed in HDDs. Our motivation behind this assumption is that HDDs have much larger capacity than that of SSDs. In addition, HDDs are much cheaper than SDDs. Other prefetching studies in the realm of multi-level storage systems have similar assumption that data blocks are initially stored in the lower-level of the storage hierarchy [23].

IPO reserves a small portion determined by Equation 5.1 to retain copies of prefetched data. Instead of migrating the data blocks from HDDs to SSDs, the prefetching process only move copies of the original data. This strategy works very well for read-intensive applications, because read-only data cached in upper-level storage can be quickly discarded without being moving back to a lower-level storage device. Replicas across multi-levels allow storage system to conserve bandwidth when cached blocks are evicted from I/O buffers in SSDs.

5.4 The IPO Algorithm

This section presents an algorithm that guides us in implementing the IPO mechanism for multi-level storage systems. We also give examples to demonstrate that IPO improve I/O performance of storage systems by reducing application stalls and elapsed time.

5.4.1 Definitions

IPO handles the informed prefetching process between SSDs and HDDs. When an application starts its execution, the TIP module assigns a number of buffers for prefetching (X_{cache}) based on the cost benefit model. This means that the informed prefetching module

continues to issue a number of concurrent read requests that equals to (X_{cache}). The concurrent reads utilize either a part or all of the parallel storage system's bandwidth. Based on the non-utilized portion of the I/O bandwidth, IPO assigns its pipelined prefetching depth.

Let Max_{BW} be the maximum number of read requests that may take place concurrently in the parallel storage system. In the event that TIP is assigning X_{cache} buffers for prefetching, the difference between X_{cache} and Max_{BW} is the the pipelined prefetching depth of IPO. The pipelined prefetching depth is determined by Equation 5.1; this number represents the maximum space that needs to be reserved for pipelined prefetching in the uppermost level (i.e, SSDs). As we will see in the performance evaluation section, pipelined prefetching depth dose not consume the SSD storage space.

$$P_{depth} = Max_{BW} - X_{cache} \tag{5.1}$$

where;

P_{depth}: IPO pipelined prefetching depth

Max_{BW}: Maximum bandwidth

X_{cache} : Number of prefetching buffers

The TIP module continues prefetching X_{cache} hinted data blocks concurrently. At the same time, IPO keeps fetching P_{depth} hinted data blocks from HDDs in a pipeline manner with TIP. IPO consists of two algorithms determining what the first hinted block to be fetched from HDDs and the number of subsequent blocks to be fetched. Initially, most of TIP's prefetching requests are found in the lowest level (i.e., HDDs) until the hinted data blocks arrive in the uppermost ones (i.e., SSDs). At this point, TIP fetches hinted blocks from one of the two storage levels. As more I/O bandwidth becomes available for IPO to fetch hinted blocks in a pipeline manner, more prefetching requests are handled in the uppermost level. Every time a data block is prefetched by TIP, it will be consumed from

the SSD's pipelined prefetching buffer and a new pipelined prefetching request is initiated. As more I/O bandwidth becomes available for pipelining, application stalls and elapsed time will be reduced (see Equation 4.1).

Recall that $T_{cpu} + T_{hit} + T_{driver}$ represents the system's single time unit needed for an application to consume a prefetched data block. In addition, our multi-level storage system consists of an HDD and an SSD. $T_{hdd-cache}$ is the disk read latency from HDD to buffer cache in the main memory, T_{hdd-ss} is the disk read latency from HDD to SSD, and $T_{ss-cache}$ is the disk read latency from SSD to buffer cache. $T_{ss-cache}$ should be less than $T_{hdd-cache}$.

5.4.2 The Pstart and Pnext Algorithms

IPO pipelines the prefetching processes. When informed prefetching initiates its first prefetching requests, IPO begins fetching requests from HDDs to SSDs. The first block to be fetched from HDD by IPO is calculated by the Pstart algorithm. The pipelined prefetching depth spans until a number of data blocks is calculated by Equation 5.1. When a data block is requested by TIP, the block will be consumed from the pipelining buffer in SSD after the block is fully fetched from SSD to the main memory. At this time, IPO issues a new pipelined prefetching request for another hinted data block. The position of that access's data block is calculated by algorithm (5). $T_{consume-ss}$ represents the time needed to consume a single pipelined preftching block from SSD (i.e. $T_{consume-ss} = T_{ss-cache}$).

The Pstart Algorithm: Determine the first block fetched from HDD

```
xcachecounter = 0
accesstime = - (T_cpu + T_hit + T_driver)
for blockcounter = 1 to T_hdd-ss / (T_cpu + T_hit + T_driver) do
  xcachecounter ++
  accesstime += (T_cpu + T_hit + T_driver)
  if accesstime ≥ T_hdd-ss then
    Pstart = blockcounter
    return Pstart
  end if
  if xcachecounter = X_cache then
```

```
    if T_stall−ss(Xcache) > 0 then
      accesstime += T_stall−ss(Xcache)
    end if
    if accesstime ≥ T_hdd−ss then
      Pstart = blockcounter + 1
      return Pstart
    end if
    xcachecounter = 0
  end if
end for
```

Algorithm 4. returns the hinted block position for which IPO begins pipelining. The first hinted block to be fetched from HDD depends on the T_{hdd-ss} and $T_{stall-ss}(X_{cache})$ values that may take place during the beginning of informed prefetching, assuming that all data is already in the SSD. The algorithm calculates the hinted data block's position that will be accessed after enough time to have it read from HDD to SSD.

The Pnext Algorithm:

```
totaltime = T_hdd−ss + T_consume−ss
xcachecounter = 0
accesstime = - (T_cpu + T_hit + T_driver)
for blockcounter = 1 to totaltime / (T_cpu + T_hit + T_driver) do
  xcachecounter ++
  accesstime += (T_cpu + T_hit + T_driver)
  if accesstime ≥ totaltime then
    Pnext = Pstart + blockcounter - 1
    return Pnext
  end if
  if xcachecounter = X_cache then
    if  T_stall−ss(Xcache) > 0 then
      accesstime += T_stall−ss(Xcache)
    end if
    if accesstime ≥ totaltime then
      Pnext = Pstart + blockcounter
      return Pnext
    end if
    xcachecounter = 0
  end if
end for
```

Algorithm 5. returns the next hinted data block that IPO should fetched from HDD to SDD when a previously pipelined data block is consumed from buffer in SDD. $T_{consume-ss}$ is the time to consume a block in SSD. The application stalls for at least $T_{stall-ss}(\mathrm{X}_{cache})$. These stalls reduce the prefetching depth.

Equations 5.2 and 5.1 are used in the above two algorithms to determine (1) the first blocks to be fetched from HDD to SDD and (2) the number of the subsequent blocks to be fetched from HDD to SDD.

$$IPO_{start} = \frac{T_{hdd-ss}}{(T_{cpu} + T_{hit} + T_{driver})} - (\frac{\frac{\frac{T_{hdd-ss}}{(T_{cpu}+T_{hit}+T_{driver})}}{X_{cache}} \times T_{stall-ss}(X_{cache})}{(T_{cpu} + T_{hit} + T_{driver})}) \quad (5.2)$$

where;

IPO_{start}: The first block to be fetched by IPO from HDD.

Figures 5.3 and 5.4 illustrate an example of IPO working with the TIP module. Max_{BW} is equals 5 and X_{cache} equals 2. IPO uses the remaining 3 slots for pipelined prefetching. Figures 5.5 and 5.6 show the non-IPO case. In the IPO case, the stalls and elapsed time of the application are reduced.

5.4.3 The IPO Algorithm

Algorithm 3: IPO

```
Pstart = call pipelining start block
Max_BW = Maximum number of concurrent reading requests
Pdepth = Pstart + (Max_BW - X_cache) - 1
while informed prefetching do
  if bandwidth shortage then
    shrink the pipelining depth by 1
  end if
  if pipelined data block is altered then
    discard the pipelined data block
  end if
  if is the first prefetching then
    for block = Pstart To block = Pdepth do
      pipeline block to SSD
    end for
  else if SSD buffer is consumed then
    Pnext = call (Next to Prefetch) for the consumed block
    if the Pnext data block is not already in the SSD then
      pipeline Pnext to the SSD
    end if
  end if
```

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
1	I	O	O	O	O	AX														
2	I	-	-	-	-	A	X													
3						I	-	O	O	O	AX									
4							I	-	-	-	-	AX								
5 ss	II	-	-	-	-	-	-	-	a		I	-	O	O	AXx					
6 ss	II	-	-	-	-	-	-	-	a			I	-	-	-	AXx				
7 ss	II	-	-	-	-	-	-	-	a						I	-	O	O	AXx	
8																I	-	-	-	O
9																			I	-
10																				
11 ss															II	-	-	-	-	-
12 ss																II	-	-	-	-
13 ss																			II	-

I: Initiate prefetch to buffer cache
II: Initiate prefetch to SSD
-: Prefetch in progress
A: Arrived in buffer cache
a: Arrived in SSD
X: Consumed from buffer cache
x: Consumed from SSD
O: Stall
<>: Early Arrival to SSD

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
8	AX																			
9	-	O	O	AX																
10	I	-	-	-	O	AX														
11 ss	-	-	a	I	-	-	O	AXx												
12 ss	-	-	-	a		I	-	-	O	AXx										
13 ss	-	-	-	-	-	-	a	I	-	-	O	AXx								
14										I	-	-	O	O	AX					
15												I	-	-	-	O	AX			
16															I	-	-	O	O	AX
17 ss								II	-	-	-	-	-	-	-	a	I	-	-	-
18 ss										II	-	-	-	-	-	-	-	a		I
19 ss												II	-	-	-	-	-	-	-	a

Figure 5.3: Average stalls when using IPO and a fixed number of buffers for parallel prefetching in the buffer cache. The maximum number (Max_{BW}) of read requests is 5. Informed prefetching buffers = 2, and the rest 3 spaces of the bandwidth are used for pipelined prefetching. $T_{hdd-cache} = 5$, $T_{hdd-ss} = 8$, $T_{ss-cache} = 4$, and $X_{cache} = 2$. The first accesses stall for 5 time units. Before IPO fetches hinted blocks from HDD to SSD, the application stalls for $T_{stall-hdd}(\mathrm{X}_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 2$ time units every 3 accesses. IPO continues to fetch 3 hinted blocks each time from HDD. When a prefetched block is consumed from SSD, a new pipelined prefetching request is initiated by IPO. When IPO is employed, stalls time becomes 40 and elapsed time is 76 time units.

Access Number	Time (One Time Step = $(T_{cpu} + T_{hit} + T_{driver})$)																			
	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
17 ss	AXx																			
18 ss	-	O	O	AXx																
19 ss	I	-	-	-	AXx															
20				I	-	O	O	O	AX											
21					I	-	-	-	-	AX										
22									I	-	O	O	O	AX						
23 ss	II	-	-	-	-	-	-	-	a	I	-	-	-	<>	AXx					
24 ss				II	-	-	-	-	-	-	-	a		I	-	O	O	AXx		
25 ss					II	-	-	-	-	-	-	-	a		I	-	-	-	AXx	
26																		I	-	O
27																			I	-
28																				
29 ss															II	-	-	-	-	-
30 ss																		II	-	-
31 ss																			II	-

Access Number	Time (One Time Step = $(T_{cpu} + T_{hit} + T_{driver})$)																			
	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
26	O	O	AX																	
27	-	-	-	AX																
28			I	-	O	O	O	AX												
29 ss	-	-	a	I	-	-	-	<>	AXx											
30 ss	-	-	-	-	-	a		I	-	O	O	AXx								
31 ss	-	-	-	-	-	-	a		I	-	-	-	AXx							
32												I	-	O	O	O	AX			

Figure 5.4: Continue Figure 5.3

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
1	I	O	O	O	O	AX														
2	I	-	-	-	-	A	X													
3						I	-	O	O	O	AX									
4							I	-	-	-	-	AX								
5											I	-	O	O	O	AX				
6												I	-	-	-	-	AX			
7																I	-	O	O	O
8																	I	-	-	-

I: Initiate prefetch to buffer cache
II: Initiate prefetch to SSD
-: Prefetch in progress
A: Arrived in buffer cache
a: Arrived in SSD
X: Consumed from buffer cache
x: Consumed from SSD
O: Stall
<>: Early Arrival to SSD

Access Number	Time (One Time Step = ($T_{cpu} + T_{hit} + T_{driver}$))																			
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
7	AX																			
8	-	AX																		
9	I	-	O	O	O	AX														
10		I	-	-	-	-	AX													
11						I	-	O	O	O	AX									
12							I	-	-	-	-	AX								
13											I	-	O	O	O	AX				
14												I	-	-	-	-	AX			
15																I	-	O	O	O
16																	I	-	-	-

Figure 5.5: Average stalls when using a fixed number of buffers for parallel prefetching in buffer cache. $T_{hdd-cache} = 5$, and $X_{cache} = 2$. The first accesses stall for 5 time units. All data is read from HDD. The application stalls for $T_{stall-hdd}(X_{cache}) = T_{hdd-cache}$ - $3(T_{cpu} + T_{hit} + T_{driver}) = 2$ time units every 3 accesses. In the non-IPO case, stalls time is 45 and elapsed time is 81.

Access Number	Time (One Time Step = ($Tcpu + Thit + Tdriver$))																			
	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
15	AX																			
16	-	AX																		
17	I	-	O	O	O	AX														
18		I	-	-	-	-	AX													
19						I	-	O	O	O	AX									
20							I	-	-	-	-	AX								
21											I	-	O	O	O	AX				
22												I	-	-	-	-	AX			
23																I	-	O	O	O
24																	I	-	-	-

Access Number	Time (One Time Step = ($Tcpu + Thit + Tdriver$))																					
	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
23	AX																					
24	-	AX																				
25	I	-	O	O	O	AX																
26		I	-	-	-	-	AX															
27						I	-	O	O	O	AX											
28							I	-	-	-	-	AX										
29											I	-	O	O	O	AX						
30												I	-	-	-	-	AX					
31																I	-	O	O	O	AX	
32																	I	-	-	-	-	AX

Figure 5.6: Continue Figure 5.5

end while

The IPO algorithm calls the Pstart algorithm to determine the first hinted block to be fetched from HDD. Next IPO calculates the pipelined prefetching depth (i.e., Pdepth), which is affected by the available I/O bandwidth. Then, IPO initiates a prefetching request to fetch hinted blocks from HDD to SSD. Every time a cached block is consumed from SSD by the TIP module, IPO starts prefetching the next hinted block from HDD to SSD. Bandwidth shortage means TIP is unable to complete its prefetching requests on time from any level. This may be caused by the pipelined prefetching process and limited I/O bandwidth. In the event of bandwidth shortage, IPO reduces its pipelined prefetching depth by one until the shortage is eliminated. The IPO algorithm checks if a data block to be fetched has been modified before the block is requested by TIP. In this case, the prefetched copy will be discarded to preserve consistency. If the data is cached in the SSD. the pipelined prefetching request will be discarded by IPO.

5.5 Performance Evaluation

This section evaluates IPO's performance through extensive simulations. After presenting the simulation environment, we discuss the impact of IPO on application elapsed times.

5.5.1 Simulation Environment

IPO shares the same simulation environment of iPipe in terms of architecture, traces, assumption, and parameters all of which were validated in Chapter 3.

Since both TIP and IPO work concurrently, the maximum number of read requests issued to HDDs is Max_{BW}. According to Equation 4.2, the prefetching horizon of reading data from the HDD to the buffer cache equals 63 in case 10 MB or 200 MB data blocks are being used. In this simulation, we simulate IPO and record its performance when Max_{BW} value is set to 15. Note that this value is 15 in the TIP study [1]. Assuming that each single read request needs one disk to not face any congestions, Max_{BW} in this case equals 15.

The IPO pipelined prefetching depth does not consume the SSD space. If TIP is assigning $(\text{X}_{cahce}) = 1$, pipelined prefetching depth becomes 14. In this case, IPO needs 140 MB in the 10MB-block-size case and 2.8 GB) in the 200-MB-block-size case. The buffers are stripped through the entire disk array of 15 disks.

5.5.2 Elapsed Time Improvement

The simulation results show that when TIP is using very few prefetching buffers (e.g. $X_{cahce} = 1$), IPO reduces the total elapsed time by approximately 56% when using 10 MB data blocks and 34% when using 200 MB blocks. Figure 5.7 shows the total elapsed time when Max_{BW} equals 15 in the event of 10 MB data blocks. Figures 5.8 and 5.9 show the total elapsed time when Max_{BW} equals 15 and 200 MB data blocks are being used. As the number of X_{cahce} increases, the difference between IPO and non-IPO cases decreases. This is because TIP's prefetching process becomes more efficient at hiding the application's

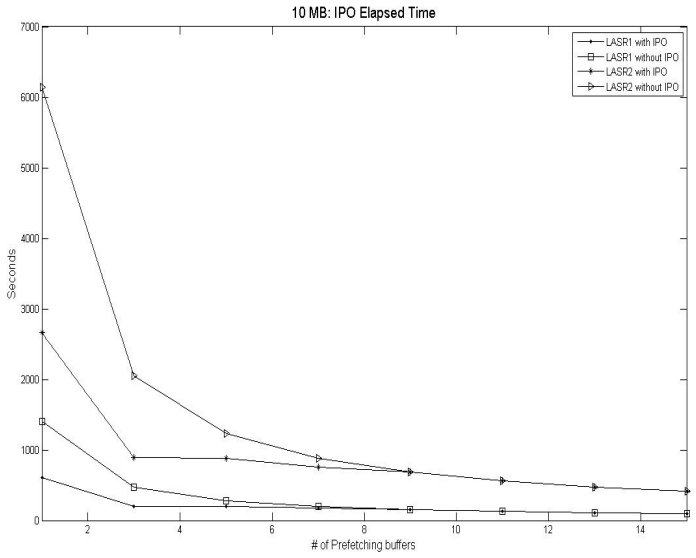

Figure 5.7: IPO reduces application elapsed time. 10 MB block size. Total elapsed time when using 1 to 15 prefetching buffers. $\text{Max}_{BW} = 15$.

stalls. At the same time, IPO's pipelined prefetching process shows the most performance improvement during TIP's most critical situation (i.e., TIP uses very few X_{cahce} buffers - e.g., 1). Also, due to pipelining limitations, IPO does not exhibits a significant performance improvement when X_{cahce} is larger than 9, especially as Max_{BW} becomes smaller; however, the 10-MB-block-size case shows better performance improvement because SSD performs better than HDD when block size is 10MB.

It is clear that the 200-MB-block-size case shows significantly larger elapsed time values, because both SSD and HDD have longer read latencies to access blocks whose size is 200MB. Another reason is that SSD is much faster than HDD when it comes to the case of accessing large (e.g., 200MB) data blocks.

Performance improvement offered by IPO becomes more pronounced when I/O bandwidth of the parallel storage system increases. This is a result IPO being able to issue more pipelined prefetching requests to HDDs and bring more data to the uppermost level.

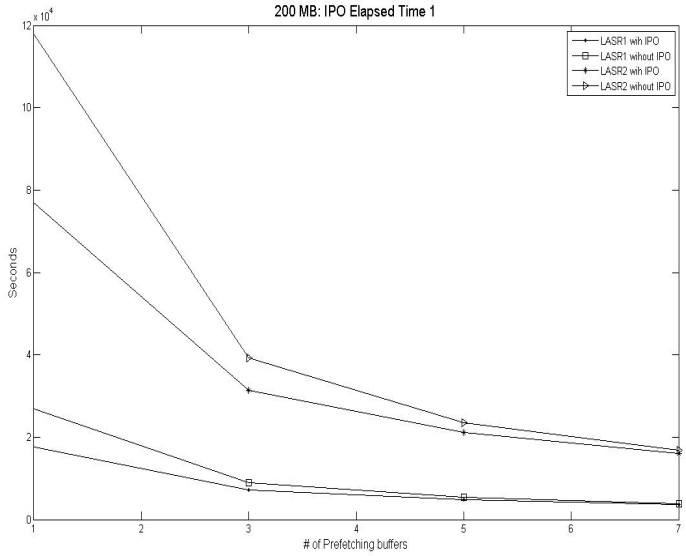

Figure 5.8: IPO reduces application elapsed time. 200 MB block size. Total elapsed time when using 1 to 7 prefetching buffers. $\text{Max}_{BW} = 15$.

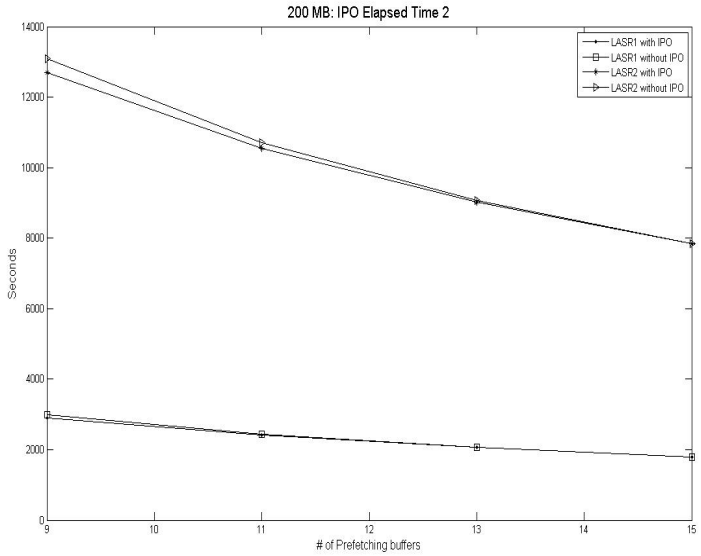

Figure 5.9: IPO reduces application elapsed time. 200 MB block size. Total elapsed time when using 9 to 15 prefetching buffers. $\text{Max}_{BW} = 15$.

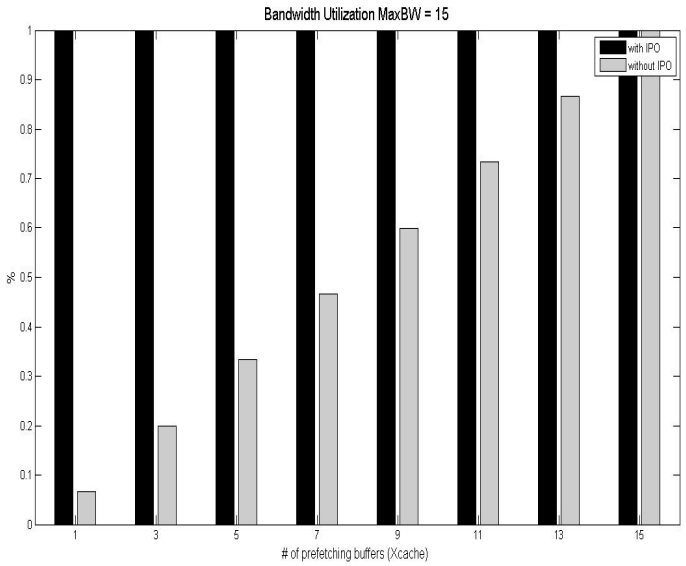

Figure 5.10: Bandwidth utilization when X_{cahce} is varied from 1 to 15 in both IPO and non-IPO cases. IPO fully utilizes the bandwidth.

5.5.3 Bandwidth Utilization

Recall that IPO utilizes unused I/O bandwidth in parallel storage systems. When the TIP module is using few buffers for prefetching, the rest of the I/O bandwidth is used by IPO. Again, in this experiment, the block size is set to 10 MB and 200 MB, respectively. Figure 5.10 shows the bandwidth utilization when using different values of X_{cahce} between 1 and 15 for both IPO and non-IPO cases.

5.5.4 Increasing the Max_{BW} Value

We observed that IPO's performance improvement is not as significant as that of iPipe because of the limited I/O bandwidth. When available I/O bandwidth Max_{BW} increases, IPO tends to perform like iPipe. This is because the storage system's I/O bandwidth becomes high enough. When the block size is 10 MB, IPO can improve the system performance similarly like iPipe when the Max_{BW} equals to 154 (see Equation 4.4).

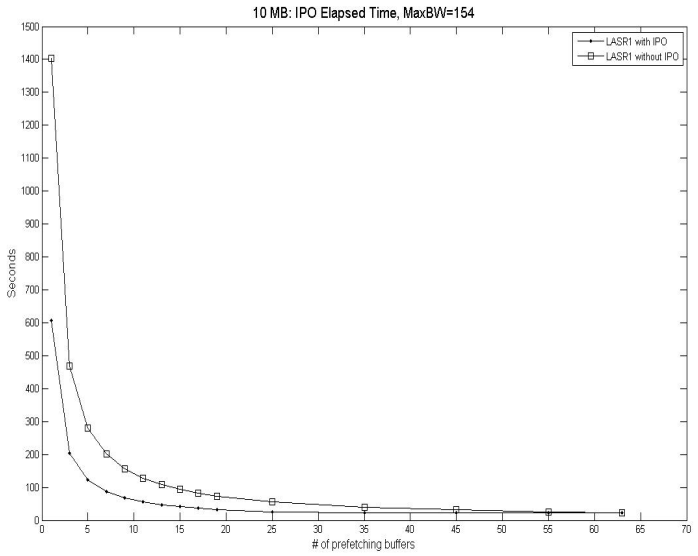

Figure 5.11: IPO reduces the elapsed time. Total elapsed time when the X_{cahce} value is increased from 1 to 63. Max_{BW} is set to 154.

Figure 5.11 shows application elapsed time when the X_{cahce} value is increased from 1 to 63 and Max_{BW} is 154. The LASR1 trace is used in this simulation study. IPO shows similar results to those of iPipe using LASR1 (see also Figures 4.14, 4.15, and 4.16).

5.6 Summary

IPO is an extension of the iPipe mechanism presented in the previous chapter. IPO differs from iPipe in that it can fully support multi-level storage systems with limited I/O bandwidth. IPO decides the number of concurrent prefetching requests issued to HDDs in accordance with available I/O bandwidth offered by both SDDs and HDDs. IPO ensures that the pipelined prefetching scheme does not make SDDs overly loaded. On average, performance improvements provided by IPO are not as significant as those offered by iPipe due to the limited I/O bandwidth. It reduces the application elapsed time by approximately 56% when the X_{cahce} is set to 1. As the X_{cahce} value increases, IPO's performance improvement is decreasing. Experimental results show that increasing the Max_{BW} value helps in making

IPO performance improvement more noticeable. The results reported in this chapter indicate that our proposed pipelined prefetching solutions are very beneficial to parallel storage systems offering sufficient I/O bandwidth.

Chapter 6

IPODS: Informed Prefetching in Distributed Multi-level Storage Systems

6.1 Overview

In the previous two chapters, we proposed informed prefetching algorithms (i.e., iPipe and IPO) that aim to hide I/O latency by invoking I/O parallelisms and prefetching hinted accesses in multi-level storage systems. I/O access hints are provided by applications so that hinted blocks can be prefetched prior to their accesses. In this Chapter, we extend the iPipe and IPO schemes to meet the needs of distributed multi-level storage systems, where massive amounts of data are allocated to a set of storage servers built with multi-level storage devices.

A distributed storage system consists of several storage servers among which data are stripped and stored [72] [83] [68] [82]. Clients can store and retrieve data to and from the storage servers through networks. Distributed storage systems aim at offering high I/O performance, high bandwidth, and scalability [67] [69] [66]. Apart from storage devices in distributed systems, network and server latencies raise major I/O performance issues, which can be addressed by prefetching techniques.

Several architectures of distributed storage systems were proposed in previous studies. For example, Figures 6.1 and 6.2 show two existing architectures designed in prior research projects [65] and [71]. In both architectures, there are several distributed storage nodes and multiple clients. Data blocks are stripped among the storage nodes, all of which are connected by a network.

In this Chapter, we focus on a way of extending the IPO mechanism in the realm of distributed multi-level storage systems. Note that the terms servers, nodes, and sites are used exchangeable throughout this Chapter. In a distributed storage system, each storage

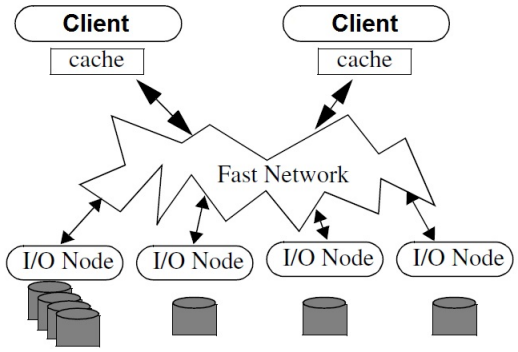

Figure 6.1: The architecture of a distributed parallel storage system. Several distributed clients and storage nodes are connected by a network.
T.Madhyastha; G. Gibson; C. Faloutsos: Informed prefetching of collective input/output requests, Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), Portland, Oregon, 1999.

server consists of multi-level storage devices (see the previous two Chapters for details on multi-level storage systems) and data blocks are stripped among multiple storage servers. We decide to extend the IPO scheme rather than its earlier iPipe version, because IPO does not rely on an unreasonable assumption that storage systems offer unlimited I/O bandwidth. In this part of study, we consider limited number of storage nodes with limited aggregated I/O bandwidth. The goal of this study is to reduce disk read latency, which is a part of the total prefetching time latencies. Since each storage server consists of a multi-levels storage devices, the IPO mechanism in each server enables TIP to locate prefetching requests in the uppermost storage level of the server. The IPODS helps in reducing application stalls and elapsed time in distributed and parallel storage systems.

6.2 Motivations and Objectives

The motivation of the IPODS study is four-fold. First, little attention has been paid to informed prefetching in distributed/parallel multi-level storage systems. Please refer to the related work chapter for a summary of research projects related to distributed and parallel storage systems. Second, distributed and parallel storage systems have high scalability, because massive amounts of data can be striped and distributed across a large number of

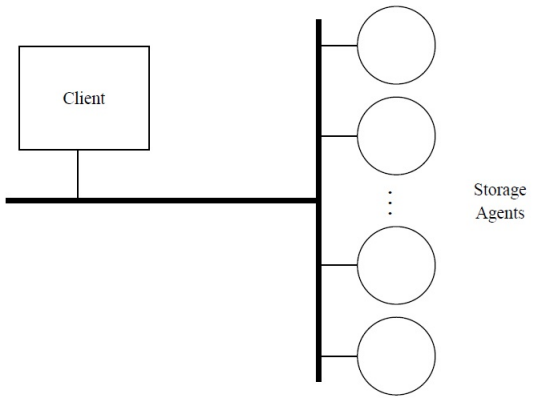

Figure 6.2: [71] Another architecture of a distributed parallel storage system. Several distributed clients and storage nodes are connected by a network.
Luis Cabrera , Darrell D.E. Long: SWIFT: USING DISTRIBUTED DISK STRIPING TO PROVIDE HIGH I/O DATA RATES, University of California at Santa Cruz, Santa Cruz, CA, 1991.

storage nodes and servers. Third, each storage node and server in distributed and parallel system can be built with multi-level storage devices, where hot data blocks are stored and cached in upper-level storage for quick accesses. Prefetching techniques can be employed to preload hinted data blocks to the upper-level storage to shorten disk read latencies, because upper-level storage shows good I/O performance in terms of read latency.

There are the following two research objectives in this part of study:

- To incorporate informed prefetching in distributed/parallel multi-level storage systems.
- To reduce prefetching overhead by leveraging multiple prefetchers working in parallel.
- To evaluate the performance of the proposed IPODS scheme by measuring applications' stalls and elapsed time.

6.3 IPODS Design Issues

When a client accesses data blocks from a multi-level storage server via network connections, the IPODS mechanism brings hinted blocks to the upper-level storage. Although

IPODS is unable to reduce network latency that is out the scope of this dissertation, IPODS is capable of shortening long I/O read latencies.

The first design issue is to enable multiple prefetchers to collaborate in a pipelining manner. Similar to iPipe and IPO, IPODS have several prefetching modules residing in each storage server. The number of prefetchers in a storage server depends on the number of storage levels. An N-level storage server requires N-1 prefetchers. For example, one prefetcher fetches data blocks from tape to hard drives, whereas another prefetcher loads data from hard drives to solid state disks (SSDs). Unlike iPipe and IPO, IPODS should incorporate a prefetcher to bring data from remote storage servers to a client's local I/O buffers.

A second design issue in IPODS is a coordination mechanism between a pair of client local I/O buffer and buffers in SSDs (i.e., upper-level device) of a remote storage server. The prefetching module fetches data from remote storage servers to local buffers contains both client and server portions. The client portion of the prefetching module takes hints from applications at the client. Then, the client portion needs to check if the hinted blocks have been cached in the local buffer. If the hinted blocks are not available in the local buffer, a request will be sent to the server portion to have the hinted blocks fetched from the remote server. The prefetching module handling the client and server portions are working in parallel with the other prefetching modules residing in remote storage servers.

6.3.1 The IPODS Architecture

Before presenting the IPODS implementation details, let us describe the hardware and software architectures of the IPODS prefetching mechanism.

Hardware Architecture of IPODS

Figure 6.3 illustrates the hardware architecture of IPODS. The system consists of client nodes and distributed storage nodes. Each client node maintains a local buffer caches;

applications running on clients nodes retrieve data from remote storage nodes. Applications disclose their future access hints to client nodes, which prefetch data from remote storage nodes to local buffers. Each storage node built with N-level storage devices consists of N-1 prefetching modules. There is one prefetcher between ith and (i+1)th levels. In this dissertation study, we configure each storage node as a two-level storage device - the upper level is a solid state disk and the lower level is a hard drive. A large data block are stripped and stored across multiple storage nodes. Data blocks are transferred Client and I/O storage nodes through networks. Unlike iPipe, IPODS are designed for distributed/parallel storage systems with limited I/O bandwidth.

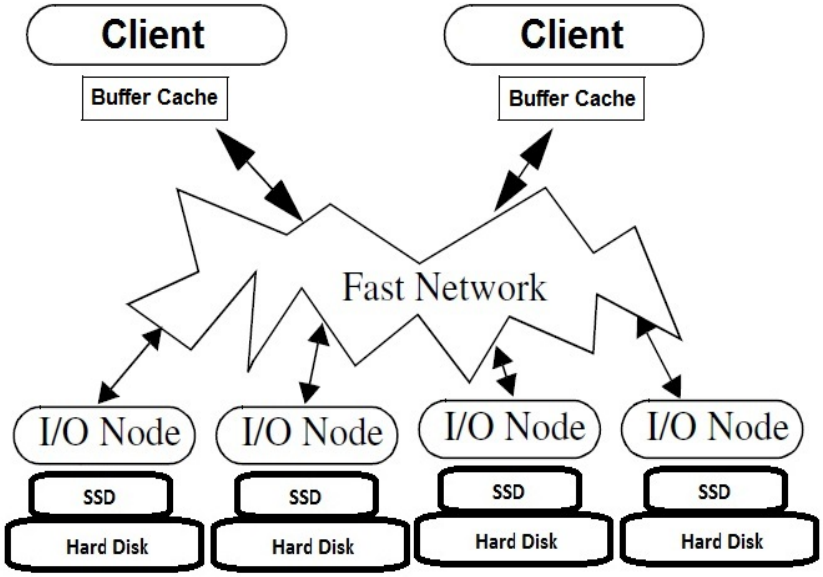

Figure 6.3: Distributed/Parallel Multi-level Storage System: The system shows several distributed storage nodes and clients connected by a network. Each I/O storage node consists of a two-level storage device containing both SSD and HDD.
T.Madhyastha; G. Gibson; C. Faloutsos: Informed prefetching of collective input/output requests, Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), Portland, Oregon, 1999.

Software Architecture of IPODS

Figures 5.1 and 5.2 represent the software architecture of IPO. IPODS and IPO share a similar software architecture. IPODS relies on the TIP module [1] to process hints disclosed

by applications. IPODS is in charge of controlling buffers in all the storage levels while performing informed prefetching process. When IPODS receives a prefetching request, it checks local buffer first before searching SSDs and HDDs in remote storage nodes. IPODS incorporates IPO to control the distributed storage nodes and to perform pipelined prefetching process based on the application hints.

6.3.2 Assumptions

We assume that data blocks are initially placed in HDDs in remote storage nodes. Like IPO, IPODS reserves a small portion in the SSD for the pipelined prefetching process. This portion - defined as prefetching depth - is represented by Equation 5.1.

The maximum I/O bandwidth represents that maximum number of concurrent read requests that may take place in a distributed system without any I/O congestions or delays. We assume that the maximum I/O bandwidth scales up with the number of the nodes [65]. In addition to disk I/O bandwidth, network bandwidth affect the performance of distributed/parallel storage systems. In this dissertation study, we only focus on the disk I/O issues and the network I/O problems are out scope of this research.

6.4 The IPODS Algorithm

6.4.1 Definitions

IPODS incorporates IPO to deal with the informed prefetching process in each remote storage nodes. When an application starts its execution on a client node, the TIP module [1] assigns X_{cache} number of buffers for prefetching based on the cost-benefit model. In other words, TIP keeps requesting (X_{cache}) concurrent prefetching requests from the remote storage nodes. Concurrently fetching a number of blocks utilizes either a part or all of the distributed/parallel storage system's I/O bandwidth. Recall that IPO assigns its pipelined prefetching depth based on the non-utilized portion of the disk bandwidth. Max_{BW} is the maximum number of data blocks that may be requested concurrently in the

distributed/parallel storage system. The value of Max_{BW} depends on the number of nodes and the available aggregated I/O bandwidth. The IPO's pipelined prefetching depth is determined by Equation 5.1. The pipelined prefetching depth represents the maximum space needed to be reserved in the SSD to cache hinted data blocks fetched from HDDs.

Since a hinted block must be fetched from a remote storage node to a local buffer, the time spent fetching the block includes network transfer time and server latency as well as the disk I/O access time. Equation 6.1 calculates the time spent fetching a hinted block from a remote storage node.

$$Remote_{Read} = T_{Disk} + T_{Network} + T_{Server} \tag{6.1}$$

where;

Remote_{Read} [1]: Time spent in fetching a hinted block from a remote node.

T_{Disk}: Disk I/O access time.

$\text{T}_{Network}$: Network transfer time.

T_{Server}: Server processing time.

Recall that (T_{cpu} + T_{hit} + T_{driver}) represents the time needed for an application to consume a single block in the buffer cache. In addition, a distributed multi-level storage system consists of several storage nodes where each one has an HDD and an SSD. $\text{T}_{hdd-network-cache}$ is the time spent in retrieving a single data block over a network from a remote storage node's HDD. $\text{T}_{ss-network-cache}$ is the time needed to fetch a single data block over a network from a remote node's SSD. T_{hdd-ss} is the disk read latency from the HDD to SSD of a remote node. $\text{T}_{ss-network-cache}$ is less than $\text{T}_{hdd-network-cache}$, because the performance of SSD is higher than that of HDD. If the network bandwidth is very low, the performance

[1] R. Patterson, Hugo, G. Gibson, D. Stodolsky, and J. Zelenka: Informed prefetching and caching, *In Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79-95, CO, USA, 1995.

gap between $T_{ss-network-cache}$ and $T_{hdd-network-cache}$ is diminished. This is because the network rather than hard drives becomes a serious performance bottleneck. In our research, we assume that the overhead introduced by servers can be ignored (i.e., $T_{Server} = 0$), because storage servers simply fetch data rather than processing data.

The IPO pipelined prefetching process enables TIP to find more hinted blocks in SSDs. IPO consists of two algorithms that determine the first appropriate hinted blocks to be fetched and the total number of prefetched blocks. The pipelined prefetching process reduces time spent in fetching data from remote storage nodes. IPODS extends IPO to meet the needs of distributed/parallel multi-level storage systems and reduces elapsed times of applications running on client nodes.

6.4.2 The IPODS Algorithm

IPODS employs the IPO scheme to perform pipelined prefetching in distributed multi-level storage systems. We implement the Pstart and Pnext algorithms in IPODS, because these two algorithms are the core of the IPO scheme. IPODS is different from IPO in the sense that IPODS has to manage local buffers in client nodes. In addition, IPODS must consider network transfer time as well as storage node latencies as parts of the prefetching time.

In a single storage node system, moving a data block from HDD or SSD to the buffer cache is faster than moving the same data block from HDD to SSD (T_{hdd-ss}). In a distributed system, however, T_{hdd-ss} is expected to be less than ($T_{hdd-network-cache}$) and ($T_{ss-network-cache}$). This is because both SSD and HDD are in the same node, moving data blocks from HDD to SSD within a node is faster than moving data from a remote storage node to a client through networks. This results in more application and TIP stalls which consequently opens more room than in a single node system for IPODS to prefetch data blocks in a pipeline manner. This is due to the fact here, IPODS can guarantee the prefetched data will arrive in SSDs and local buffer before being requested by TIP.

6.5 Performance Evaluation

In what follows, we evaluate the performance of IPODS through simulation studies. Let us first illustrates the simulation environment.

6.5.1 Simulation Environment

The simulation environment for IPODS is the similar to that for IPO, because IPODS is an extension of IPO to feed the needs of a distributed/parallel storage systems containing several storage nodes.

We built a simulator using C++ programming language to simulate a distributed storage system in which IPODS is implemented. Again, we use the elapsed time as a key performance evaluation metric. For comparison purpose, we also simulator the same distributed system where IPODS is not employed. We evaluate the impact of buffer size on system performance.

The simulated distributed system consists of a client node offering local buffers and a set of remote storage nodes built with two-level storage devices (i.e., a SSD level and a HDD level). The client and all the storage nodes are connected by a network. Figure 6.3 shows the architecture of the simulated distributed storage system. Data is initially stored in the HDDs whereas prefetched blocks are stored in buffers in SSDs in storage nodes and local buffers in the client. The TIP module manages the buffer cache and IPODS controls buffers in all the storage nodes.

Again, we use the LASR traces (i.e., machine01 or LASR1; machine06 or LASR2) that consist respectively of 11686 and 51206 I/O read requests [97] [98]. According to our validation results presented in Chapter 3, data block size must be larger than 200 MB to ensure that SSDs offer better I/O performance than HDDs. For data blocks that are smaller than 200 MB, there is no significant difference between reading data from a remote SSD and reading data from a remote HDD, since the network becomes a performance bottleneck rather than HDDs.

In the simulated distributed storage system, we configure the system parameters using the data validated in Chapter 3. For LASR1 and LASR2, we assume that each exclusive I/O read system call represents a single request for a 200 MB data block. The trace's I/O reading requests sequence represents the future access hints. In addition, each cache buffer in both client and storage servers are large enough to accommodate a large data block.

According to our validation chapter 3, the system parameters used in our simulation study are: block size equals 200 MB, and (T_{cpu} + T_{hit} + T_{driver}) equals 0.02 seconds. Data is located in the HDDs. The average latency of reading a 200MB data block from a remote SSD and HDD to the buffer cache over a LAN network is 4.158 and 4.43 seconds, respectively. The time spent reading a 200MB block from the HDD to the SSD T_{hdd-ss} equals 4.5 seconds. There is no write latency to the buffer cache; similar assumption can be found in [1].

Since both TIP and IPODS work concurrently, the maximum number of read requests to HDDs is set to Max_{BW}. Based on Equation 4.2, the prefetching horizon of reading data from HDDs to buffer cache is 222 data blocks. Similar to the IPO simulation study, we simulate IPODS by setting Max_{BW} to 15 concurrent read requests. We choose 15 because in the TIP study, 15 disks were tested in the performance evaluation [1]. In the IPODS simulator, each storage node's disks can individually handle a single read request without I/O congestion.

6.5.2 Improving Elapsed Time

The simulation results show that IPODS reduces the total elapsed time by about 6% when the TIP module is using very few buffers for prefetching (e.g., X_{cahce} is set to 1, 2, and 3). Figure 6.4 shows the total elapsed time for varying numbers of prefetching buffers when Max_{BW} equals 15. As the number of X_{cahce} increases, the performance difference between the IPODS and non-IPODS cases decreases because TIP becomes more efficient in reducing the application's stalls.

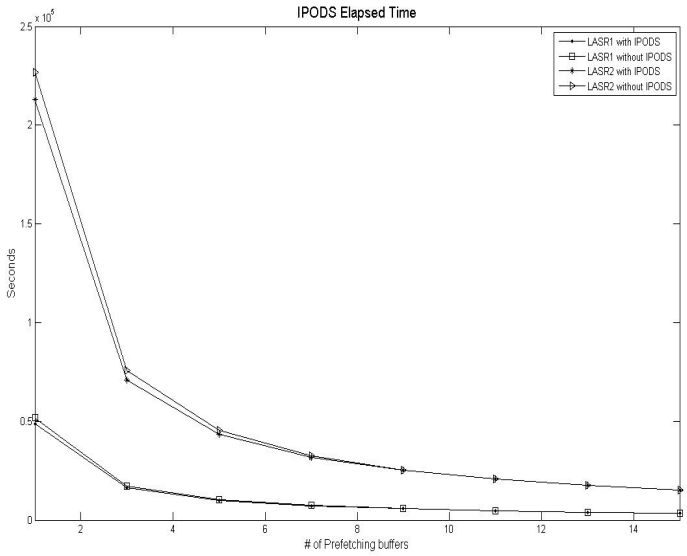

Figure 6.4: Total elapsed time when the number of prefetching buffers is varied from 1 to 15. $\text{Max}_{BW} = 15$. IPODS reduces the elapsed time.

Importantly, the IPODS pipelined prefetching process shows its best performance improvement when TIP uses few X_{cahce} buffers (e.g. $X_{cahce} = 1$). The experimental results suggest that IPODS can enhance I/O performance of distributed and parallel storage systems where the TIP module has limited or no cache buffer to utilize. On the other hand, when X_{cahce} is greater than or equal to 9 especially when (Max_{BW}) is limited, IPODS can not exhibit any performance improvement.

6.6 Summary

In this chapter, we describe the IPODS prefetching scheme - an extension of an earlier IPO version. IPODS are designed to meet the needs of distributed/parallel multi-levels storage systems, where networks introduce significant overhead to the entire prefetching process. Like iPipe and IPO, IPODS reduces I/O access time by hiding the long I/O time through prefetching. Our results show that IPODS can judiciously reduce the elapsed times of applications by approximately 6% when the cache buffer is small (e.g., $X_{cahce} = 1$). When X_{cahce} increases, the improvement offered by IPODS start diminishing.

Chapter 7

Prototype Development

In this chapter, we describe our prototypes for the informed prefetching schemes proposed in the previous three chapters. Because we are unable to test our design in a large scale distributed system, we develop our prototypes using a computing cluster in the storage systems research laboratory at Auburn University. The laboratory is located in Room 2104 of the Shelby Building. We replay real-world traces to evaluation the performance of the prototypes. Then, we compare experimental results obtained from the prototypes with those obtained from the simulators to validate the correctness of our simulators.

We only replay a portion (i.e., 10%)of the two traces used to drive our simulators, because it takes a couple of days to replay an entire trace. The results generated by our prototypes can be used to evaluate the performance of our proposed pipelined prefetching schemes in a real-world storage system.

7.0.1 Objectives

Developing the prototypes helps us to achieve the following three objectives:

- To build real-world storage systems that implements our pipelined prefetching solutions.
- To evaluate performance of our prefetching schemes in a real-world storage system.
- To validate the correctness of our simulators by comparing simulation results with the prototyping results.

7.0.2 System Setup

We summarize the testbed (including hardware and software) used to develop our prototypes as follows.

- Memory: Samsung 3GB RAM Main Memory.
- HDD: Westren Digital 500GB SATA 16 MB Cache WD5000AAKS.
- SSD: Intel 2Gb/s SATA SSD 80G sV 1A.
- LAN Network Switch: [96] Network Dell Power Connect 2824.
- Scripting Language: c-shell, c++.
- Input Parameter: set of informed prefetching requests.
- Performance Metric: Elapsed time measured in Second.

7.0.3 Design Issues of the Prototypes

In the process of developing our prototypes, we need to determine the first prefetched blocks and the total number of the blocks to be prefetched in a pipeline manner. T_{hdd-ss} - time spent in moving a block from hard drive to solid state disk - is an important system parameter used to determine the first block and the number of blocks to be prefetched. To guarantee the correctness of T_{hdd-ss}, we validate this parameter in a real-world testbed in our laboratory by recording the highest value of T_{hdd-ss} (see Chapter 3 for validation details). Because the prototypes are developed in a small scale storage system using our testbed, we set this parameter as a constant. Specifically, T_{hdd-ss} is set to 0.122 seconds and 4.5 seconds when block size is 10 MB and 200 MB, respectively.

The iPipe scheme differs from IPO in that iPipe takes $T_{stall-hdd(Xcache)}$ to determine the first block to be prefetched and $T_{stall-ss(Xcache)}$ to determine the prefetch depth (i.e., the total number of blocks to be prefetched). On the other hand, IPO always takes $T_{stall-ss(Xcache)}$ to

determine the first block and the total number of blocks to be fetched in a pipeline manner. In our simulation studies, these values are constants. However, we are unable to predict these values in the prototypes. Therefore, we use the best case (i.e., lowest values) of $T_{hdd-cache}$, which is 0.07 seconds in the case of 10 MB data block size and 1.6 seconds in the case of 200 MB block size as shown in Figures 3.5 and 3.8. We also set $T_{ss-cache}$ to 0.044 seconds in the case of a 10 MB data block size and 0.96 second in the case of a 200 MB block size as shown in Figures 3.6 and 3.9. These values are used to determine the minimal stall values that help our algorithms decide the proper prefetching depth and avoid incorrect results.

For the TIP module in the prototypes, we set $\mathrm{T}_{cpu}) + \mathrm{T}_{hit} + \mathrm{T}_{driver}$ to 0.00192 seconds when block size is 10 MB and to 0.037 seconds when block size is 200 MB. These settings are validated in Chapter 3. When it comes to T_{cpu}, we consider the worst case where our prototype does not perform any computational overheads, meaning that applications are very I/O intensive and consuming prefetched data very quickly. Note that this assumption is conservative in the sense that our pipelined prefetching schemes can achieve better I/O performance when T_{cpu} is a large value and applications are CPU intensive.

7.0.4 Validation Process

In order to validate our simulations results, we have to check if there is a similar trend and little variation between the simulation results and prototyping results. Recall that our prototypes implement the pipelined prefetching schemes proposed in the previous three Chapters. The prototypes also reply 1000 I/O read requests from the two real-world I/O traces. Although we only reply 1000 requests in the prototypes, the prototypes can handle hundreds of thousands of requests. We simply reply 1000 requests from the two traces, because it takes a couple of days to fully replay the entire trace. We conduct one experiment to show that performance trend of replaying 1000 requests is the same as that of replaying the entire I/O traces; the disk access time for pipelined prefetching and non-pipelined prefetching are bounded by a particular range. We set $T_{cpu} + \mathrm{T}_{hit} + \mathrm{T}_{driver}$ in the prototypes to a

constant, which represents the worst case (i.e., smallest value). This enables us to notice the system performance trend (with and without our solutions) and to expect the results in terms of elapsed time when am I/O trace is replayed. Equation 7.1 calculates the expected elapsed time for a real world application, whose I/O access pattern is represented as a trace replayed in our prototypes.

$$Prot_{Elapsed} = \left(\frac{Trace\#ofReads}{1000}\right) \times PrototypingElapsedTime \quad (7.1)$$

where;

$Prot_{Elapsed}$: Expected prototyping elapsed time of a given trace

7.1 Prototypes

In this section, we discuss our prototypes developed for the iPipe, IPO, and IPODS prefetching schemes. We also show results obtained from the prototypes and compare prototyping results with earlier simulation results to validate the correctness of our simulators.

7.1.1 The iPipe Prototype

In this subsection, we present a prototype for the iPipe pipelined prefetching solution. When using iPipe, the first few hinted blocks are fetched from the HDDs (i.e., the lowest storage level) and the rest hinted blocks are fetched from the SSDs (i.e., the uppermost storage level) if the hinted blocks are cached in the SSDs. We measure the prototyping results in terms of elapsed time with and without iPipe that handles 1000 read requests. In the iPipe simulator, the TIP module uses the number of buffers anywhere from 1 to 63 and the block size is either 10 MB or 200 MB. Then, we collect prototyping results by replaying a portion of the LASR1 or LASR2 traces. Finally, we compare the variation ratio between the iPipe simulation results and the iPipe prototyping results.

Tables 7.1 and 7.5 show the prototyping results in terms of elapsed time in seconds for the 10 MB and 200 MB cases, respectively. The prototyping results show that iPipe improves the system performance by approximately 51% when the block size is 10 MB and 25% when the block size is 200 MB. The performance improvement offered by iPipe is more pronounced when the TIP module is using few prefetching buffers. As the number of prefetching buffers increases, the performance difference between iPipe and the non-iPipe counterpart is decreasing.

When the block size is 10 MB, the prototyping results are close to the simulation results. For example, the difference between the simulation results and the prototyping results is only about 9%. The iPipe simulator is more accurate when the block size is large (e.e., 200 MB). For example, the difference between the simulation and prototyping results is as low as 5%. The simulator is more accurate for large block size, because the simulator always uses the worst case of input parameters.

In the iPipe prototype, we implemented an application whose access pattern is the same as the LASR traces machine01 (LASR1) and machine06 (LASR2), which consist of 11686 and 51206 I/O reading requests, respectively. Tables 7.2 and 7.6 show the prototyping results in term of elapsed time measured in seconds when the application is running in the iPipe prototype. Both the iPipe simulator and prototype share the same performance trend in term of improvement over the non-iPipe counterpart.

Tables 7.3 and 7.7 show the iPipe simulation results illustrated in the iPipe chapter. Tables 7.4 and 7.8 show a comparison between the improvement ratio of the simulation results and the prototyping results. When we compare each case's simulation and prototyping results for different numbers of prefetching buffers, we notice that they are close to each other. The simulation results show higher values by about 3-18%, because the iPipe simulator uses the highest (worst case) recorded disk read latency. In the 10 MB case, prototyping results show higher values than the simulation results by about 3-18% when X_{cache} is larger than or equal to 11. We observe that during the prototyping experiments, several read requests

experience higher disk I/O time than those requests in the simulator. We refer these read requests with high I/O access time as extreme disk requests. This variation ratio is very low, because we use the highest validated disk read latency in our iPipe simulator.

To clarify the effect of these extreme disk requests recorded during our prototyping experiments, let us consider the following motivational example. Suppose that T_{cpu} + T_{hit} + T_{driver} equals 1 time unit, $T_{hdd-cache}$ in simulation equals 5 time units, and $T_{hdd-cache}$ in prototyping does not exceeds 4 time units but sometimes records some extreme values equal to 6 time units. According to Equation 4.2, the prefetching horizon in the iPipe simulator equals 5. We compare both simulation and prototyping stalls when X_{cache} equals 1 and 4, which are considered small and large values, respectively. In addition, we assume that the application issues 8 I/O read requests in both simulator and prototyping. In the simulation case, the requests' read latencies is (5,5,5,5,5,5,5,5) while in the prototyping case we assume that the accesses are read in disk read latencies of (6,4,4,4,6,4,4,4). When X_{cache} is set to 1, the application stalls for 5,4, and 3 time units when $T_{hdd-cache}$ equals 6,5, and 4, respectively (see Equation 4.1). When X_{cache} is 4, the application stalls for 2,1, and 0 time units when $T_{hdd-cache}$ equals 6,5, and 4, respectively. When applying these stall values to the given stream of I/O read requests, we observe the following facts:

- In the simulation case, when X_{cache} is set to 1, the application stalls for 32 time units.
- In the prototype case, when X_{cache} is set to 1, the application stalls for 28 time units.
- In the simulation case, when X_{cache} is set to 4, the application stalls for 2 time units.
- In the prototype case, when X_{cache} is set to 4, the application stalls for 4 time units.

The above motivational example indicates that the iPipe prototype produces more stalls than its simulator when the value of X_{cache} increases.

Table 7.1: Total elapsed time measured in seconds when the iPipe prototype is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
With iPipe	47.88	16.2851	10.0297	7.40379	5.81344
Without iPipe	99.5	34.9642	22.1658	15.9048	12.9741
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
With iPipe	4.90562	4.24443	3.79232	3.39027	3.08462
Without iPipe	11.338	10.0755	9.79321	7.99748	7.35371
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
With iPipe	2.54144	2.44104	2.34504	2.13651	2.07306
Without iPipe	6.79856	5.4483	5.05102	3.34087	3.16796

Table 7.2: Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe prototype. The number of prefetching buffers is set to a range between 1 to 63. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
LASR1 With iPipe	559.6273482	190.3076786	117.2070742	86.52068994	67.93585984
LASR1 Without iPipe	1161.996241	408.5916412	259.0295388	185.8634928	151.6153326
LASR2 With iPipe	2452.188772	833.8948306	513.5808182	379.1184707	297.6830086
LASR2 Without iPipe	5091.663489	1790.376825	1135.021955	814.4211888	664.3517646
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
LASR1 With iPipe	57.32707532	49.60040898	44.31705152	39.61869522	36.04686932
LASR1 Without iPipe	132.495868	117.742293	114.4434521	93.45855128	85.93545506
LASR2 With iPipe	251.1971777	217.3402826	194.1895379	173.6021656	157.9510517
LASR2 Without iPipe	580.573628	515.926053	501.4711113	409.5189609	376.5540743
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
LASR1 With iPipe	29.69926784	28.52599344	27.40413744	24.96725586	24.22577916
LASR1 Without iPipe	79.44797216	63.6688338	59.02621972	39.04140682	37.02078056
LASR2 With iPipe	130.1369766	124.9958942	120.0801182	109.4021311	106.1531104
LASR2 Without iPipe	348.1270634	278.9856498	258.6425301	171.0725892	162.2185598

Table 7.3: Total elapsed time measured in seconds when the iPipe simulator is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
LASR1 With iPipe	607.612	202.537	121.522	86.8018	67.5125
LASR1 Without iPipe	1402.32	467.44	280.464	200.331	155.813
LASR2 With iPipe	2662.65	887.551	532.53	380.379	295.85
LASR2 Without iPipe	6144.72	2048.24	1228.94	877.817	682.747
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
LASR1 With iPipe	55.2375	46.7394	40.5075	35.7419	31.9796
LASR1 Without iPipe	127.484	107.871	93.488	82.4894	73.8063
LASR2 With iPipe	242.059	204.819	177.51	156.627	140.14
LASR2 Without iPipe	558.611	472.671	409.648	361.454	323.406
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
LASR1 With iPipe	24.3045	22.4358	22.4365	22.4369	22.4371
LASR1 Without iPipe	56.0928	40.0663	31.1627	25.4967	22.4371
LASR2 With iPipe	106.506	98.3142	98.3149	98.3153	98.3155
LASR2 Without iPipe	245.789	175.563	136.549	111.722	98.3155

Table 7.4: Comparison between iPipe's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
LASR1 Simulation	0.566709453	0.566710166	0.566710879	0.566708098	0.566708169
LASR1 prototyping	0.51839143	0.534235018	0.54751464	0.53449336	0.551919594
LASR2 Simulation	0.566676757	0.566676268	0.566675346	0.566676198	0.566676968
LASR2 prototyping	0.51839143	0.534235018	0.54751464	0.53449336	0.551919594
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
LASR1 Simulation	0.566710332	0.566710237	0.566709096	0.566709177	0.566709075
LASR1 prototyping	0.567329335	0.578737532	0.61276027	0.576082716	0.580535539
LASR2 Simulation	0.56667699	0.566677456	0.566676757	0.566675151	0.566674706
LASR2 prototyping	0.567329335	0.578737532	0.61276027	0.576082716	0.580535539
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
LASR1 Simulation	0.566709096	0.440033145	0.280020666	0.120007687	0
LASR1 prototyping	0.626179662	0.551962998	0.535729417	0.360492925	0.345616738
LASR2 Simulation	0.566677109	0.440006152	0.280002783	0.120000537	0
LASR2 prototyping	0.626179662	0.551962998	0.535729417	0.360492925	0.345616738

Table 7.5: Total elapsed time measured in seconds when the iPipe prototype is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
With iPipe	1360.27	455.467	285.398	200.495	158.36
Without iPipe	1832.43	653.29	371.157	273.445	246.693
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
With iPipe	130.818	112.293	97.3989	86.2939	80.076
Without iPipe	174.627	148.697	133.107	127.165	108.789
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
With iPipe	60.3609	44.974	38.793	37.6105	37
Without iPipe	86.2886	63.5934	49.8966	40.6426	37.1776

Table 7.6: Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe prototype. The number of prefetching buffers is set to a range between 1 to 63. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
LASR1 With iPipe	15896.11522	5322.587362	3335.161028	2342.98457	1850.59496
LASR1 Without iPipe	21413.77698	7634.34694	4337.340702	3195.47827	2882.854398
LASR2 With iPipe	69653.98562	23322.6432	14614.08999	10266.54697	8108.98216
LASR2 Without iPipe	93831.41058	33452.36774	19005.46534	14002.02467	12632.16176
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
LASR1 With iPipe	1528.739148	1312.255998	1138.203545	1008.430515	935.768136
LASR1 Without iPipe	2040.691122	1737.673142	1555.488402	1486.05019	1271.308254
LASR2 With iPipe	6698.666508	5750.075358	4987.408073	4418.765443	4100.371656
LASR2 Without iPipe	8941.950162	7614.178582	6815.877042	6511.61099	5570.649534
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
LASR1 With iPipe	705.3774774	525.566164	453.334998	439.516303	432.382
LASR1 Without iPipe	1008.36858	743.1524724	583.0916676	474.9494236	434.4574336
LASR2 With iPipe	3090.840245	2302.938644	1986.434358	1925.883263	1894.622
LASR2 Without iPipe	4418.494052	3256.36364	2555.0053	2081.144976	1903.716186

Table 7.7: Total elapsed time measured in seconds when the iPipe simulator is tested. The number of prefetching buffers is set to a range between 1 to 63. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
LASR1 With iPipe	17531.8	5843.93	3506.36	2504.54	1947.98
LASR1 Without iPipe	26877.8	8959.27	5375.56	3839.69	2986.42
LASR2 With iPipe	76811.8	25603.9	15362.4	10973.1	8534.64
LASR2 Without iPipe	117774	39257.9	23554.8	16824.8	13086
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
LASR1 With iPipe	1593.8	1348.65	1168.79	1031.28	922.726
LASR1 Without iPipe	2443.44	2247.31	1791.85	1581.05	1414.62
LASR2 With iPipe	6982.89	5908.6	5120.79	4518.34	4042.73
LASR2 Without iPipe	10706.7	9059.52	7851.59	6927.87	6198.62
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
LASR1 With iPipe	701.272	500.909	432.431	432.399	432.382
LASR1 Without iPipe	1075.11	767.937	597.284	488.687	432.382
LASR2 With iPipe	3072.47	2194.62	1894.67	1894.64	1894.62
LASR2 Without iPipe	4710.95	3364.97	2617.2	2141.34	1894.62

Table 7.8: Comparison between iPipe's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**	**9**
LASR1 Simulation	0.347721912	0.347722527	0.347721912	0.347723384	0.347720682
LASR1 prototyping	0.257668779	0.30281039	0.231058555	0.266781254	0.358068531
LASR2 Simulation	0.34780342	0.347802608	0.347801722	0.347802054	0.34780376
LASR2 prototyping	0.257668779	0.30281039	0.231058555	0.266781254	0.358068531
$\mathbf{X}_{cache}$	**11**	**13**	**15**	**17**	**19**
LASR1 Simulation	0.347722882	0.347697728	0.347718838	0.347724613	0.34772165
LASR1 prototyping	0.250871858	0.244820003	0.268266132	0.321402115	0.263932934
LASR2 Simulation	0.347801844	0.347802091	0.347802165	0.347802427	0.347801607
LASR2 prototyping	0.250871858	0.244820003	0.268266132	0.321402115	0.263932934
$\mathbf{X}_{cache}$	**25**	**35**	**45**	**55**	**63**
LASR1 Simulation	0.347720698	0.347721232	0.27600438	0.11518211	0
LASR1 prototyping	0.30047654	0.292788245	0.222532197	0.074603987	0.00477707
LASR2 Simulation	0.34780246	0.347803992	0.276069846	0.115208234	0
LASR2 prototyping	0.30047654	0.292788245	0.222532197	0.074603987	0.00477707

7.1.2 The IPO Prototype

In this subsection, we describe our performance evaluation using an IPO prototype. Like the iPipe prototype, the IPO prototype simply replays 1000 read requests from the two LASR traces. We measure the application's elapsed time using the IPO prototype. Then, we compare the results obtained from the IPO prototype with the simulation results.

Tables 7.9 and 7.13 show the lapsed time measured in the IPO prototype when block size is set to 10 MB and 200 MB, respectively. The prototyping results confirm that the IPO scheme improves the performance of multi-level storage systems by 51% when block size is 10 MB and by 25% when the block size is 200 MB. We also observe that when the number of prefetching buffers increases, the performance gap between IPO and its non-IPO counterpart is diminishing. Given the same multi-level storage system, applications issuing small requests (e.g., 10MB) gains more benefits from IPO than applications issuing large requests (e.g., 200MB). Small blocks enjoy more benefits from IPO than large blocks, because SSDs show better performance than HDDs in this case.

We implement an application in the IPO prototype to resemble real-world applications whose access patterns are the same as the LASR traces. Tables 7.10 and 7.14 show elapsed time measured from the IPO prototype when the application is tested. The results confirm that both the IPO simulator and its prototype share the same performance trend in term of performance improvement over the non-IPO case.

Table 7.11 reveals the IPO simulation results when the block size is 10 MB; Table 7.15 shows the results when the block size is as large as 200 MB.

Tables 7.12 and 7.16 illustrate the comparison between the simulation results and the prototyping results. We observe that both the simulation and prototyping results share very similar elapsed times. Like the iPipe prototype, the IPO prototype exhibits smaller elapsed time than the IPO simulator when block size is small (e.g., 10MB) and X_{cache} is less than 11. The opposite is true when X_{cache} is larger than or equal to 11. The variation between the simulation and the prototyping results is as small as 3% in all the IPO cases.

Table 7.9: Total elapsed time measured in seconds when the IPO prototype is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
With IPO	47.9343	20.2705	17.9612	14.5456
Without IPO	99.4349	34.9642	22.1658	15.9048
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
With IPO	12.5878	11.2401	9.99	9.79321
Without IPO	12.9741	11.338	10.0755	9.79321

Table 7.10: Total elapsed time measured in seconds when the LASR traces are replayed by the IPO prototype. The number of prefetching buffers is set to a range between 1 to 15. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 With IPO	560.1602298	236.881063	209.8945832	169.9798816
LASR1 Without IPO	1161.996241	408.5916412	259.0295388	185.8634928
LASR2 With IPO	2454.523766	1037.971223	919.7212072	744.8219936
LASR2 Without IPO	5091.663489	1790.376825	1135.021955	814.4211888
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 With IPO	147.1010308	131.3518086	116.74314	114.4434521
LASR1 Without IPO	151.6153326	132.495868	117.742293	114.4434521
LASR2 With IPO	644.5708868	575.5605606	511.54794	501.4711113
LASR2 Without IPO	664.3517646	580.573628	515.926053	501.4711113

Table 7.11: Total elapsed time measured in seconds when the IPO simulator is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 With IPO	607.876	202.792	201.128	172.036
LASR1 Without IPO	1402.32	467.516	280.552	200.392
LASR2 With IPO	2662.92	887.79	880.872	753.498
LASR2 Without IPO	6144.72	2048.28	1229.03	877.908
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 With IPO	155.768	127.448	107.878	93.5731
LASR1 Without IPO	155.87	127.547	107.878	93.5731
LASR2 With IPO	682.69	558.602	472.678	409.672
LASR2 Without IPO	682.792	558.701	472.678	409.672

Table 7.12: Comparison between IPO's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 10 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 Simulation	0.566521193	0.566235166	0.283099033	0.141502655
LASR1 prototyping	0.517932838	0.420249856	0.189688619	0.085458478
LASR2 Simulation	0.566632816	0.566568047	0.283278683	0.141711888
LASR2 prototyping	0.517932838	0.420249856	0.189688619	0.085458478
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 Simulation	0.000654391	0.000776184	0	0
LASR1 prototyping	0.029774705	0.00863468	0.008485931	0
LASR2 Simulation	0.000149387	0.000177197	0	0
LASR2 prototyping	0.029774705	0.00863468	0.008485931	0

Table 7.13: Total elapsed time measured in seconds when the IPO prototype is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
With IPO	1361.64	548.053	355.994	266.736
Without IPO	1832.43	653.29	371.157	273.445
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
With IPO	239.318	174.533	148.564	133.107
Without IPO	246.693	174.627	148.697	133.107

Table 7.14: Total elapsed time measured in seconds when the LASR traces are replayed by the IPO prototype. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 With IPO	15912.12504	6404.547358	4160.145884	3117.076896
LASR1 Without IPO	21413.77698	7634.34694	4337.340702	3195.47827
LASR2 With IPO	69724.13784	28063.60192	18229.02876	13658.48362
LASR2 Without IPO	93831.41058	33452.36774	19005.46534	14002.02467
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 With IPO	2796.670148	2039.592638	1736.118904	1555.488402
LASR1 Without IPO	2882.854398	2040.691122	1737.673142	1555.488402
LASR2 With IPO	12254.51751	8937.136798	7607.368184	6815.877042
LASR2 Without IPO	12632.16176	8941.950162	7614.178582	6815.877042

Table 7.15: Total elapsed time measured in seconds when the IPO simulator is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 With IPO	17533	7181.56	4842.92	3650.45
LASR1 Without IPO	26877.8	8960.76	5377.32	3840.85
LASR2 With IPO	76813	31457.1	21215.7	15988.5
LASR2 Without IPO	117774	39258.7	23556.5	16826.6
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 With IPO	2900.08	2408.85	2059.59	1793.48
LASR1 Without IPO	2987.52	2444.64	2067.66	1793.48
LASR2 With IPO	12705.6	10552.5	9018.85	7852.05
LASR2 Without IPO	13086.9	10708.4	9059.66	7852.05

Table 7.16: Comparison between IPO's simulation results and the prototyping results. Total elapsed time measured in seconds when the LASR traces are replayed by the iPipe simulator and prototype. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 Simulation	0.347677265	0.198554587	0.09938036	0.04957236
LASR1 prototyping	0.256921138	0.161087725	0.040853332	0.024535098
LASR2 Simulation	0.347793231	0.198722831	0.099369601	0.049808042
LASR2 prototyping	0.256921138	0.161087725	0.040853332	0.024535098
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 Simulation	0.029268423	0.014640192	0.003902963	0
LASR1 prototyping	0.029895457	0.00053829	0.000894436	0
LASR2 Simulation	0.029136006	0.014558664	0.004504584	0
LASR2 prototyping	0.029895457	0.00053829	0.000894436	0

7.1.3 The IPODS Prototype

Now we are in a position to present prototyping results for the IPODS scheme. We develop a prototype of a multi-level distributed storage system where the IPODS prefetching scheme is implemented. The client and storage nodes in the prototype are connected by the fast Ethernet. Like the iPipe and IPO prototypes, the IPODS prototype allow us to measure elapsed time of an application issuing 1000 read requests to the multi-level storage system. In the IPODS prototype, we set the block size to 200 MB and the maximum concurrent reads to 15. Thus, the TIP module in the IPODS prototype can choose the number of informed prefetching buffers anywhere between 1 and 15. We compare the IPODS prototyping results with the simulation results.

Table 7.17 reveals the prototyping results in terms of elapsed time in seconds. As mentioned earlier, the IPODS prototype processes 1000 read requests issued by the application that replay the two LASR traces. The IPODS prototype confirms that IPODS can reduced the application's elapsed time by approximately 6% when the number of prefetching buffers is set to a small value (e.g., 1-3). With the increasing number of prefetching buffers, the performance gaps is narrowing.

The application implemented in the IPODS prototype and simulator used the two LASR traces machine01 (LASR1) or machine06 (LASR2), which contain 11686 and 51206 read requests, respectively. Table 7.18 shows the prototyping results in terms of elapsed time in seconds when the application is executed in the prototype. Table 7.18 indicates that the IPODS prototype and simulator share the same performance trend in term of improving system performance over the non-IPODS case.

Table 7.19 plots the the IPODS simulation results presented in the previous IPODS chapter. Table 7.20 shows the comparison between the simulation and prototyping results. We observe that both the simulator and prototype agree with each other when the number of prefetching buffers is set from 1 to 9. When there are 11 to 15 buffers, both the simulator and prototype show 0% improvement over the non-IPODS case due to the pipeline limitation.

Table 7.17: Total elapsed time measured in seconds when the IPODS prototype is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
With IPODS	4158	1401.48	888.555	650.74
Without IPODS	4430	1520.81	916.11	667.74
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
With IPODS	518.703	428.184	364.925	320.925
Without IPODS	526.37	429.777	365.925	320.925

Table 7.18: Total elapsed time measured in seconds when the LASR traces are replayed by the IPODS prototype. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 With IPODS	48590.388	16377.69528	10383.65373	7604.54764
LASR1 Without IPODS	51768.98	17772.18566	10705.66146	7803.20964
LASR2 With IPODS	212914.548	71764.18488	45499.34733	33321.79244
LASR2 Without IPODS	226842.58	77874.59686	46910.32866	34192.29444
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 With IPODS	6061.563258	5003.758224	4264.51355	3750.32955
LASR1 Without IPODS	6151.15982	5022.374022	4276.19955	3750.32955
LASR2 With IPODS	26560.70582	21925.5899	18686.34955	16433.28555
LASR2 Without IPODS	26953.30222	22007.16106	18737.55555	16433.28555

We use SCP (secure copy) to read data from the remote HDDs and SSDs; we take the average of the disk access time measured during the parameter validation process. For this reason, the IPODS simulation and prototyping results are very close when X_{cache} is set to 1. When X_{cache} increases, the IPODS prototype's elapsed time is slightly higher (about 1%-8%) than those of the IPODS simulator.

7.2 Summary

The goal of this part of study is three-fold. First, we built real-world storage systems that implements our pipelined prefetching solutions. Second, we evaluated the performance of our informed prefetching solutions in a real-world system. Last, we validated the correctness of our simulators presented in the previous chapters by comparing simulation results with the prototyping results.

Table 7.19: Total elapsed time measured in seconds when the LASR traces are replayed by the IPODS simulator. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 With IPODS	48590.9	16200	9933.42	7246.72
LASR1 Without IPODS	51769	17259.2	10357.2	7397.95
LASR2 With IPODS	212915	70973.4	43515	31742.4
LASR2 Without IPODS	226843	75615.6	45371.9	32409.7
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 With IPODS	5750.29	4704.81	3982.53	3454.88
LASR1 Without IPODS	5754.39	4708.83	3982.53	3454.88
LASR2 With IPODS	25202.5	20621.7	17449.7	15123.9
LASR2 Without IPODS	25206.6	20625.7	17449.7	15123.9

Table 7.20: Total elapsed time measured in seconds when the IPODS simulator is tested. The number of prefetching buffers is set to a range between 1 to 15. The block size is 200 MB.

$\mathbf{X}_{cache}$	**1**	**3**	**5**	**7**
LASR1 Simulation	0.061390021	0.061370168	0.040916464	0.02044215
LASR1 prototyping	0.061399549	0.078464765	0.030078266	0.025459011
LASR2 Simulation	0.061399294	0.061392094	0.040926212	0.020589515
LASR2 prototyping	0.061399549	0.078464765	0.030078266	0.025459011
$\mathbf{X}_{cache}$	**9**	**11**	**13**	**15**
LASR1 Simulation	0.0007125	0.000853715	0	0
LASR1 prototyping	0.0145658	0.003706573	0.0027328	0
LASR2 Simulation	0.000162656	0.000193933	0	0
LASR2 prototyping	0.0145658	0.003706573	0.0027328	0

The three prototypes replay real-world traces to evaluation the performance of the iPipe, IPO, and IPODS prefetching schemes. After comparing the simulators with the prototypes, we conclude that both our prototyping and simulations results share similar performance trends. Specifically, for iPipe and IPO, the simulation and prototyping results vary by 3-18%. For IPODS, the simulation and prototyping results vary by only 1-8%. The prototyping results confirm that the simulator presented in the previous chapters can be used to evaluate the performance of our proposed pipelined prefetching solutions.

Chapter 8

Conclusions & Future Work

This chapter first concludes this dissertation by summarizing our contributions to the field of informed prefetching. Then, we outline future research directions.

8.1 Main Contributions

Transparent Informed prefetching or TIP aims to utilize storage bandwidth and prefetch data blocks to the cache before they are accessed by applications. Traditional informed prefetching schemes can hide the latency of accessing storage systems by invoking disk I/O parallelisms and fetching data based on application-disclosed hints. Disk access time effects the informed prefetching because long disk access time increases application stalls. When disk speed is slow, informed prefetching needs more buffers at the expense of demand caching (LRU) for prefetching.

A multi-level (a.k.a., hybrid) storage system consists of multiple types of storage devices that differ in their specifications (hardware, speed, size, and the like) [87]. Upper levels exhibit better speed performance. Multi-level storage systems offer cost-effective solutions for large-scale data centers.

In this dissertation, we investigate approaches to incorporating the TIP module in multi-level storage systems to leverage informed prefetching mechanisms to boost I/O performance of storage systems. The goal of this research is to demonstrate that building an informed prefetching pipeline can significantly improve the I/O performance of multi-level storage systems.

To achieve the above goal, we proposed three solutions (iPipe, IPO, and IPODS) that perform informed prefetching in a pipeline manner. We illustrate a way of using application

hints to initiate prefetching among multiple storage levels like main memory, solid state disks, and hard disk drives. The centerpiece of the iPipe, IPO, and IPODS schemes is a pipeline in which we split the informed prefetching process into a set of independent prefetching steps among the multiple storage levels. We showed how to integrate this pipeline with informed prefetching and caching to manage I/O buffers at various storage levels.

In the iPipe scheme, we assume that multi-level storage systems have sufficient I/O bandwidth and enough I/O parallelism, meaning that there is no I/O congestions in the storage systems. This assumption was relaxed when we extend iPipe into the IPO scheme. Finally, we developed the IPODS scheme to meet the needs of distributed and parallel storage systems. The next three subsections summarizes our main contributions.

8.1.1 iPipe: An Pipelined and Informed Prefetching

In the first phase of this dissertation research, we developed iPipe - a new informed prefetching technique powered by a pipeline. We started this research by assuming that multi-level storage storage systems have scalable I/O bandwidth and parallelism, meaning that that there is no I/O congestion. This assumption of scalable I/O enables a prefetching mechanism to issue a large number of concurrent read requests without introducing long I/O queues.

iPipe aims to reduce application I/O stalls and elapsed times by preloading hinted data blocks to multi-level storage systems' uppermost level, which has better I/O performance than lower levels. iPipe contains multiple informed prefetching modules running concurrently in a pipeline manner. The number of prefetchers in a storage server depends on the number of storage levels. An N-level storage server requires N-1 prefetchers. The prefetchers in lower-level storage devices help in boosting I/O performance of the prefetchers in upper-levels. Consequently, iPipe reduces stall times, application elapsed times, and the prefetching horizon. iPipe also improves the benefit of using more buffers for prefetching.

We developed an iPipe simulator to evaluate the performance of our pipelined prefetching scheme. The accuracy of the iPipe simulator was validated by a prototype running in a real-world multi-level storage system. The simulation results show that iPipe reduces the total elapsed time by up to 56% when the block size is 10 MB and by 34% when the block size is 200 MB. The experimental results also show that as the number buffers increases, the performance gap between iPipe and non-iPipe cases diminishes because the prefetching module in the upper level becomes more efficient to reduce the application stalls.

8.1.2 IPO: Informed Prefetching Optimization

In the iPipe study, we assumed that multi-level storage systems have sufficient I/O bandwidth. Unfortunately, our empirical experiments indicate that parallel storage systems may have I/O congestions; evidence shows that there is a maximum number of concurrent read requests processed in multi-level storage systems. In the second phase of our dissertation study, we show that an upper-level prefetching mechanism many not fully utilize I/O bandwidth, suggesting that unused I/O bandwidth can be allocated for lower-level prefetching mechanisms to bring hinted blocks from lower-level to upper-level storage in a pipeline fashion. We conducted experiments to illustrate that the pipelined prefetching process largely depends on available I/O bandwidth for lower-level prefetchers.

In the IPO scheme, the lower-level prefetcher trigger a informed prefetching request when the following conditions are met. First, the upper level SDDs have sufficient available I/O bandwidth. Second, fetching hinted blocks from HDDs to SSDs will not saturate the I/O bandwidth of SSDs and HDDs. We believe that the main concern is SSDs' bandwidth, because the SSDs are accessed by two prefetching modules. Compared with HDDs, SSDs are more likely to have their I/O bandwidth saturated. Our IPO design ensures that a lower-level prefetcher will never slow down the performance of its upper-level prefetching counterpart.

The experimental results show that performance improvements offered by IPO are not as significant as those offered by iPipe due to the limited I/O bandwidth. Nevertheless, IPO reduces elapsed time by approximately 56% when the X_{cahce} is set to 1. As the X_{cahce} value increases, IPO's performance improvement is decreasing. This part of the study indicates that our proposed IPO is very beneficial to parallel storage systems offering sufficient I/O bandwidth.

8.1.3 IPODS: Pipelined Prefetching in Distributed/Parallel Storage Systems

In the third phase of the dissertation study, we extended the IPO schemes to meet the needs of distributed/parallel multi-level storage systems, where massive amounts of data are allocated to a set of storage servers built with multi-level storage devices.

Similar to iPipe and IPO, IPODS have several prefetching modules residing in each storage server. Unlike iPipe and IPO, IPODS should incorporate a prefetcher to bring data from remote storage servers to a client's local I/O buffers. IPODS has a coordination mechanism between a pair of client local I/O buffer and buffers in SSDs (i.e., upper-level device) of a remote storage server. IPODS checks if hinted blocks have been cached in the local buffer. If the hinted blocks are not available in the local buffer, a request will be sent to the remote storage server to have the hinted blocks fetched from the server.

We developed the IPODS simulator and prototype to evaluate the performance of IPODS deployed in distributed and parallel storage systems. Our results show that IPODS can judiciously reduce the elapsed times of applications by approximately 6%. Importantly, IPODS shows its best performance improvement when the there are a limited number of prefetching buffers available in the upper-level storage devices.

8.1.4 Prototypes for iPipe, IPO, and IPODS

We implemented three prototypes for the above informed prefetching schemes. We developed our prototypes using a computing cluster in our research laboratory at Auburn.

The prototypes replays real-world traces to evaluation the performance of the three pipelined prefetching mechanism. The prototypes help us to validate the correctness of the simulators in which iPipe, IPO, and IPODS are implemented. The validate process was performed by comparing experimental results obtained from the prototypes with those obtained from the simulators.

After comparing the simulators with the prototypes, we confirmed that both our prototyping and simulations results share similar performance trends. For example, for iPipe and IPO, the simulation and prototyping results vary by 3-18%. When it comes to IPODS, the simulation and prototyping results vary by only 1-8%. The prototyping results confirm that the three simulators can be used to evaluate the performance of the iPipe, IPO, and IPODS solutions.

8.2 Future Work

There are several important issues that we were not able to address in this dissertation. These issues open many future research opportunities for us. In this section, we will discuss our future research directions.

8.2.1 Data Migration

Our pipelined prefetching schemes can hide the latency of accessing storage systems by loading hinted blocks to upper-level storage devices. Our solutions invoke disk I/O parallelisms and fetching data based on application-disclosed hints. When data blocks are fetched to upper-level storage devices, the lower-level storage devices keep the original copies of the data blocks.

As the first future research direction, we will investigate data migration schemes. Instead of sending copies of the data to the uppermost level, our prefetching mechanisms can migrate the entire data blocks; obviously, the uppermost level's space limitation is an obstacle that we may face. Data migration requires moving data back and forth among the different

storage levels. This activity consumes storage system I/O bandwidth, because a prefetching scheme must move the prefetched data blocks back to lower storage level when the blocks are evicted from I/O buffers. We will show that the data migration scheme will be very beneficial to applications with mixed workload of both reads and writes.

8.2.2 The Cost-Benefit Model

Our pipelined prefetching schemes relies on the important cost-benefit model, that helps to divide the compound cache space between the demand misses cache (LRU) and prefetching. The division may change frequently in a dynamic workload conditions. Our pipelining mechanisms can adjust their pipelined prefetching depth according to available I/O bandwidth. In this dissertation research, we fix the number of prefetching buffers. This strategy is good as long as workload conditions do not change dramatically.

In the future, we will extend our pipelined prefetching schemes by dynamically adjusting the number of prefetching buffers for fast changing I/O load. To allow our prefetching modules to change the number of prefetching buffers, we must rely on a good cost-benefit model that can estimate the cost and benefit of increasing or decreasing prefetching buffer size. We plan to integrate the cost-benefit model in multiple prefetchers deployed in multiple level storage systems. Because each prefetching module individually handles a pair of two consecutive storage level, each prefetcher must independently control its prefetching buffer size.

8.2.3 Write Performance

We only focus on read performance of multi-level storage systems. Our proposed iPipe, IPO, and IPODS are designed to improve performance of read-intensive applications supported by multi-level storage systems. When it comes to write-intensive applications, iPipe, IPO, and IPODS may not be able to boost the I/O performance of the applications.

This motivate us to investigate in the future a pipelined prefetching module that hand efficiently handle write requests. We also will develop a simulator to evaluate the performance of multi-level storage systems supporting applications with mixed workload of both writes and reads.

8.2.4 Most Recently Used Policy

The TIP module employed in our solutions uses most recently used policy (MRU)for informed caching. As discussed in chapter 3, we did not implement informed caching in our solution. This is mainly because we cannot predict if a particular hinted access will be cached when TIP performs prefetching. In one of our future project, we will integrate the (MRU) policy into our pipelined prefetching mechanism for multi-level storage systems. Such an integration will enable us to determine if a particular hinted access will be cached at its prefetching phase.

8.2.5 Extending Storage Hierarchy

In this dissertation research, we only evaluated the performance of iPipe, IPO, and IPODS using two-level storage systems that contain solid state disks and hard drives. In the future, we plan to further evaluate our three pipelined prefetching scheme in a three-level storage systems, where tapes are used as the lowest storage level. Tapes have a huge storage capacity and less speed performance compared to HDDs and SSDs. IBM System Storage TS2240 Tape Drive Express is a good example of this [95]. It has a capacity of 1.6 TB and a speed of approximately 120 MB/s.

To incorporate tapes, we need to extend our added a prefetcher to fetch hinted blocks from tapes and hard disks and solid state disks. Fetching hinted blocks from tapes is different from fetching blocks from hard drives, because data tapes have very slow seek times. However, once the tape is positioned, the tape drive can stream data much faster. This

motivate us to group hinted blocks together so that hinted blocks sequentially stored on the tape can be fetched at the same time to reduce long seek time.

8.2.6 Caching and Benchmarking

Caching prevents our prefetching schemes from loading hinted blocks data again. Both caching and pipelined prefetching improve performance of informed prefetching. In our solutions, we assumed the worst case in which informed caching was not implemented. In our future work, we will seamlessly integrate informed caching with pipelined prefetching mechanisms to further improve I/O performance of multi-level storage systems.

We will evaluate performance of the new design using real-world I/O-intensive benchmarks. The benchmarks to be considered in our future research include XDataSlice, Sphinx, Agrep, and Gnuld. In addition, we will use the IOzone file system benchmark used to measure the performance of file system operations [100]. Other benchmarks to be considered are some Linux commands such as (grep) and (wc) [31].

8.2.7 Various Solid State Disks

In our iPipe study, we conducted extensive experiments using various solid state disks. However, the system parameters of these solid state disks were collected from devices vendors' websites [90] [91] [92] [93] [94] [95]. In the future, we will have to validate these parameters in the context of multi-level storage systems.

Solid state disks provided by different vendors have different capacity and speed [91] [92] [93]. In our future work, we will test our solutions using a wide range of SSD products. We will purchase more different types of solid state disks to perform such experiments.

8.2.8 Block Sizes

Our tests show that the SSD's performance in respect to the HDD is affected by the data block size. For example, reading a data block from SSD is faster than from HDD about

56% when the block size is 10 MB and by about 34% when it is 200 MB. Also, SSDs in a distributed storage system shows better performance than HDDs by about 6% when the block size is 200 MB.

In our future work, we will investigate a possibility of varying block sizes in a multi-level storage system. Future multi-level storage systems should dynamically choose the most appropriate block size for each storage level under I/O different workload.

8.2.9 Fast Networks for Distributed Storage Systems

Our prototypes show that IPODS provided 6% performance improvement when it was implemented in a 2-level distributed storage system in our laboratory. We observed that the network is an I/O performance bottleneck. Madhyastha *et al.* used a fast network to implement their solutions [65]. In our future work, we also plan to further improve the performance of IPODS using a fast network, such as fiber channel networks.

Bibliography

[1] R. Patterson, Hugo, G. Gibson, D. Stodolsky, and J. Zelenka: Informed prefetching and caching, *In Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79-95, CO, USA, 1995.

[2] J. Griffioen and R. Appleton: Reducing file system latency using a predictive approach, *In Proceedings of the 1994 USENIX Annual Technical Conference*, pages 197 207, Berkeley, CA, USA, 1994.

[3] Chuanpeng Li , Kai Shen , Athanasios E. Papathanasiou: Competitive prefetching for concurrent sequential I/O, *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007,* March 21-23, 2007, Lisbon, Portugal [doi¿10.1145/1272996.1273017]

[4] Vellanki, V., Chervenak, A.L., *A Cost-Benefit Scheme for High Performance Predictive Prefetching*, *Proceedings of the AMC/IEEE SC99 Conference*, 1999.

[5] Chen, Y., Byna, S., Sun, X., *Data Access History Cache and Associated Data Prefetching Mechanisms*, *Proceedings of the AMC/IEEE Conference on Supercomputing*, Reno, NV, Nov, 2007, pp. 1-12.

[6] Wang, J.Y.Q, Ong, J.S., Coady,Y., Feeley, M.J., *Using Idle Workstations to Implement Predictive Prefetching*, *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, Aug. 2000, pp. 87-94.

[7] Domenech, J., Sahuquillo, J., Gil, J.A., Pont, A., *The Impact of the Web Prefetching Architecture on the Limits of Reducing Users Perceived Latency*, IEEE/WIC/ACM International Conference on Web Intelligence, Hong Kong, China, Dec. 2006, 740-744.

[8] Jeon, J., Lee, G., Cho, H., Ahn, B., *A Prefetching Web Caching Method Using Adaptive Search Patterns*, 2003 IEEE Pacific Rim Conference On Communications, Computers, And Signal Processing, Aug. 2003, vol. 1, pp. 37-40.

[9] Z. Zhang ; K. Lee ; X. Ma ; Y. Zhou: PFC: Transparent Optimization of Existing Prefetching Strategies for Multi-Level Storage Systems, *In Proceedings of 28th International Conference on Distributed Computing System*, pages 740 - 751, Beijing, China, 2008.

[10] M. Huizinga, D. and S. Desai: Implementation of informed prefetching and caching in linux, *In Proceedings of the International Conference on Information Technology*, pages 443 448, Las Vegas, NV, USA, 2000.

[11] R. Hugo Patterson , Garth A. Gibson , M. Satyanarayanan: A status report on research in transparent informed prefetching, *ACM SIGOPS Operating Systems Review*, v.27 n.2, pages: 21-34, 1993.

[12] A.Tomkins, R. Hugo Patterson and G. Gibson: Informed multi-process prefetching and caching, *In Proceedings of the 1997 ACM SIGMETRICS international conference on measurement and modeling of computer systems*, pages 100 - 114, Seattle, WA , USA, 1997.

[13] Fay Chang , Garth A. Gibson: Automatic I/O hint generation through speculative execution, *Proceedings of the third symposium on Operating systems design and implementation*, p.1-14, February 1999, New Orleans, Louisiana, United States

[14] Pei Cao , Edward W. Felten , Anna R. Karlin , Kai Li: A study of integrated prefetching and caching strategies, *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*,p.188-197, May 15-19, 1995, Ottawa, Ontario, Canada [doi¿10.1145/223587.223608].

[15] Susanne Albers , Markus Bttner : Integrated prefetching and caching in single and parallel disk systems, *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, June 07-09, 2003, San Diego, California, USA [doi¿10.1145/777412.777431].

[16] R.H. Patterson, G.A. Gibson, M. Satyanarayanan: Using Transparent Informed Prefetching (TIP) to Reduce File Read Latency , *In Proceedings of Conference on Mass Storage Systems and Technologies*, Pages: 329-342, Greenbelt, MD, 1992.

[17] D. Rochberg, G.A. Gibson: Prefetching over a network: early experience with CTIP, *ACM SIGMETRICS Performance Evaluation Review*, v.25 n.3, Pages: 29-36, 1997.

[18] F. Jason: Multi-level memory prefetching for media and stream processing , *In IEEE International Conference on Multimedia and Expo*, pages 101 104, St. Louis, MO, USA, 2002.

[19] Song J. and X. Zhang: Ulc: A file block placement and replacement protocol to effectively exploit hierarchical locality in multilevel buffer caches, *In Proceedings of the 24th International Conference on Distributed Computer Systems*, pages 168 - 177, St. Louis, MO, USA, 2004.

[20] J. Kang and W. Sung: A multi-level block priority based instruction caching scheme for multimedia processors, *In Proceedings of the 24th International Conference on Distributed Computer Systems*, pages 125 132, Antwerp, Belgium, 2001.

[21] S. Przybylski, M. Horowitz, and J. Hennessy: Performance tradeoffs in cache design, *In 15th Annual International Symposium on Computer Architecture*, pages 290 298, Honolulu, HI, USA, 1988.

[22] S. Przybylski, M. Horowitz, and J. Hennessy: Characteristics of performance-optimal multi-level cache hierarchies, *In 16th Annual International Symposium on Computer Architecture*, pages 114 - 121, Stanford University, CA, USA, 1989.

[23] M. Nijim: Modelling Speculative Prefetching for Hybrid Storage Systems, *In Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, pages 143 - 151, Macau, 2010.

[24] M. Nijim, Z. Zong, X. Qin, Y. Nijim: Multi-Layer Prefetching for Hybrid Storage Systems: Algorithms, Models, and Evaluations, *In 2010 39th International Conference on Parallel Processing Workshops, DOI 10.1109/ICPPW.2010.18*.

[25] T. Kimbrel, P. Cao, E. Felten, A. Karlin, K. Li: Integrated Parallel Prefetching and Caching, *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 262 - 263, PA,USA, 1996.

[26] R. Hugo Patterson, G. Gibson: Exposing I/O concurrency with informed prefetching, *Proceedings of the third international conference on on Parallel and distributed information systems*, pages 7 - 16, Austin, TX, USA, 1994.

[27] Jon A. Solworth , Cyril U. Orji: Write-Only Disk Caches, *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, p.123-132, May 23-26, 1990, Atlantic City, New Jersey, United States [doi¿10.1145/93597.98722]

[28] David Kotz , Carla Schlatter Ellis: Caching the writeback policies in parallel file systems, *Journal of Parallel and Distributed Computing*, v.17 n.1-2, p.140-145, Jan./Feb. 1993 [doi¿10.1006/jpdc.1993.1012]

[29] Mendel Rosenblum: The design and implementation of a log-structured file system, *University of California at Berkeley*, Berkeley, CA, 1992

[30] Y. Chen ; Byna, S. ; X. Sun ; Thakur, R. ; Gropp, W: Exploring Parallel I/O Concurrency with Speculative Prefetching, *Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 422 - 429, Portland, OR, USA, 2008.

[31] J. Lewis, M. I. Alghamdi, M. A. Assaf, X.-J. Ruan, Z.-Y. Ding, and X. Qin: An Automatic Prefetching and Caching System, *In Proceedings of the 29th International Performance Computing and Communications Conference (IPCCC)*, 2010.

[32] Y. Chen ; Byna, S. ; X. Sun ; Thakur, R. ; Gropp, W: Hiding I/O Latency with Preexecution Prefetching for Parallel Applications, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1 - 10, Austin, TX, USA, 2008.

[33] Gabriel Rivera , Chau-Wen Tseng: Locality optimizations for multi-level caches, *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*,p.2-es, November 14-19, 1999, Portland, Oregon, United States [doi¿10.1145/331532.331534].

[34] Achim Kraiss , Gerhard Weikum: Integrated document caching and prefetching in storage hierarchies based on Markov-chain predictions, *The VLDB Journal The International Journal on Very Large Data Bases*,v.7 n.3, p.141-162, August 1998 [doi¿10.1007/s007780050060]

[35] Surendra Byna , Yong Chen , Xian-He Sun , Rajeev Thakur , William Gropp: Parallel I/O prefetching using MPI file caching and I/O signatures, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, November 15-21, 2008, Austin, Texas.

[36] Xiaonan Zhao, Zhanhuai Li, and Leijie Zeng: A hierarchical storage strategy based on block-level data valuation, *In Fourth International Conference on Networked Computing and Advanced Information Management*, pages 36 41, Gyeongju, South Korea, 2008.

[37] Song Jiang, Kei Davis, and Xiaodong Zhang: Coordinated multilevel buffer cache management with consistent access locality quantification, *In IEEE Transcations on Computers*, volume 56, pages 95 108, Baltimore, MD, USA, 2007.

[38] Srinivas Kashyap, Samir Khuller, Yung-Chun Wan, and Leana Golubchik: Fast reconfiguration of data placement in parallel disks, *Proceedings of the eighth Workshop on Algorithm Engineering and Experiments and the Third Workshop on Analytic Algorithmics and Combinatorics* , 2006.

[39] Mary G. Baker , John H. Hartman , Michael D. Kupfer , Ken W. Shirriff , John K. Ousterhout: Measurements of a distributed file system, *Proceedings of the thirteenth ACM symposium on Operating systems principles*, p.198-212, October 13-16, 1991, Pacific Grove, California, United States [doi¿10.1145/121132.121164]

[40] Mirjana Spasojevic , M. Satyanarayanan: An empirical study of a wide-area distributed file system, *ACM Transactions on Computer Systems (TOCS)*, v.14 n.2, p.200-222, May 1996 [doi¿10.1145/227695.227698]

[41] A Vijay Srinivas , D. Janakiram : A model for characterizing the scalability of distributed systems, *ACM SIGOPS Operating Systems Review* , Volume 39 Issue 3, July 2005.

[42] M. Satyanarayanan : Scalable, Secure and Highly Available Distributed File Access, *Computer*, vol. 23, no. 5, pp. 9-21, May 1990.

[43] Dhruba Borthakur: The Hadoop Distributed File System: Architecture and Design, *The Apache Software Foundation.*, 2007, DOI: http://hadoop.apache.org/ common/docs/r0.18.0/hdfs_design.pdf

[44] Dhruba Borthakur: HDFS Architecture, *The Apache Software Foundation.*, 2008, DOI: http://hadoop.apache.org/ common/docs/r0.20.0/hdfs_design.pdf

[45] Shafer, J., Rixner S., Cox A.L : The Hadoop distributed filesystem: Balancing portability and performance , *IEEE International Symposium on Performance Analysis of*

Systems Software (ISPASS), pp. 122 - 133 , White Plains, NY , March 2010. DOI: 10.1109/ISPASS.2010.5452045 .

[46] Diana Moise, Gabriel Antoniu, Luc Boug: Improving the Hadoop map/reduce framework to support concurrent appends through the BlobSeer BLOB management system , *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, pp. 834 - 840 , Chicago, IL, June 2010. DOI: 10.1145/1851476.1851596

[47] Jeffrey Dean , Sanjay Ghemawat: MapReduce: a flexible data processing tool, *Communications of the ACM*, , v.53 n.1, January 2010 [doi¿10.1145/1629175.1629198]

[48] Jeffrey Dean , Sanjay Ghemawat: MapReduce: simplified data processing on large clusters, *Communications of the ACM*, v.51 n.1, January 2008 [doi¿10.1145/1327452.1327492]

[49] G. Hughes; J. Murray: Reliability and security of RAID storage systems and D2D archives using SATA disk drives, *ACM Transactions on Storage*, Vol. 1, Issue 1, Pages: 95 107, 2005.

[50] Q. Zou; Y. Zhu, D. Feng: A Study of Self-similarity in Parallel I/O Workloads, *MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Pages:1 - 6 , 2010, DOI: 10.1109/MSST.2010.5496978 .

[51] Ganger, G.R. ; Worthington, B.L. ; Hou, R.Y. ; Patt, Y.N: Disk arrays: high-performance, high-reliability storage subsystems, Journal: Computer, issn: 0018-9162, volume 27, pages 30-36, doi: 10.1109/2.268882 Ann Arbor, MI, USA, 1994.

[52] Alexander Thomasian: Multi-level RAID for very large disk arrays, *ACM SIGMETRICS Performance Evaluation Review*, v.33 n.4, March 2006 [doi¿10.1145/1138085.1138091]

[53] Maria E. Gomez , Vicente Santonja: Analysis of Self-Similarity in I/O Workload Using Structural Modeling, *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems,* p.234, March 24-28, 1999.

[54] Nancy Tran , Daniel A. Reed: ARIMA time series modeling and forecasting for adaptive I/O prefetching, *Proceedings of the 15th international conference on Supercomputing,* p.473-485, June 2001, Sorrento, Italy [doi¿10.1145/377792.377905]

[55] James Oly , Daniel A. Reed: Markov model prediction of I/O requests for scientific applications, *Proceedings of the 16th international conference on Supercomputing,* June 22-26, 2002, New York, New York, USA [doi¿10.1145/514191.514214]

[56] J. No: A Design for Hybrid File System, *European Conference for the APPLIED MATHEMATICS and INFORMATICS*, Applied Mathematics and Informatics, ISBN: 978-960-474-260-8, Pages: 143 - 148, Vouliagmeni, Athens, Greece, 2010.

[57] G. Gibson et al: A Cost-Effective, High-Bandwidth Storage Architectures, *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, 1998.

[58] Mishra, S.K. ; Mohapatra, P.: Performance Study of RAID-5 Disk Arrays with Data and Parity Cache, *Proceedings of 1996 International Conference on Parallel Processing*, Pages: 222-229, Ithaca, NY , 1996.

[59] S.H. Baek et al: Reliability and performance of hierarchical RAID with multiple controllers, *Proceedings of twentieth annual ACM symposium on principles of distributed computing*, Pages: 246 254, 2001.

[60] W. MEADOR: DISK ARRAY SYSTEMS, *Proceedings of COMPCON Spring '89. Thirty-Fourth IEEE Computer Society International Conference*, Pages: 143 - 146 , San Francisco, CA , USA, 1989.

[61] Y. Wu, A. G. Dimakis, and K. Ramchandran: Deterministic regenerating codes for distributed storage, *presented at the Allerton Con. Control, Computing, and Communication*, Urbana-Champaign, IL, Sep. 2007.

[62] Dimakis A.G., Godfrey P.B., Yunnan Wu, Wainwright M.J., Ramchandran K.: Network Coding for Distributed Storage Systems, *IEEE TRANSACTIONS ON INFORMATION THEORY*, VOL. 56, NO. 9, SEPTEMBER 2010

[63] R. Hou, J. Menon; and Y. Patt: Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array, *Proceedings of on Systems Sciences*, Pages: 70-79, 1993.

[64] A. Drapeau ; R. Katz: Striped tape arrays, *Proceedings of 12th IEEE Symposium on Mass Storage Systems*, 1993.

[65] T.Madhyastha; G. Gibson; C. Faloutsos: Informed prefetching of collective input/output requests, *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Portland, Oregon, 1999.

[66] Fay Chang; Jeffrey Dean; Sanjay Ghemawat; Wilson C. Hsieh; Deborah A. Wallach; Mike Burrows; Tushar Chandra; Andrew Fikes; Robert E. Gruber: Bigtable: A Distributed Storage System for Structured Data, *ACM Transactions on Computer Systems (TOCS)*, v.26 n.2, pages: 1-26, 2008.

[67] Brian Tierney ; Jason Lee ; Ling Tony Chen ; Hanan Herzog ; Gary Hoo ; Guojun Jin ; William E. Johnston: Distributed parallel data storage systems: a scalable approach to high speed image servers, *Proceedings of the second ACM international conference on Multimedia*, pages: 399-405, San Francisco, CA, 1994.

[68] Edward K. Lee ; Chandramohan A. Thekkath: Petal: distributed virtual disks, *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages: 84-92, Cambridge, Massachusetts, 1996.

[69] Moyer, S.A.; Sunderam, V.S: PIOUS: a scalable parallel I/O system for distributed computing environments, *Proceedings of Scalable High-Performance Computing Conference*, pages: 71 - 78, Knoxville, TN, 1994.

[70] D. Feng, Q. Zou, H. Jiang, and et al.: A novel model for synthesizing parallel i/o workloads in scientific applications, *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'08)*, Tsukuba, Japan, September 2008.

[71] Luis Cabrera , Darrell D.E. Long: SWIFT: USING DISTRIBUTED DISK STRIPING TO PROVIDE HIGH I/O DATA RATES, *University of California at Santa Cruz*, Santa Cruz, CA, 1991.

[72] D. D.E. Long , Bruce R. Montague , Luis Cabrera: SWIFT/RAID: A DISTRIBUTED RAID SYSTEM, *University of California at Santa Cruz*, Santa Cruz, CA, 1994.

[73] Sanjay Ghemawat , Howard Gobioff , Shun-Tak Leung: The Google file system, *Proceedings of the nineteenth ACM symposium on Operating systems principles*, October 19-22, 2003, Bolton Landing, NY, USA [doi¿10.1145/945445.945450]

[74] Sage A. Weil , Scott A. Brandt , Ethan L. Miller , Darrell D. E. Long , Carlos Maltzahn: Ceph:a scalable, high-performance distributed file system, *Proceedings of the 7th symposium on Operating systems design and implementation*, November 06-08, 2006, Seattle, Washington.

[75] John H. Howard , Michael L. Kazar , Sherri G. Menees , David A. Nichols , M. Satyanarayanan , Robert N. Sidebotham , Michael J. West: Scale and performance in a distributed file system, *ACM Transactions on Computer Systems (TOCS)*, v.6 n.1, p.51-81, Feb. 1988 [doi¿10.1145/35037.35059]

[76] Chandramohan A. Thekkath , Timothy Mann , Edward K. Lee: Frangipani: a scalable distributed file system, *Proceedings of the sixteenth ACM symposium on Operating systems principles*, p.224-237, October 05-08, 1997, Saint Malo, France [doi¿10.1145/268998.266694]

[77] Siegel A., Birman K., Marzullo K.: Deceit: A Flexible Distributed File System, *Proceedings of the Workshop on the Management of Replicated Data, 1990*, p.15-17, 8-9 Nov 1990, Houston, TX , USA .

[78] M. Satyanarayanan , John H. Howard , David A. Nichols , Robert N. Sidebotham , Alfred Z. Spector , Michael J. West,: The ITC distributed file system: principles and design, *Proceedings of the tenth ACM symposium on Operating systems principles*,p.35-50, December 1985, Orcas Island, Washington, United States [doi¿10.1145/323647.323633]

[79] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steer.: Coda: A highly available file system for a distributed workstation environment, *IEEE Transactions on Computers,*39(4):447459, 1990.

[80] Narayan, S., Chandy, J.A: Parity Redundancy in a Clustered Storage System, *International Workshop on Storage Network Architecture and Parallel I/Os, 2007. SNAPI.,*page(s): 17 - 24, Volume: Issue: , 24-24 Sept. 2007

[81] C.K. Yang, T. Mitra and T. Chiueh: A Decoupled Architecture for Application-Specific File Prefetching, *Freenix Track of USENIX 2002 Annual Conference,* 2002.

[82] R. W. Watson , R. A. Coyne: The parallel I/O architecture of the high-performance storage system (HPSS), *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, Page: 27, 1995.

[83] John H. Hartman , John K. Ousterhout: The Zebra striped network file system, *ACM Transactions on Computer Systems (TOCS)*, v.13 n.3, pages: 274-310, 1995.

[84] Brian L. Tierney , William E. Johnston , Hanan Herzog , Gary Hoo , Guojun Jin , Jason Lee , Ling Tony Chen , Doron Rotem: Using high speed networks to enable distributed parallel image server systems, *Proceedings of the 1994 conference on Supercomputing*, Pages: 610-619, Washington, D.C, 1994.

[85] W. Tuma: Comparison of drive technologies for high transaction databases, *Storage developer conference*, Santa Clara,2008, doi: www.storage-developer.org

[86] S. Rizvi ; T. Chung: Flash SSD vs HDD: High Performance Oriented Modern Embedded and Multimedia Storage Systems, *Proceedings of 2010 2nd International Conference on Computer Engineering and Technology*, Pages: V7-297 - V7-299, Chengdu, 2010.

[87] T. Kaneko: Optimal Task Switching Policy for a Multilevel Storage System, *IBM Journal of Research and Development*, vol.18, no.4, pp.310-315, July 1974.

[88] Nanopoulos, A., Katsaros, D., Manolopoulos, Y., *A Data Mining Algorithm for Generalized Web Prefetching*, IEEE Transactions on Knowledge and Data Engineering, Sep. 2003, vol. 15, no. 5.

[89] Hadoop Archive Guide, doi: http://hadoop.apache.org/mapreduce/docs/ r0.21.0/hadoop_ archives.html

[90] Kingston Valueram, doi: http://www.valueram.com/desktop/memory.asp

[91] Intel X25-E Extreme SATA Solid-State Drive, doi: http://www.intel.com/ design/flash/nand/extreme/index.htm

[92] Corsair Nova Series V64 Solid-State Hard Drive, doi: http://www.corsair.com/ solid-state-drives/nova-series.html

[93] Corsair Force Series F115 Solid-State Hard Drive, doi: http://www.corsair.com/ force-feries-f115-solid-state-hard-drive.html

[94] WD Caviar Green, doi: http://www.wdc.com/en/products/Products.asp?DriveID=559

[95] Ibm system storage ts2240 tape drive express model, doi. http://www03.ibm.com/systems/storage/tape/ts2240/.

[96] DELL PowerConnect 2824 Switch, doi:http://www.dell.com/us/business/p/powerconnect-2824/pd.

[97] Lasr trace machine01, doi:http://iotta.snia.org/traces/list/Subtrace?parent=LASR+Traces.

[98] Lasr trace machine06, doi:http://iotta.snia.org/traces/list/Subtrace?parent=LASR+Traces.

[99] RamSpeed Cache and memory benchmarking tool, doi: http://alasir.com/software/ramspeed/

[100] IOzone Filesystem Benchmark, doi: http://www.iozone.org/