ABSTRACT

ZHANG, ZHE. Adding Coordination to the Management of High-End Storage Systems. (Under the direction of Dr. X. Ma).

Today's scientific and commercial applications rely heavily on high-end computing (HEC) facilities, including large scale datacenters, supercomputers, and so forth. In these facilities, the storage subsystems are playing an increasingly important role in the overall computing experience perceived by users. Meanwhile, it is a challenging task to provide high performance and reliability to those high-end storage systems due to their high I/O demands, large scales, and complex architectures.

We observe that in addition to the well-recognized lack of I/O resources relative to computing demands in an aggregate perspective, one main challenge faced by high-end storage systems lies in the growing scale and complexity of the entire environment. Individually developed system components or algorithms often behave with isolated local optimizations, and handle concurrent user workloads without considering inter-workload relationships.

The author's Ph.D. research focuses on three novel instances of bringing adaptive coordination to the management of commercial and scientific high-end storage systems, at different levels of the HEC storage hierarchy. Firstly, on a single storage server, we present a memory cache allocation mechanism which coordinates multiple concurrent sequential access streams with different request rates. Our work is based on the interesting observation that this problem bears a strong resemblance to situations long studied in the field of *supply chain management (SCM)*, used by large vendors and retailers. Furthermore, in a multi-level storage architecture, we address the problem of information distortion in uncoordinated prefetching operations on different storage caches. We develop a simple information sharing mechanism, as well as a transparent hierarchy-aware optimization component named *PreFetching-Coordinator (PFC)*, which monitors both upper- and lower-level caches, and adjusts the aggressiveness of lower-level prefetching. Finally, we improve the data availability in an entire distributed storage system by coordinating it with the HPC job scheduler and remote data sources.

We implemented the proposed techniques in real software environments, including a state-of-the-art operating system kernel, a widely used job scheduler and a popular parallel

file system, as well as verified simulators. Our experimental results collected from real system experiments and simulations show that our proposed techniques can significantly improve system performance and reliability by coordinating among system components and requests.

Adding Coordination to the Management of High-End Storage Systems

by

Zhe Zhang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Operations Research and Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

____
Dr. R. Handfield

____
Dr. F. Mueller

____
Dr. S. Vazhkudai

____
Dr. X. Ma
Chair of Advisory Committee

____
Dr. W. Stewart
Co-Chair of Advisory Committee

DEDICATION

To my grandfather, Zhaolong Zhang.

BIOGRAPHY

Born in Xi'An city in northwestern China, Zhe Zhang moved with his family to Beijing, the capital of China when he was 10 years old. Zhe finished his undergraduate study in 2003 from the University of Science and Technology of China with a Bachelor's degree in Computer Science and Engineering. In 2004 summer he entered the North Carolina State University to pursue his graduate study co-majoring in Operations Research and Computer Science. He worked on stereo image rendering for his Master's thesis project under the direction of Dr. David McAllister and obtained his Master of Science degree in Operations Research in May 2006. Since summer 2006 Zhe has been conducting research in the Parallel Applications and Middleware (PALM) group led by Dr. Xiaosong Ma, with a focus on improving the reliability and efficiency of distributed systems. Zhe has worked as summer interns at the Oak Ridge National Laboratory, Cisco Systems, IBM T.J. Watson Research Center, and Microsoft Research Cambridge, respectively.

ACKNOWLEDGMENTS

During my graduate study at the North Carolina State University I have been very fortunate to work with and learn from many people.

First of all I would like to give special gratitude to my advisor, Dr. Xiaosong Ma, for her extraordinary patience and insights in guiding me through this stressful yet exciting period of time. Most importantly, her encouragement and respect for my creativity and her own attitude toward work inspired my interest in doing research and deeply affected my life. I am also very thankful to Dr. Sudharshan Vazhkudai and Dr. Frank Mueller who provided so much invaluable advice throughout my Ph.D. research and in particular directed the project on data fault tolerance. Sudharshan also mentored my internship at the Oak Ridge National Laboratory, which is the kickoff of my life in systems research filled with segmentation faults and kernel panics. I would also like to sincerely thank Dr. William Stewart and Dr. Robert Handfield for serving on my dissertation committee, and for providing the precious feedbacks in my preliminary exam.

I am also deeply grateful to my collaborators in my thesis research projects. Amit Kulkarni has made crucial contributions to the implementation of SCM-based algorithms in the Linux kernel, and I owe him an apology for dragging him to the lab in midnights before paper deadlines to fix our testbed. Kyuhyung Lee has contributed equally as myself in the development of PFC, and I have learned from his passionate and hard-working attitude. I am especially thankful to Chao Wang, my close collaborator in the fault tolerance project in the past three years and my two-time roommate in Oak Ridge and Raleigh. The project would not have succeeded without Chao's major contributions in developing the replication and recovery mechanisms in Lustre, and through working with him I gained tremendously on both development skills and vision on computer systems. My sincere appreciation also goes to Dr. Yuanyuan Zhou for her guidance in the prefetching projects and the great career advice, as well as Dr. John Cobb and Dr. Gregory Pike for their help in setting up experimental environments at ORNL.

I would like to thank the National Science Foundation and the United States Department of Energy for funding my Ph.D. research, and the Oak Ridge National Laboratory for providing multiple cluster testbeds and valuable job traces.

I thank all past and current members of the PALM group, Heshan Lin, Jiangtian

Li, Alex Balik, Feng Ji, Amey Deshpande, Jagan Srinivasan, Divya Dinakar, Fei Meng, Prakash Ramaswamy and Steve Koonce, for the inspiring discussions and interesting chats, which make the office an attractive place.

I am much obliged to Dr. Fan Ye, Dr. Hao Yang and Dr. Minkyong Kim at IBM T.J. Watson Research Center as well as Dr. Eno Thereska and Dr. Dushyanth Narayanan at Microsoft Research Cambridge for their wonderful mentorship of my internships, and their warm help in many aspects of my career development. I also want to thank Dr. David McAllister, who directed my Mater's thesis project and taught me a lot on doing research at the beginning of my graduate study.

My life would have been much less enjoyable without my friends at NCSU and elsewhere: Yuan Sun, Zhuang Wu, Yue Chao, Wei Mu, Hong Tian, Yingjie Li, Xuezhong Wang, Huiying Zhang, Hanyu Cui, Jianying Li, Dong Wu, Liming Pan and Zushou Hu. Thanks for all the wonderful moments.

I thank my dear parents, Shuqing Zhang and Xiuqing Du, who have always trusted and encouraged me without reservation, and taught me a lot about how to be an honest and humble person.

Finally, I want to express my deepest appreciation to the love and support from my wife, Xianhua Kong. She helped me through all the frustrations, shared my successes, calmly alerted me to my mistakes, and created so much happiness in our lives.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 High-end Computing Facilities

High-end computing (HEC), as opposed to commodity computing, refers to computing systems designed and developed for high performance and high reliability, which also have relatively high costs. Such systems are typically owned and operated by research institutes or companies, for purposes exceeding personal computing and entertainment. In the rest of this section, we discuss HEC in more details in the *scientific* and *commercial* contexts, respectively.

In the field of scientific computing, or computational science, programs are usually executed in parallel to solve large problems with high speed. HEC facilities with highly parallel architectures are critical for this parallel processing paradigm. Representative examples of such parallel systems include *clusters* of servers with high-speed interconnections, and *supercomputers*, which are large-scales proprietary machines with tightly integral of components.

Today's state-of-the-art supercomputers allow scientists to perform computation tasks at unprecedented scales and resolutions. A recent issue of *Science* published an article by Liu et al. [78] on simulating abrupt climate change for the purpose of understanding natural global warming. Aiming to complete the world's first continuous simulation of 21,000 years of Earth's climate history and provide forecasts for 200 years into the future, their work requires around 5 million processor hours of computing power [90]. Fortunately, with the Jaguar system [57] at the Oak Ridge National Laboratory [89] , the second fastest supercomputer in the world as of June 2009 [58], their simulation is expected to finish in 2011.

In the commercial world, high-end *datacenters*(or *server farms*) have been heavily used by contemporary enterprises of all sizes to perform different types of business operations, such as web hosting, financial transaction processing, and database management. Compared with scientific computing, commercial applications have relatively irregular and asynchronous computing patterns. Therefore, although most datacenters also use clustered servers to achieve high performance, their components display more heterogeneity.

Besides the traditional model of executing the aforementioned business applications on local datacenters, the rapid emergence of *Cloud Computing* causes more large scale datacenters to be used. As of May 2009, Amazon's cloud computing system, called AWS [5], has a peak workload of 80,000 work requests per second, 52 billion objects stored and thousands of developers making use of it [6]. It even uses 30% more bandwidth than Amazon's e-commerce business. In Cloud Computing, customers' applications are virtualized into portable units, and executed on shared computing resources. While eliminating the necessity for small and medium businesses to maintain separate computing facilities, this trend shifts more responsibilities to large scale datacenters that are "on the clouds". We quote from a widely cited report by researchers at UC Berkeley about the importance of large scale datacenters in Cloud Computing [4]:

> *We argue that the construction and operation of extremely large-scale, commodity-computer datacenters at low-cost locations was the key necessary enabler of Cloud Computing, for they uncovered the factors of 5 to 7 decrease in cost of electricity, network bandwidth, operations, software, and hardware available at these very large economies of scale. These factors, combined with statistical multiplexing to increase utilization compared a private cloud, meant that cloud computing could offer services below the costs of a medium-sized datacenter and yet still make a good profit.*

1.2 Challenges in High-end Storage Systems

In the HEC facilities discussed in Section 1.1, the aggregate CPU processing power is growing rapidly with the increasing system size, especially considering the advances in processor technologies such as multi-core and many-core. It has been summarized that processor speeds have been doubling every 18 to 24 months [94]. Meanwhile, the performance of hard disk drives, which are main data storage devices in today's computer systems, is improving at a much slower rate. The disk bandwidth has been improving at about 40% per year due to increases in rotation speed and recording density, while the disk latency is improving at only about 10% per year [54]. Therefore, storage and I/O have been observed as major bottlenecks constraining the overall computing experience perceived by users [3, 4, 91, 114].

With the processor-I/O gap discussed above, it has become increasingly important to efficiently manage high-end storage systems. However, this is a challenging task due to many reasons, and we view the following as the key ones.

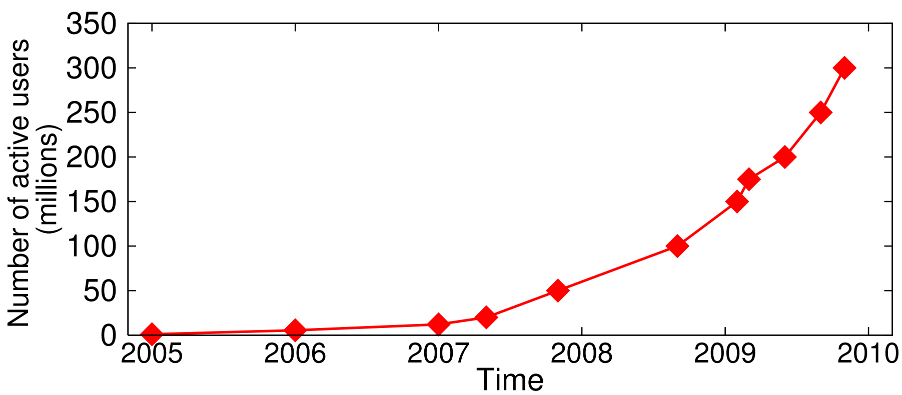


Figure 1.1: Growth of active Facebook users

Firstly, modern applications are becoming highly I/O-intensive, with large amounts of data collected and stored. It is reported that a total of 500 TB per year of imaging data is created in the UPMC medical system, and 200 of London's traffic cams are generating 8TB of data each day [43]. Online service companies, including search engines, multimedia hosts and social network sites, also process large numbers of customer requests, associated with massive amounts of data. Table 1.2 shows the online video usage data in the US in September of 2009, as collected by Nielsen[87]. It can be seen that within a month, all top 10 vendors deliver hundreds of thousands of video streams, while YouTube has over six million video streams and one hundred thousand unique viewers. Figure 1.1 further illustrates the rapid growth of online services by showing the number of active users of the social network site Facebook [37] as published in their company timeline [38]. Online customers also have similar temporal request patterns (except for the difference caused by time zones), which further increases the concurrency of data accesses. One significant challenge coming with heavy and highly concurrent I/O demands is the intensive competition for limited system resources, especially during peak times.

Table 1.1: US online video usage data of September 2009

Video brand	**Total streams**	**Unique viewers**
YouTube	6,688,367	106,180
Hulu	437,407	13,519
Yahoo!	228,494	30,084
MSN/WindowsLive/Bing	180,104	18,109
Fox Interactive Media	139,634	14,342
Nickelodeon Kids and Family Network	127,654	5,303
Turner Sports and Entertainment Digital Network	123,665	6,062
MTV Networks Music	116,839	9,647
ESPN Digital Network	114,652	9,299
Facebook	110,418	23,161

Secondly, growing with the volume of demands are the scales of high-end storage systems that handle them. Figure 1.2 illustrates the architecture of the Jaguar supercomputer. Within the dashed rectangle is the storage subsystem. As of early 2008, Jaguar has 72 Dell dual-core I/O servers with 174 Gigabytes of main memory. The underlying disk storage is provided by 18 Silicon Storage Appliance (S2A) 9550 couplets from Data Direct Network Inc. [35], with a total of 576 Terabytes of capacity. The current storage system, named the *Spider File System*, has been expanded to 192 dual-socket quad-core I/O servers with over 14 Teraflops in performance and 3 Terabytes of memory, as well as 48 S2A 9900 storage couplets providing 10 Petabytes of RAID 6 capacity from 13,440 1 Terabyte SATA drives. Many issues arise with the large numbers of storage components. For example, the number of failures in a system increases with a roughly linear rate with the number of components. It has therefore become a challenging task to provide high availability to stored data.

Finally, storage system components are also having more interactions with each other, in both vertical (across levels) and horizontal (across peers) dimensions. A piece of data typically travels through multiple levels of cache before reaching the application. Moreover, data striping is widely used, where a file is divided into chunks, and consecutive

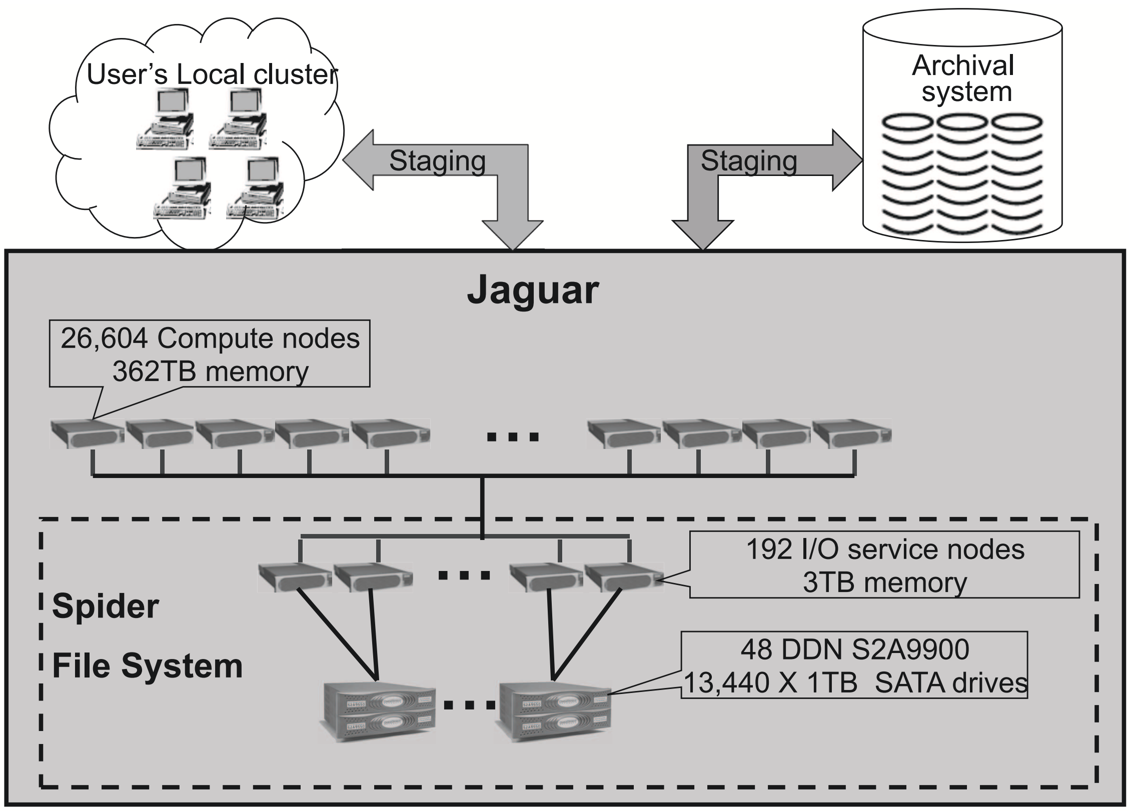


Figure 1.2: Storage architecture of Jaguar

blocks are placed on different storage components. In the Jaguar system shown in Figure 1.2, the Lustre parallel file system [26] is deployed on both the I/O servers and the compute nodes, which divides each file into 1MB chunks and stripes it over 4 storage nodes by default. Users of Jaguar also need to submit their computation jobs to job queues managed by the scheduler, and perform data transfers along the lifetime of jobs: the input data for a job is to be staged in from archival systems or users' local clusters before the job is dispatched for execution, and the result data is to be staged out after job completions. The impact of this factor is manyfold. Both resource allocation and fault tolerance become more challenging with the interactions and interdependencies among multiple components.

1.3 Motivation for Adding Coordination in System Management

From the issues stated in Section 1.2, it can be observed that the main challenge faced by high-end storage systems is not merely the lack of I/O resources relative to computing demands in an aggregate perspective. The difficulty also lies in the large and ever-increasing scale of the entire environment, and the emerging complexity of interactions and interdependencies among entities within the environment. Individually developed system components or algorithms, however, often behave with isolated local optimizations and fail to address the growing scale and complexity. Moreover, in handling concurrent workloads they often ignore inter-workload relationships, and lose opportunities of optimized global resource allocation. Therefore, we consider *adding coordination in system management* as an intuitive and fundamental solution to address the challenge faced by high-end storage systems.

The benefits of adding coordination in system management comes from many perspectives. Actually, this strategy has been successfully adopted in various areas in modern computing systems.

One of the most notable examples is data placement and rebalancing in parallel and distributed file systems, such as Lustre [26], Panasas [86, 123], and Google File System (GFS) [42]. When a file is created or expanded, the file system examines the entire pool of storage nodes and select a subset of them to host the new data. In this coordinated selection process, storage nodes are compared and ranked according to multiple criteria, including available capacity, current load level, failure probability and so forth. After placing of a piece of data on a storage node, the system will also adaptively rebalance the data layout due to events like node failures, new node joining, node storage device expansion and so forth. It has been pointed out in the above papers that, without adding the above coordination among storage nodes, certain nodes will soon become too "hot" and severely degrade the performance of the entire system.

Caching is a fundamental technique to address the processor-I/O gap mentioned above, by putting certain data blocks in the main memory based on prediction of future accesses, in order to reduce visits to the hard disk drive. Caching was originally done in a computer's local memory, typically for data stored on its locally attached disks. With the network latency and bandwidth approaching and surpassing those of hard disk, a number of cooperative caching mechanisms have been proposed, which coordinate the contents of local caches in a distributed system, so that one node's data request can be satisfied from another node's cache [33, 55, 102].

There has also been much work on adding coordination in fault tolerance techniques. In a recent paper [133], Zhong et al. propose calculating storage objects replication degrees according to its object request probabilities, object sizes, and space constraints. This mechanism maximizes the expected overall service availability at the price of weakening the protection for certain objects.

Despite the continuous research and development efforts on adding coordination to high-end storage systems, with the rapid development of computing and storage techniques we found that many important perspectives have been overlooked by previous studies. For instance, prefetching [51] is becoming increasingly important in today's file systems to improve sequential data access performance [94], but in contrast with caching, existing prefetching mechanisms often fail to coordinate among workloads and machines. In other words, they focus more on adjusting the prefetching behavior of a workload based on its own access patterns, and assuming the application, the cache and the disk are on the same machine [70, 74]. Another example is the lack of coordination between a supercomputer's storage subsystem and other components along a job's workflow, including the job scheduler, compute nodes, and other external systems connected by wide area network.

1.4 Our Contributions

Based on the above observations, the author's Ph.D. research focuses on three novel instances of adding coordination to the management of commercial and scientific high-end storage systems, which significantly improve the overall performance and reliability, as summarized below.

Coordinated, Rate-aware Multi-stream Prefetching As an important technique to hide disk I/O latency, prefetching has been widely studied, and dynamic adaptive prefetching techniques have been deployed in diverse storage environments. However, one significant issue not well addressed by previous research is how to handle the prefetching resource allocation between concurrent sequential access streams with different request rates.

Interestingly, we found that these problems bear a strong resemblance to situations long studied in the field of *supply chain management (SCM)*, used by retailers such as Wal-Mart. In this dissertation, we demonstrate how to perform the problem mapping and then apply SCM principles in practice, particularly from the branch of *inventory theory*, to improve data prefetching performance in storage systems. More specifically, we applied two SCM policies to capture data access rates of sequential streams and dynamically configure the sequential prefetching parameters.

We implemented these SCM-based strategies in the Linux kernel prefetching module, and evaluated the performance with three types of workloads. The results indicate that the SCM approaches are able to generate up to a 55.0% of performance improvement for a real-world server workload benchmark, and up to 33.3% for a combination of Linux I/O-intensive applications.

This work produced the majority of results of the paper [130]. A patch of the Linux 2.6.18 kernel was developed with new prefetching algorithms.

Coordinated, Hierarchy-aware Multi-level Prefetching The multi-level storage architecture has been widely adopted in servers and data centers. However, existing multi-level storage studies have focused mostly on cache replacement strategies, leaving multi-level prefetching a relatively unexplored topic.

In this dissertation, we show that prefetching algorithms designed for single-level systems may have their limitations magnified when applied to multi-level systems. Overly conservative prefetching will not be able to effectively use the lower-level cache space, while overly aggressive prefetching will be compounded across levels and generate large amounts of wasted prefetch. We consider information distortion as the main reason for this deficiency, and map it to the "bullwhip effect" in multi-echelon inventory management.

Based on this vision we take an innovative approach to this problem: rather than designing a new, multi-level prefetching algorithm, we develop coordination mechanisms which are compatible with existing single-level algorithms. We first present a simple information sharing scheme directly based on SCM policies. Furthermore, we develop PreFetching-Coordinator (PFC), a hierarchy-aware optimization applicable to any existing prefetching algorithms. PFC does not require any application hints, a priori knowledge on the application access pattern or the native prefetching algorithm, or modification to the I/O interface. Instead, it monitors the upper-level access patterns as well as the lower-level cache status, and dynamically adjusts the aggressiveness of the lower-level prefetching activities.

We evaluate the proposed mechanisms with an extensive simulation study using a verified multi-level storage simulator, an accurate disk simulator, and access traces with different access patterns. Our results indicate that simple information sharing can effectively improve the performance of certain pairs of single-level algorithms. Meanwhile, PFC dynamically controls lower-level prefetching in reaction to multiple system and workload parameters, improving the overall system performance in all 96 test cases. Working with four well-known existing prefetching algorithms adopted in real systems, PFC obtains an improvement of up to 35% to the average request response time, with an average improvement of 14.6% over all cases.

This work generated the paper [131] and part of [130]. We also developed a multi-level storage simulator as described above.

Coordinated Job Data Staging, Scheduling, Replication and Recovery Storage systems in supercomputers are a major reason for service interruptions. RAID solutions alone cannot provide sufficient protection as: 1) the growing average disk recovery time makes RAID groups increasingly vulnerable to additional disk failures during reconstruction, and 2) RAID does not help with higher-level faults such as I/O node failures.

In this dissertation we present a complementary framework to provide high data availability for supercomputing centers. Our work is based on the observation that files in the supercomputer scratch space are typically accessed by batch jobs, whose execution can be anticipated. Therefore, we propose an automatic scheme which stages in job data "just-in-time" for the job execution. We also transparently, selectively, and temporarily replicate "active" job input data by coordinating the parallel file system with the batch job scheduler. We further extend the parallel file system with the awareness of job data's remote sources and the ability to transparently fetch unavailable pieces in order to reconstruct corrupted files.

We have implemented the staging, replication and reconstruction schemes in the Moab job scheduler and the Lustre parallel file system. We also take data operation time tested on real clusters as inputs and evaluate the schemes with extensive simulation studies. Our results show that with our solution the average wait time for supercomputing jobs could be reduced by up to 80%, while the additional storage space overhead is always below 1.8%.

This work has generated three research papers [120, 119, 132], and a job scheduling software that is currently deployed at the Oak Ridge National Laboratory.

1.5 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 introduces the coordinated, rate-aware multi-stream prefetching schemes which appeared in [130]. Chapter 3 presents coordinated, hierarchy-aware multi-level prefetching, including the problem mapping to multi-echelon inventory control which appeared in [130] and a joint study with Kyuhyung Lee on the PFC framework which appeared in [131]. Chapter 4 presents the fault tolerance solution consisting of coordinated job data staging, scheduling, replication and recovery, which appeared in [120, 119, 132]. Finally, the dissertation is concluded in Chapter 5. For reader's convenience, related work is surveyed separately in Chapters 2∼4.

Chapter 2

Coordinated, Speed-aware Multi-stream Prefetching

2.1 Motivation and Contributions

Prefetching is an important technique widely used in computer systems. In storage systems, single- or multi-level alike, it reduces the visible disk I/O costs by exploiting spatial localities in applications' access patterns.

Despite the plethora of studies done on data prefetching for storage systems, certain fundamental issues have not been well addressed. In particular, currently storage systems face the increased scale in two dimensions.

First, with the advances in server configurations, especially due to the emergence of multi-core/many-core nodes, the level of request concurrency is expected to continue to rise for both server and personal computing workloads. Existing adaptive prefetching algorithms focus on identifying the sequentiality versus randomness in the application access pattern, as well as identifying individual sequential access streams. Such analysis helps manage space allocation between prefetching and demand paging, and allows per-stream prefetching aggressiveness control [45, 75].

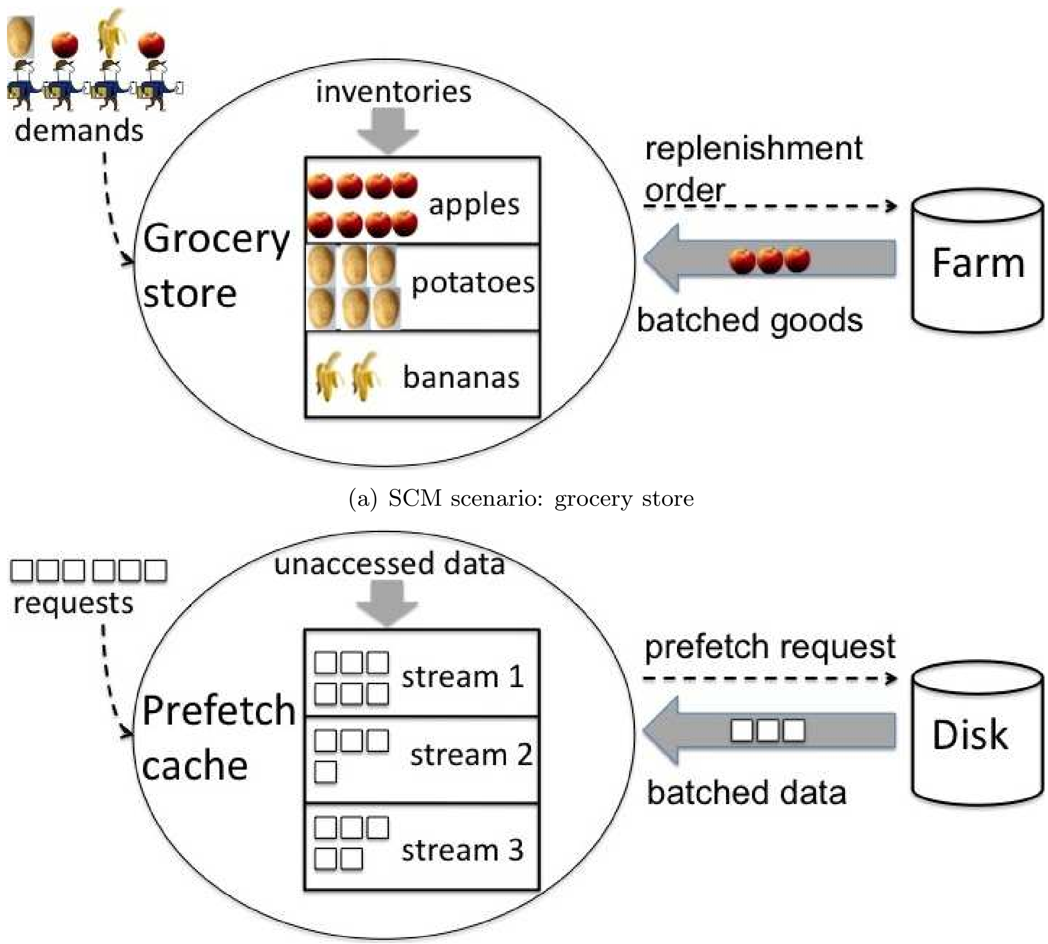
However, existing approaches seldom explicitly consider the application request rates, in particular the relative access speed of multiple sequential streams, in their runtime decision making for dynamic prefetching. This may cause the loss of important timing information crucial to effective prefetching and balanced resource allocation. In most cases, the same prefetching settings will be applied to slow and fast streams, as long as they display the same sequentiality in accesses. Fast streams may not be able to have sufficient prefetched blocks to catch up with the application's pace, while slow streams may have many blocks unconsumed sitting in the buffer cache. To make it worse, the faster streams often have shorter computation periods to overlap with the I/O requests even when the program does I/O asynchronously, making the penalty directly visible to end users.

Second, the I/O stack is becoming increasingly deep with multiple layers of servers and their buffer caches, including database buffer caches, file system caches, storage server caches, etc. In multi-level storage systems, the caching/prefetching strategies working at an upper level may alter the application access behavior and mislead lower levels to prefetch too aggressively or conservatively [128, 131].

Interestingly, we discovered that the above unaddressed data prefetching issues share many common properties with another field of study with a history of over half a century - *supply chain management (SCM)*. This is illustrated in Figure 2.1, where we see the one-to-one correspondence between SCM and dynamic data prefetching problems. Basically, in both contexts, items are retrieved to better serve *anticipated* customer requests, based on observations on the past and current request behavior. Correct and timely fetching of the predicted items will greatly enhance the customer perceived system performance, while mistakes (such as stocking up the wrong items or performing the fetching too eagerly/leisurely) can waste precious resources in terms of storage/inventory and data/goods transportation capacity.

In this chapter, we present a fresh approach to improving prefetching: borrowing and extending ideas from SCM theory and practice to enhance data prefetching algorithms deployed in modern storage systems. Our major contributions are summarized below:

1. We identified the mapping between data prefetching and supply chain management problems, and proposed the novel application of SCM principles to controlling adaptive prefetching parameters.
2. We performed dynamic prefetching by explicitly measuring level and deviation of per-stream access rates and conducting resource allocation accordingly, which to our best knowledge have not been done previously. Rate-aware resource allocation has been applied in other subfields of computer systems research, such as the proportional share scheduling in real time systems [59]. However, those existing techniques can not be



(a) SCM scenario: grocery store

(b) Storage system scenario: buffer cache

Figure 2.1: Similarities in SCM and storage system problems

directly applied in the prefetching context due to significant differences in the problem spaces.

3. We implemented the SCM-based algorithms in the Linux kernel, as well as a verified multi-level storage server simulator, and evaluated their performance with Linux I/O intensive applications and real-world server workload traces. Our results indicate that SCM solutions generate considerable performance improvements over existing prefetching algorithms.

The rest of this chapter is organized as follows. Section 2.2 provides background discussions in both SCM and prefetching. Section 2.4 presents the mapping of a prefetching problem in modern storage systems to an SCM problem, so that SCM solutions can be applied. In Section 2.5 we discuss how to use SCM policies to data prefetching. Experimental methods and results are presented in Section 2.6. Finally, Section 2.7 provides a brief conclusion and discussions about future directions.

2.2 Inventory Theory Overview

In this section, we give an overview on relevant studies in supply chain management.

A *supply chain* is consists of “all the parties involved, directly or indirectly, in fulfilling a customer request” [24]. In particular, SCM research is still very active today, partly due to the challenges and opportunities brought by the advances in IT technology and supply chain globalization.

While many researchers have been renovating SCM for the web era, in this research we exploit technology transfer in the other direction. The specific SCM area of interest is *inventory theory*, which studies how to efficiently manage the inventory for goods and satisfy customer demands with low costs, by carefully distributing limited resources.

Let us give a brief introduction on inventory theory.[1] Figure 2.2 illustrates the periodic fluctuation in the inventory level for each type of goods in the grocery store example. The items are consumed by customers, so the inventory level will decrease. When the inventory level drops to the *reorder point*, another batch of items will be ordered, which will arrive

[1] Throughout this thesis, we adopt SCM terms and definitions from two widely used supply chain management textbooks [24, 106].

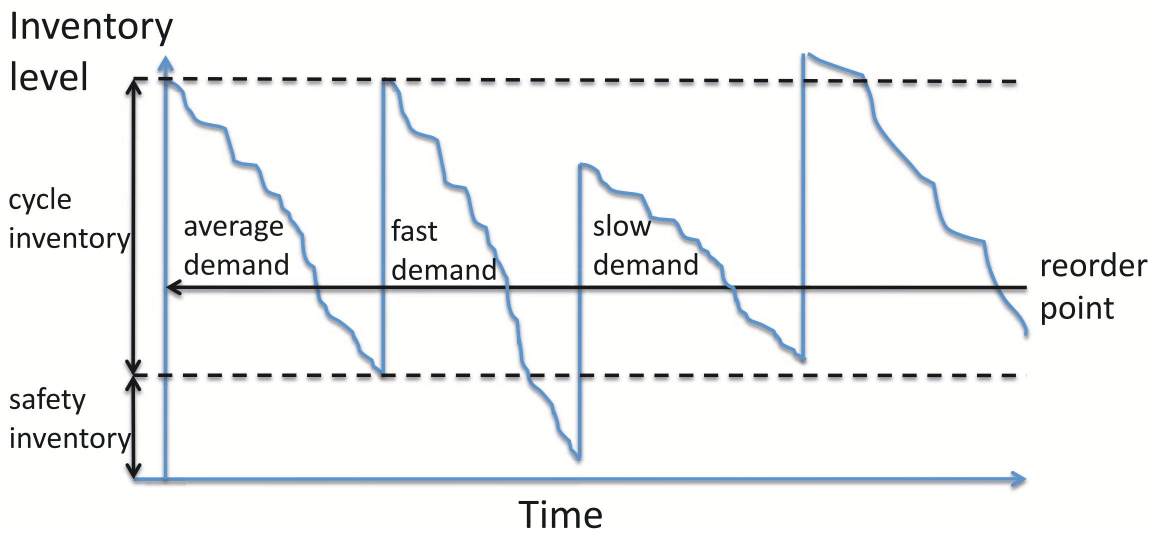


Figure 2.2: Sample inventory level fluctuation and associated terms

after a certain amount of time, and boosts the inventory level. Much inventory research effort has been spent on configuring the two components of the desired inventory level: the *cycle inventory*, used to handle stable customer requests and achieve economic/efficient ordering and transportation, and the *safety inventory*, used to prepare for irregularity or burstiness in customer demands. Intuitively, if the customer demand for this type of goods is constant, then the safety inventory can be lowered to zero.

Similar to SCM and inventory theory, another mathematical tool originally developed for business management is *queuing theory* [52]. Over the past decades, queueing theory has been applied in various fields in computer science with great success [64], especially in telecommunication, operating system scheduling and computer networks. While queueing models fit well with problems where multiple clients share certain critical resources (data channels or CPUs) in a time-splitting manner, inventory theory is able to model problems where a certain amount of space (such as CPU cache or memory) is the shared resource. In addition, the ordering operations in a supply chain and the prefetching operations in a storage system are system-level enhancements selectively placed on a subset of incoming requests, where queuing models do not fit as well. We believe that by following the successful example of queueing theory, SCM and inventory theory may also potentially become powerful tools in computer science research.

In Section 2.4, we describe detailed mapping of our target data prefetching problem to this inventory theory context.

2.3 Related Work

The problem of data prefetching has been widely studied [115]. In this section we present a survey of related work on this topic.

Access pattern detection Many sophisticated prefetching algorithms have been proposed to prefetch data blocks in a non-sequential manner (e.g., with strided patterns) based on access pattern analysis and mining [7, 32, 40, 70, 74, 108]. However, most modern storage systems adopt *sequential prefetching* [44, 45, 107], which prefetches a number of blocks right next to the blocks requested, due to its simplicity and applicability to a wide range of application workloads [44]. Therefore, the discussion in this chapter is based on sequential prefetching, although our proposed approaches are portable to prefetching algorithms that establish correlations between data blocks.

Prefetching aggressiveness control For sequential prefetching, "what to prefetch" is given and the key problem is to decide "how much to prefetch" and "when to prefetch", two metrics related to prefetching aggressiveness. In many systems today, this is determined at runtime through two prefetching parameters: *prefetch degree*, to control how much data to prefetch for each prefetching request, and *trigger distance*, to control how early to issue the next prefetch request. While there have been a number of studies on intelligent dynamic prefetching, by manipulating the trigger distance and/or the prefetch degree [44, 73], they focus more on identifying the sequentiality in accesses and do not measure or exploit the application request rate information.

One specific example of dynamic sequential prefetching algorithm used widely in systems today is the Linux kernel prefetching algorithm [14], which we improved in this project using our SCM-based optimizations. This algorithm performs per-file access pattern analysis to determine whether a file is currently accessed sequentially, and doubles the prefetch degree when continuous sequential accesses are observed. With the 2.6 kernels, the prefetch degree and trigger distance are always set to equal values, with a default upper limit of 32 blocks. If the streak of sequential accesses is interrupted, the Linux algorithm will scale back the prefetching degree (by default to 3 blocks).

Another important related issue is how to allocate the shared memory cache space between prefetching and demand paging, both of which hide the I/O latency. There are a number of previous studies about managing prefetched data in a cache shared with demand paged data [18, 63, 72, 93, 96]. Several other solutions alleviate the problem of *cache pollution* [92, 100] by carefully bounding the space that prefetched data can use. As a recent example, the SARC algorithm deployed in IBM production systems [45] uses two separate LRU queues for sequential and random data respectively, and adjusts their sizes according to the access pattern. Our proposed approach can be used in a complementary way, partitioning resource between prefetching and demand paging through one of the above strategies, then among concurrent sequential access streams.

Resource allocation among multiple streams There has been a number of projects on caching/prefetching for workloads with multiple concurrent access streams [9, 17, 44, 73, 75, 112]. With two of them, namely LRU-SP [17] and AMP [44], although access rates are not measured explicitly, fast and slow streams will be treated differently through eviction and detection of request waiting events. However, LRU-SP is designed for cache replacement and does not change the prefetching behavior. The underlying model for AMP does consider the per-stream request rate, while its algorithm design and evaluation focus on throttling the overall prefetching aggressiveness without distinguishing between fast and slow streams. The TIPTOE algorithm [112] estimates computation time between I/O requests. But it does so by relying on application hints, and at most two concurrent applications are used in the simulation-based evaluations. Our approach, in contrast, explicitly performs per-stream resource allocation based on level and deviation of access rates observed at runtime.

2.4 Problem Mapping

To apply SCM insights and techniques to data prefetching, we first need to map a prefetching problem in modern storage systems to an SCM problem. As mentioned in Section 2.1, we found striking similarities between the data prefetching and SCM scenarios, which allow us to construct the problem mapping. At the same time, there are challenging differences that prevent straight-forward or accurate mapping. These similarities and differences are discussed next, where we go through the key pairs of concepts.

2.4.1 Requests → Demand

The applications' requests for data, mapped to blocks by the file systems, can be intuitively translated into customers' demands for goods in an SCM context. Modern file and storage systems use dynamic algorithms that adjust the prefetching aggressiveness based on access pattern analysis. Similarly, in SCM, much effort has been spent on understanding and forecasting customer demands, where the demand patterns for a certain item are often modeled as time series of separate values. Advanced time series analysis techniques have been used to capture different characteristics of demand patterns [106]. It is also natural for us to map each sequential access stream (*i.e.*, blocks requested by one client in a server system) to a type of goods in the SCM context, as shown in Figure 2.1. In both areas, access/demand pattern analysis and subsequent prefetching have been performed on per-stream or per-type basis.

The normal distribution and other distributions from the exponential family [65] are the mostly commonly observed customer demand patterns in SCM [24]. Most SCM techniques, including the algorithms to be discussed in section 2.5, are based on such demand models. In our analysis of real read-intensive applications' I/O traces, we have found that their data access patterns share several important properties with the exponential family distributions, while at the same time possessing unique characteristics. Figure 2.3 shows the data access pattern of a representative stream from an SPC[2] OLTP trace. We partitioned the access trace into many 10ms time windows. The x axis lists different demand rates (in terms of KBs of data requested in a window), while the height of the vertical bars illustrates the count of windows within the corresponding rate interval. In the figure we also show a fitting gamma distribution pattern. We can see that while the two patterns fit relatively well at and after gamma's peak value at 7, the stream from the trace shows rather irregular patterns in the region with lower rates.

Fortunately, the medium and higher access rates are of more interest to most prefetching algorithms, since modern storage servers typically have high expected cache hit rates (around or above 50%) [23, 134], especially for sequential access patterns. More

[2]http://traces.cs.umass.edu/index.php/Storage/Storage

comprehensive and detailed analysis of real world applications' data access patterns will potentially lead to enhanced algorithms and we consider it interesting future work.

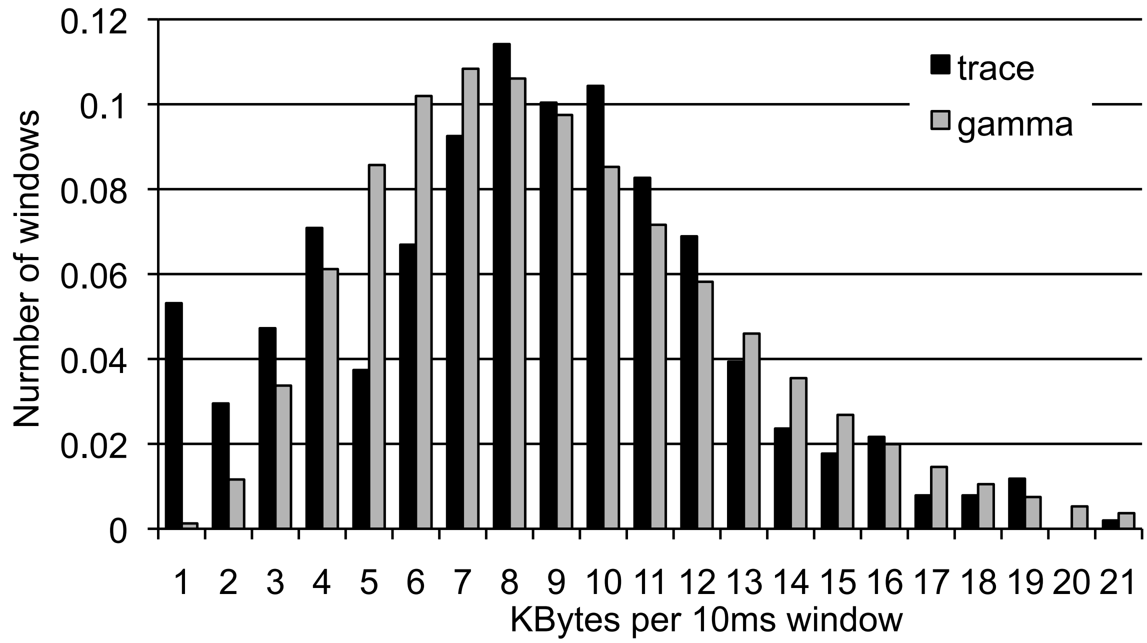


Figure 2.3: Distribution of application requested data per 10 mili-seconds

However, there is a fundamental challenge in mapping data requests to customer demands. In the SCM model, there are typically duplicate items of the same good. The customer demand specifies the type and quantity of goods needed (*e.g.*, "six apples"). In the data storage scenario, though, each data block is unique. However, when studying sequential prefetching we can achieve the mapping: assuming that correlations between data blocks have been correctly detected, any incoming data request can only refer to one possible block, which is "next to" the last accessed block in the detected logical correlation. Therefore, the unique data blocks belonging to the same access stream can be viewed as a sequence of "the next block", hence becoming a group of identical items.

2.4.2 Access Time → Lead Time

n both storage and SCM scenarios, we need to consider the time to retrieve items, both to assess the potential benefit in hiding this latency, and to determine how early we should perform the fetch. In storage systems, this is the time to retrieve requested data from the disk, or a lower-level server in multi-layer architectures. Its SCM counterpart is the *lead time*, the time that elapses from the moment that an order is placed, till the requested items are physically on the shelf.

In general, we found the data access time in storage systems possess different characteristics than the lead time modeled in SCM studies. The data access cost depends on (1) the location (in terms of placement on disk) and the amount of data requested, and (2) the system load and disk head location. In contrast, SCM models typically assume the lead time as constant or i.i.d. (independent and identically-distributed) over time, insensitive to the demand volume or supplier workload. While bridging these differences for more accurate problem mapping makes interesting future work, in this chapter, we report initial results achieved without extension of existing SCM theories to accommodate more sophisticated lead time modeling.

2.4.3 Batched Prefetching → Batched Replenishment

Contemporary file system prefetching algorithms fetch blocks in batches, rather than one at a time. Doing so takes advantage of the spatial locality in disk accesses, lowers the disk/network request overhead, and reduces energy consumption. In particular, due to the high start latency of disk accesses, a certain request size may be required to match the request rate. In SCM, intuitively, the same practice is adopted so that supply chain entities issue *batched replenishments* to lower entities or the factory. The motivation here is to exploit the *economics of scale*, examples of which include quantity discounts, sharing of per order costs, etc.

2.4.4 Prefetched Items → Inventory

Prefetching is about stocking up data blocks that are predicted to be soon accessed in the future. Many prefetching algorithms control the size of this set of fetched but not accessed data by manipulating the trigger distance, which dictates how early the prefetch request should be issued. In SCM models, the set of prefetched goods corresponds to an extensively studied concept of *inventory*, the list of goods held available in stock.

We discovered that prefetching issues not studied explicitly in storage research have been discussed in inventory theory. More specifically, existing SCM literature provides

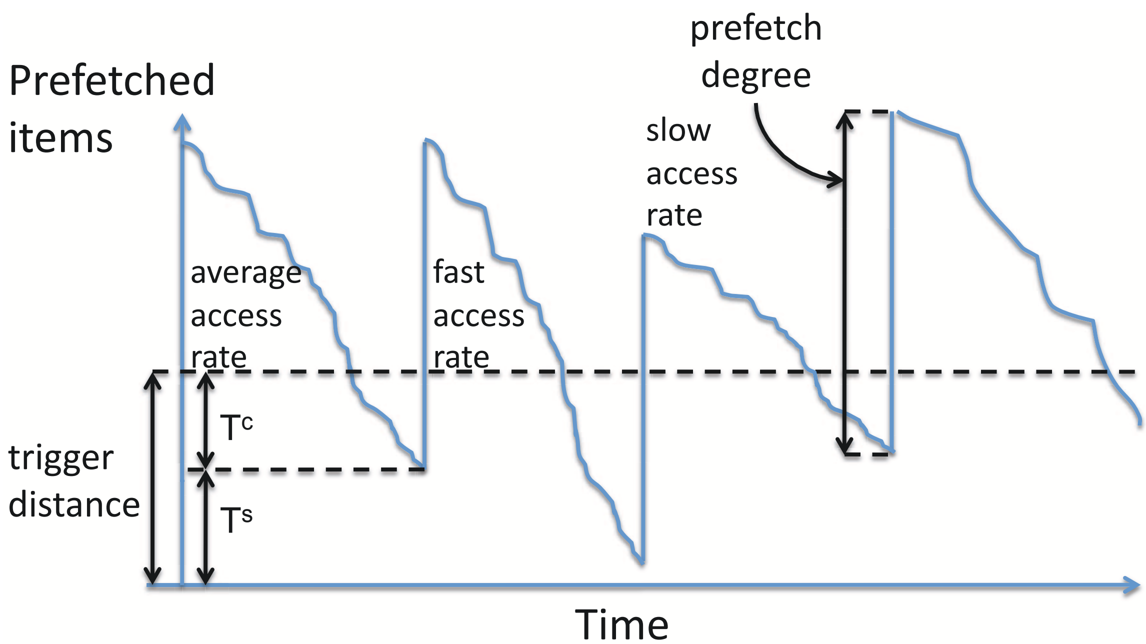


Figure 2.4: Inventory behavior in the prefetching context

insights on differentiating the functionality of the prefetched items in stock, into cycle inventory and safety inventory, as introduced in Section 2.2. We believe that the trigger distance is closely associated with the inventory notion and can be similarly decomposed.

Figure 2.4 shows how the dynamic prefetching parameters can be mapped to the inventory problem. Here the trigger distance, T, is equivalent to the reorder point in Figure 2.2. After each time the prefetch is triggered, the number of prefetched items unconsumed is expected to drop by a certain amount before the next batch of prefetched data arrives. This amount is determined by the average application request rate, as well as the fetch time (lead time) for this prefetch operation. It then corresponds to the cycle inventory and we denote this component of T as T^C. The other component, $T^S = T - T^C$, corresponds to the safety inventory. The prefetching degree can be intuitively mapped to the size of the batched replenishment. In Section 2.5, we describe how we leverage SCM policies to determine these prefetching parameters with multi-stream workloads.

One may argue that the prefetched items behave differently with grocery items in an inventory, in the sense that accessed data blocks still remain in the memory cache until replaced, while items bought by a customer will be removed from the inventory. Again, our mapping is achieved within the sequential prefetching context. When each strictly sequential stream is mapped to a type of goods, then after a prefetched block is accessed, this item can be viewed as "consumed" and will be taken off the inventory.

2.5 SCM Algorithms for Data Prefetching

As mentioned earlier, existing prefetching algorithms, while identifying the concurrent sequential access streams in mixed or multi-request workloads, do not explicitly consider the individual access rates of these streams. With SCM practices, it is intuitive that if apples and bananas are consumed at very different speeds, it does not make sense to stock up the equal amount of them in store. The counterparts in replenishment policy for trigger distance and prefetch degree are *order point* and *order quantity*. A fixed quantity Q is ordered whenever the inventory level drops under the reorder point s. Inventory theory determines the reorder point s according to the rate of and the uncertainty in the incoming demands for a certain item. Next, we discuss the SCM optimization goal and introduce two commonly used policies for determining reorder points. Although there are more advanced SCM techniques, we choose these two relatively simple policies. This is due to their potential of being adopted in widely used operating systems such as Linux, considering several factors, including runtime overhead, implementation complexity, and capacity to work for a large range of applications. We then apply them to dynamically configuring the prefetching trigger distance. In particular, we focus on the problem of concurrent, long sequential access streams with different access speeds. Such scenarios are common in personal computing, web servers, multimedia servers, and scientific data centers.

2.5.1 Performance Metrics and Objective

In the prefetching and caching environment, the cache hit ratio is one of the most important factors contributing to the overall I/O performance, measured either in request response time or throughput.

In SCM the corresponding concept is *fill rate*, which is defined to be the fraction of demand that is satisfied from product in inventory [24]. Given the customer demand distribution and the replenishment policy, the fill rate, FR, can be estimated as:

$$FR = 1 - \frac{ESC}{Q}$$

In the above ESC denotes *Expected Shortage per Cycle* (the average number of unsatisfied customer demands in each replenishment cycle), and Q is the order quantity (prefetching degree in the storage context). SCM studies have shown that the optimization of FR is a very challenging problem [106], and two heuristics policies have been widely adopted, as discussed in the following sections.

Notice that the ultimate objective of the policies is to optimize the overall system performance, a common goal in file and storage system research, especially with adaptive techniques [72]. The policies can be extended in the future to consider other objectives or constraints, such as fairness across streams and real time requirements. Further, our current policies mainly focus on improving the performance of pure sequential access patterns, which are common and important in today's servers and data centers, running applications such as video-on-demand (VOD) and scientific data visualization.

2.5.2 Equal Time Supplies

The first policy, *Equal Time Supplies (ETS)*, is an intuitive, widely used approach: A large U.S.-based international consulting firm estimates that 80% $\sim$ 90% of its clients use this policy for setting safety inventory levels [106]. It is applied after the total amount of safety inventory for an SCM entity has been determined, e.g., by a grocery store's physical storage space. Given this total inventory level, with ETS, the individual safety inventories are set so that each type of goods has an equal time period's worth of supply. In other words, the safety inventory level for individual types of goods is set proportional to their demand rates.

The same observation can be applied to application data request rates. The "total inventory level" in this case, corresponds to the overall prefetching aggressiveness. There are existing algorithms, such as SARC [45], that adjust the cache capacity and bandwidth resources between sequential accesses (prefetching) and random accesses (demand paging), according to runtime access pattern analysis. They can be leveraged to determine the total inventory level.

With the given total inventory level set, our prefetching algorithm will determine the desired trigger distance for each sequential stream. Its goal is to maintain a certain overall prefetching aggressiveness, in terms of T_{total}, the sum of the trigger distances for all concurrent sequential streams. Suppose at a given time, there are n concurrent access streams in the system, each with an access rate of R_i. Based on the ETS policy, the safety inventory component of the trigger distance, T_S, is proportional to the per-stream access rate:

$$\forall i, \frac{T_i^S}{R_i} = C_1$$

As can be seen in Figure 2.4, the cycle inventory component, T_C, is determined by how much of the prefetched items will be consumed during the time to complete the next prefetch order. As most adaptive prefetching algorithms have a fixed upper limit for the batch size (prefetching degree), all pure sequential streams will be able to reach this upper limit after a short ramp-up stage. So the time to prefetch each batch of data blocks can be assumed to be equal across all streams. Therefore, the expected amount of data to be requested while the batched prefetch request is in-flight, is then proportional to the data request rate as well:

$$\forall i, \frac{T_i^C}{R_i} = C_2$$

Consequently, the individual trigger distance should be proportional to the request rate according to the ETS policy. We can calculate the ETS-based trigger distance T_i as

$$T_i = T_i^S + T_i^C = \frac{R_i}{\sum\limits_{1 \le i \le n} R_i} \cdot T_{total}$$

2.5.3 Equal Safety Factors

As discovered by SCM researchers, one significant limitation of the ETS policy is that it does not consider the difference in the demand uncertainties of the streams. It works best when the customer demand for each type of good remains stable, which is often not the case in SCM or storage scenarios.

The *Equal Safety Factors (ESF)* policy addresses this limitation by setting the safety inventory level for each type of goods proportional to the standard deviations of their requests. The intuition is that those streams with the highest uncertainty in demands should receive a higher safety inventory level. With this approach, we first calculate the cycle inventory component of the per-stream trigger distance explicitly using the stream access rates. Suppose the time to fetch a batch of blocks using the maximum prefetch degree is $t_{prefetch}$, we have

$$T_i^C = R_i \cdot t_{prefetch}$$

Then the per-stream safety inventory component is allocated from the total safety inventory level proportionally, considering the standard deviation of access rate rather than the average rate. Suppose the standard deviation in access rate for stream i is σ_i, we have

$$T_i^S = \frac{\sigma_i}{\sum\limits_{1 \leq i \leq n} \sigma_i} \cdot (T_{total} - \sum_{1 \leq i \leq n} T_i^S)$$

Finally, for each stream i, the trigger distance is again

$$T_i = T_i^S + T_i^C$$

2.5.4 Implementing ETS/ESF in the Linux Kernel

We implemented the runtime trigger distance configuration algorithms described in Section 2.5.2 and 2.5.3 in the Linux 2.6.18 kernel. As mentioned earlier, in Linux 2.6 kernels the default value for both the prefetch degree and the trigger distance is 32 pages. Since we assume that the overall inventory level for prefetching is determined using other algorithms and is beyond this scope of this research, in our implementation we try to maintain the same amount of memory consumption as the baseline Linux prefetching algorithm. In Section 2.6, we show that by constraining the prefetching memory use, our algorithms have a rather small impact on concurrent random accesses though they target sequential streams.

With a trigger distance of T and a prefetching degree of P, the average per-stream inventory level (the total number of unconsumed prefetched blocks) is $((T + (T + P))/2 = T + P/2$. Therefore increasing the trigger distance by one is twice as expensive as doing the same with the prefetching degree, in terms of memory space. For our ETS/ESF algorithms, we start with a setting where $T = P/2$, for a balanced allocation between the two parameters (which is also adopted by another existing algorithm [44]). To maintain the same average inventory level as the Linux prefetching default parameters, we get

$$T + P/2 = P/2 + P/2 = 32 + 32/2 = 48$$

This gives us the default ETS/ESF trigger distance, $T_{default}$, of 24, and the maximum prefetching degree of 48. A lower limit for trigger distances is set to 4. In our evaluation, we compared our SCM-based strategies with both the original Linux prefetching algorithm (32-32) and the algorithm with modified parameters (24-48).

More details about the implementation of ETS and ESF algorithms in the Linux 2.6.18 kernel can be found in [67].

2.6 Performance Evaluation

We implemented our SCM-based optimizations in the Linux 2.6.18 kernel (see Section 2.3) and a multi-level storage simulator (described below in Section 2.6.1). In the rest of the section, we first give an overview of our test platforms and workloads, then report experiment results.

2.6.1 Platform and Workload Overview

Platforms

Platforms We tested our modified Linux kernel for single-level prefetching on a Dell PowerEdge 2950 III server, with two 2.33GHz quad-core Intel Xeon E5410 processors, 16GB main memory, and integrated RAID 5 with six 146 GB 15K RPM SAS 3Gbps hard drives.

Workloads We evaluate our proposed approach with three types of workloads:

1. Synthetic benchmarks. These benchmarks are created to include a group of sequential access streams, where we can easily control the per-stream access rate and rate deviation, as well as the number of concurrent streams.

2. Real Linux file transfer applications (*cp* and *scp*). For *cp* we use both local and remote (NFS) destinations. For *scp* destinations we use workstations within the same campus of our server, as well as a set of geographically dispersed sites from the PlanetLab [25].

3. Two types of server benchmarks. The first is an HTTP web server workload. At the server side we run Apache on our Dell server with modified Linux kernel. On the client side we use httperf-0.9.0 [84] to generate 20 clients (again running on PlanetLab nodes) that request file downloads with different bandwidth. The second is a SPC2-VOD-like benchmark that we implemented according to SPC specifications [3] to simulate video-on-demand workloads. In SPC2-VOD, each sequential stream has a fixed think time between consecutive I/O requests. To assess our algorithms' impact on concurrent random accesses, we also run a TPC-H trace collected on workstations at Purdue University [14], which has highly random accesses.

Unless otherwise noted, all experiments are repeated 7 times and for each result (from our proposed approaches or baseline algorithms for comparison) we report the average after eliminating the best 2 and worst 2 results. This allows us to filter out "outliers" caused by other concurrent system activities (periodic system daemons, login attempts, etc.), which are observed to happen in similar ways when we switch between prefetching algorithms.

2.6.2 Concurrent Streams with Varied Access Rates

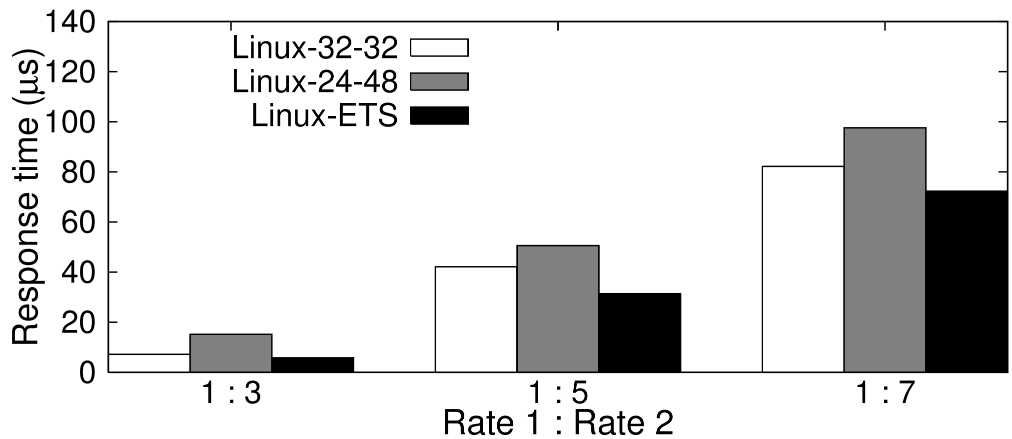
Synthetic Benchmarks

As our research in prefetching for concurrent streams focus on sequential streams, we first use synthetic workloads to examine the behavior of the original and SCM-based Linux prefetching algorithm with controllable stream access characteristics. In the synthetic benchmarks used in this section, we create multiple concurrent streams which issue pure sequential 4KB read requests, each accessing a different file of 1GB size (which is larger than any per-stream access footprint). The streams are fully overlapped with each other, starting together and running for the same length of time.

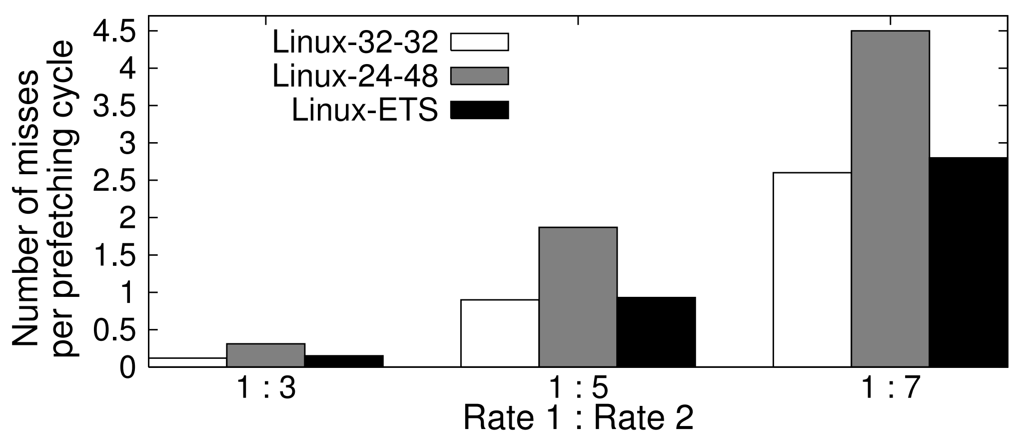
In these experiments, the system response time under four different policies are evaluated: "Linux 32-32" for original Linux kernel prefetching algorithm with both prefetch degree and trigger distance being 32 pages, "Linux 24-48" for the policy which increases prefetch degree to 48 and decreases trigger distance to 24, plus the ETS and ESF algorithms.

First, to evaluate how the ETS algorithm responds to different levels of access rate contrasts between streams, we generate a set of traces, each containing two streams

[3] http://www.storageperformance.org/specs



(a) Average per-request response time



(b) Average number of misses per prefetch cycle

Figure 2.5: Two streams with growing rate differences

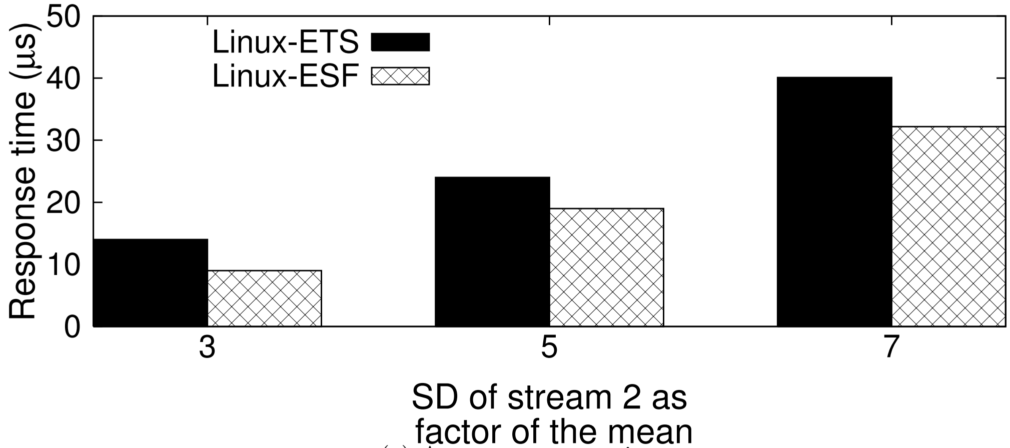
with a pre-specified rate difference. The rate of the first stream is fixed at 1000 requests/s, while that of the second stream is set to 3000, 5000 and 7000 requests/s in 3 experiments respectively. Figure 2.5(a) shows the results in terms of the average request response time (for all requests in both streams). Figure 2.5(b) portraits the average number of cache misses per prefetching cycle (ESC).

With 24-48, the total number of prefetching cycles is decreased by a third from 32-32, by enlarging the prefetching degree limit from 32 to 48. However, the overall response time is worse due to the disproportionally higher number of misses per cycle, due to the shorter and uniformly set trigger distance. Although ETS has the same number of prefetching cycles as 24-48, it efficiently allocates the total amount of trigger distance between the two streams. Thus it is able to control the number of misses per cycle to be very close to 32-32 and improves the request response time by 19.0% on average, with a maximum enhancement of 25.4%. The trigger distances used by ETS for the two streams (averaged over all prefetching cycles) are (13,35), (8,40), and (7,41), for the three different rate combinations respectively.

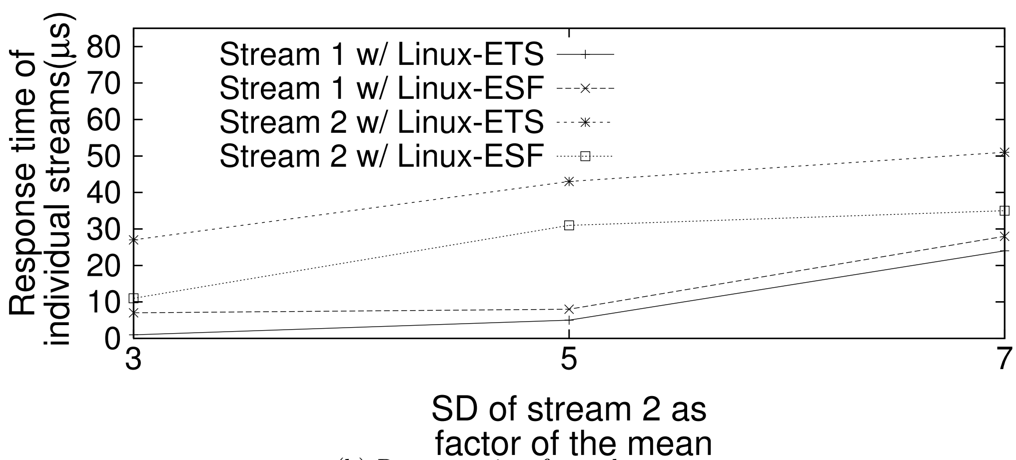
Next, we assess the effectiveness of the ESF algorithm by comparing it with ETS. with a different trace setting. Here again we have two streams, with the same mean access rates but different rate variances. The standard deviation of the first stream is fixed at the square root of the mean rate, while for the second stream it is varied from the 3 times to 5 times and then to 7 times the mean value. Figure 2.6(a) shows the average per-request response time, while Figure 2.6(b) shows the response time of the two streams separately.

As expected, ESF does perform better than ETS with less stable access streams, winning by an average of 25.4% and up to 35.7%, due to its capability of adjusting the safety inventory level according to the observed variance in request rate. As can be seen from Figure 2.6(b), the unstable stream obtains a significant performance improvement under ESF, while the more stable one suffers small degradation.

However, in the rest of our results using real applications and server workloads, where the access rate behavior is less regular, we found that ESF is consistently outperformed by ETS. We believe the reason is related to our measurement of the rate deviation, which currently uses a simple sliding window average method. In our experiments we observe significant fluctuations of the measured deviation values. This will cause unstable trigger distance setting, hurt the performance, and possibly inject access rate variances.



(a) Average response time



(b) Response time for each stream

Figure 2.6: Two streams with growing rate variances

Real Linux Applications

We then evaluate both ETS and ESF using a combination of local and remote Linux file transfer applications, each transferring multiple 128MB files. Among them, *cp-local* copies to a local file system, *cp-NFS* copies to an on-campus machine via NFS, and *scp1*∼*scp6* copy to different PlanetLab nodes (with decreasing bandwidth). Here we measure the I/O throughput of individual streams as well as the aggregate system throughput, during a fixed time period where the concurrent streams fully overlap. This experimental environment mirrors a file system in a high performance computing center where users frequently need to transfer job input/output files between the local and remote sites.

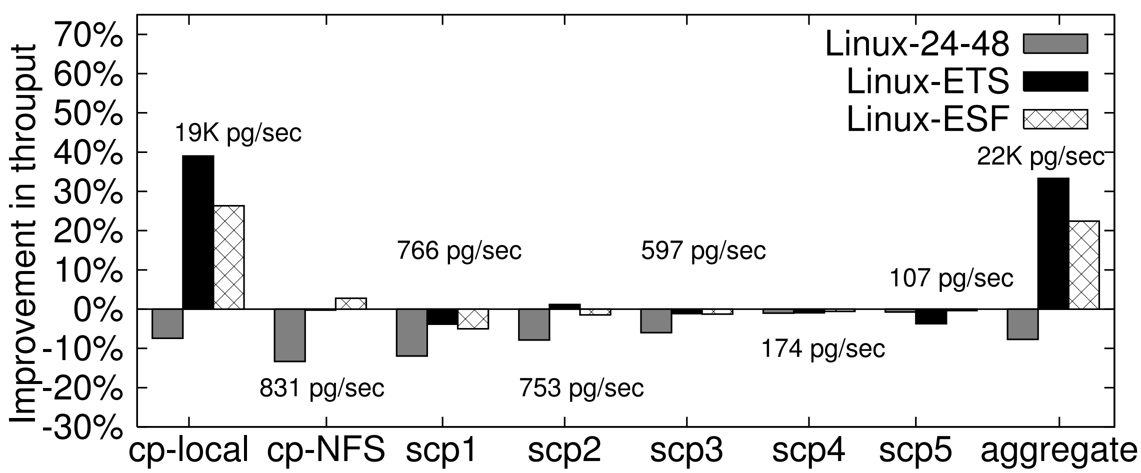


Figure 2.7: Throughput of file transfer applications

Figure 2.7 shows the improvement of three algorithms over the native 32-32 algorithm, whose per-stream and aggregate throughput numbers are marked above the bars. The aggregate throughput is improved by 33.3% with ETS and 22.4% with ESF, mainly due to the large performance gain for the fast *cp-local* stream (39.0% with ETS and 26.3% with ESF). Meanwhile, for streams *cp-NFS*∼*scp6* the SCM algorithms do not have significant impact (within +/-5%). The much smaller trigger distances assigned to them by the SCM algorithms (close to lower limit) do not cause poor performance because in most cases they can satisfy the low access rates.

HTTP Web Server

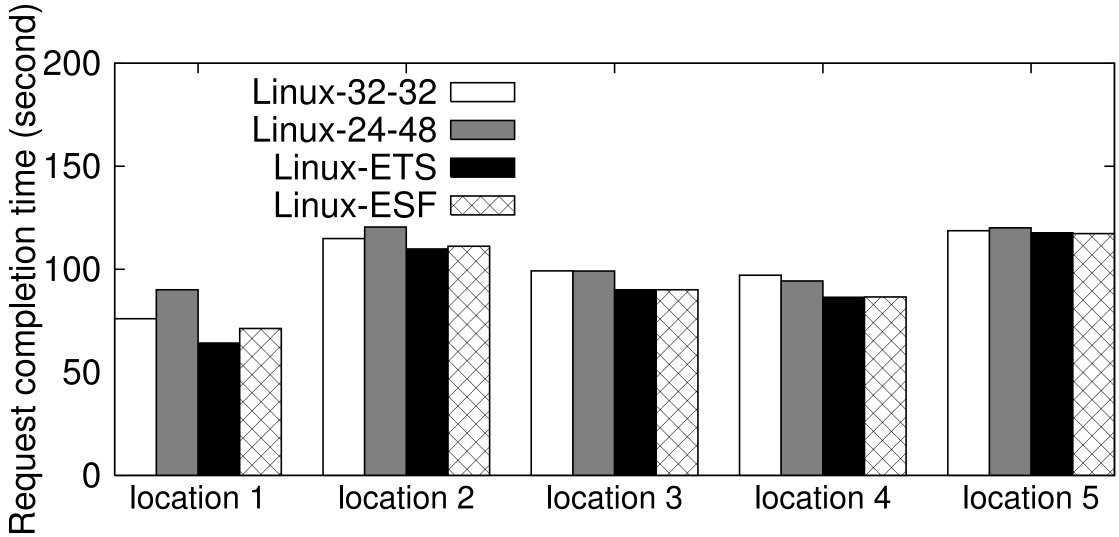


Figure 2.8: Request completion time for web clients

In this section we perform experiments on a HTTP web server handling file download requests from multiple clients. Our web clients are generated by httperf [84], a web workload generator developed by HP labs and used by previous system research work [1, 53]. Httperf is able to create a realistic environment where multiple concurrent clients connect to the server through the Internet and issue HTTP requests with specified patterns.

We use five PlanetLab nodes at different geographical locations, with decreasing connection bandwidth to the server. On each node we start four file download requests using httperf. Instead of system throughput, in this set of experiments we calculate the average request completion time for clients on each location, which is measured by httperf and is a more natural metric from a web client's point of view. To make the download requests overlap with each other, we let faster clients download larger files, so that all download requests have close execution time. This is a realistic scenario since web servers often recommend files with higher qualities to faster clients.

Figure 2.8 shows that the completion time of the fastest location is reduced by around 18% compared with the native 32-32 algorithm. The completion times for the slower clients are also reduced by 1% $\sim$ 12% despite the shorter trigger distances assigned to them. This is because the earlier completion of the fast requests allow them to eventually work with more resources. The relative performance of ESF compared with ETS is similar as in section 2.6.2. For workloads where the access rates do not vary rapidly over time, ESF's does not have an advantage in resource appropriation. On the other hand, the calculation overhead and unstable trigger distance caused by the measurement of standard deviation degrade its performance.

Server Benchmarks

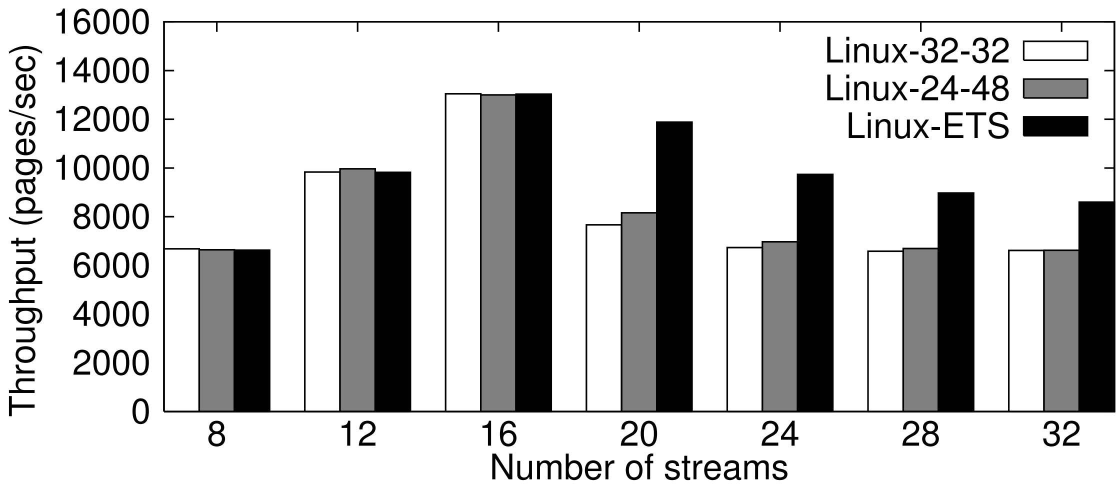


Figure 2.9: Throughput of SPC-2 VOD-like workload

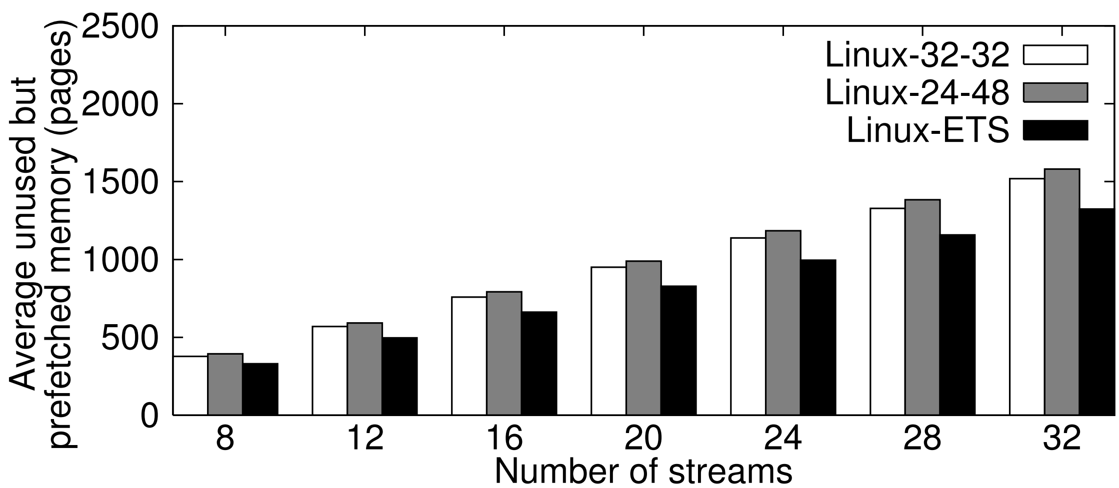
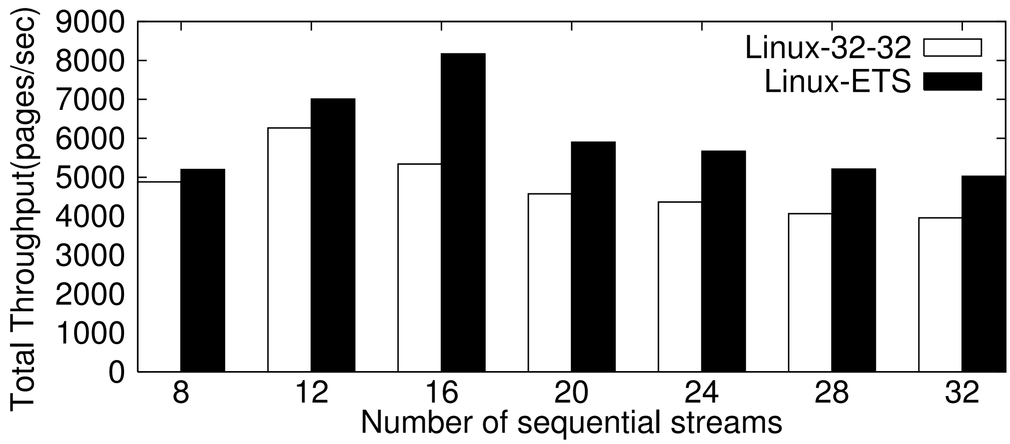
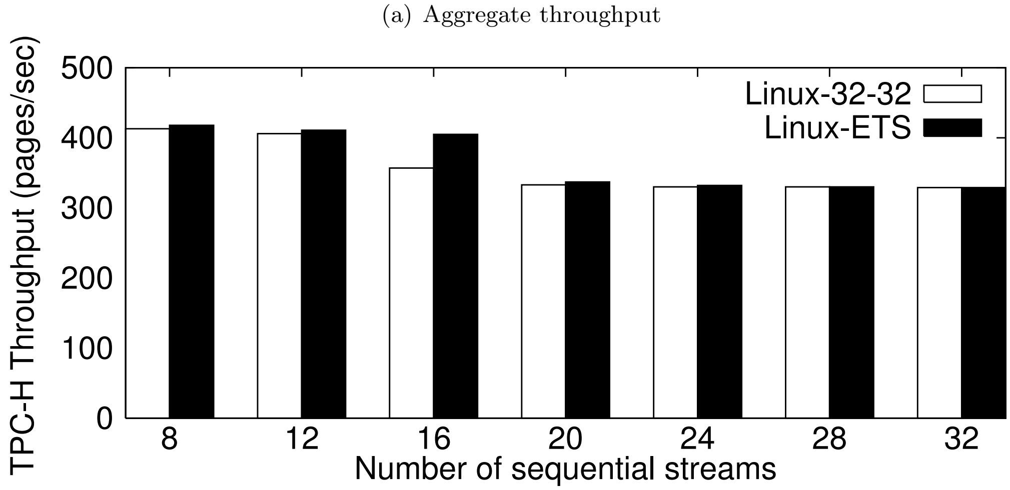


Figure 2.10: Memory usage of SPC-2 VOD-like workload



(a) Aggregate throughput



(b) TPC-H throughput

Figure 2.11: Throughput of mixed SPC-2 VOD-like and TPC-H workloads

Figure 2.9 depicts the performance for an SPC2-VOD-like workload using different algorithms. Here a certain number of sequential streams access the server, each sending requests to a separate file. All streams have the same think time between consecutive I/O requests, which is set to 33 ms. To create streams with different access rates, we use four different per-stream request size levels (16KB, 32KB, 64KB, 128KB), simulating varying video qualities. In the figure we show the aggregate throughput with different numbers of streams, each randomly selects a request size level. Since the relative performance of ESF compared with ETS is similar to that observed in sections 2.6.2 and 2.6.2, we omit it in this section. From the results we can see that when the number of streams is small, all three algorithms have nearly the same performance. That is because with relatively idle disks and short disk access time, all streams can be satisfied. However, ETS is performing significantly better than the other two algorithms when there are more than 16 streams, and therefore fast streams need more aggressive prefetching. The average improvement of ETS over the native Linux algorithm is 20.1% and the peak improvement is 55.0%. The performance of 24-48, on the other hand, is almost the same with 32-32, due to the same reason as discussed in Section 2.6.2.

Also, in Figure 2.9 we plot the average per-stream memory usage by the three algorithms. As expected, the 32-32 and 24-48 algorithms have approximately the same memory usage level. ETS, in contrast, settles to a lower number although it is designed to maintain the same memory usage as the base algorithms. This is due to that the trigger distance needs to be bounded by the prefetching degree, and the fastest streams are not able to grow their trigger distance as indicated their access rates. Overall, this shows that ETS is able to generate a 20% performance enhancement while using about 13% less memory space.

Finally, we examine the impact of our SCM algorithms on random accesses that co-exist with sequential access streams, by running the above VOD workload concurrently with a TPC-H workload, which is composed of mostly random accesses. Figure 2.11 shows the aggregate throughput of both workloads and Figure 2.9 shows the throughput of TPC-H accesses. Since the relative performance of 24-48 compared with the other two algorithms is similar to that in Figure 2.9, we focus on the comparison between 32-32 and ETS here.

We can see that with concurrent random accesses, ETS still has significantly better aggregate throughput than the native 32-32 algorithm, with an average improvement of

26.1% and a maximum of 53.0%. The average improvement is slightly higher than that with SPC2-VOD alone, mainly because that with the additional TPC-H workload, the disk access time increases, which enables ETS to show performance benefits even for small number of streams.

Finally, we observe that the TPC-H throughput is increased slightly with ETS, by up to 13% and on average 2.6%. The impact of ETS on the performance of random accesses is twofold. On one hand, with a larger prefetch degree compared with 32-32, its disk traffic pattern is more favorable to random accesses. In addition, it uses a smaller amount of memory as we observe in Figure 2.10. On the other hand, the increased sequential access throughput causes a higher workload to the disks which are shared with random accesses. The observed change in TPC-H throughput is the result of these two counter acting factors.

2.7 Summary

In this chapter, we revealed that the data prefetching problem is closely mirrored in the business discipline of supply chain management (SCM). In particular, the inventory theory models can be used to draw insights into the prefetching resource allocation problem when concurrent application request streams possess different accessing speeds. Our experiments demonstrated that the corresponding inventory theory solutions, when applied to data prefetching, can successfully improve multi-stream I/O performance.

2.8 Future Work

Through our experiments, we have found that the complexity in disk access time modeling currently prevents certain SCM policies, such as Equal Safety Factor, from working well consistently. The simpler Equal Time Supplies policy, on the other hand, generates rather stable improvement and can easily be plugged into existing dynamic sequential prefetching algorithms.

In the future, more sophisticated SCM models could be investigated, as well as efficient online I/O performance monitoring techniques, to better understand the inventory phenomena in prefetching scenarios and develop more effective performance solutions. Moreover, SCM techniques could potentially be applied in other aspects of memory resource management, including virtual memory and write buffering.

Chapter 3

Coordinated, Hierarchy-aware Multi-level Prefetching

3.1 Motivation and Contributions

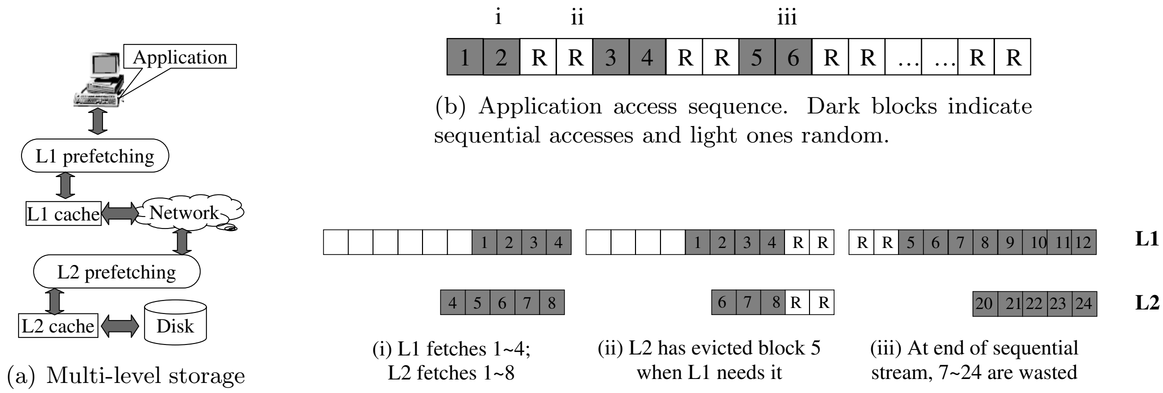
As we discussed in Chapters 1 and 2, prefetching is an important technique which exploits data locality to address the challenge of data intensive applications. The SCM-based algorithms introduced in Section 2.5 effectively improve the performance of current prefetching approaches by coordinating resource allocation among multiple sequential streams with different access rates.

Meanwhile, as service-based (especially web-based) applications prevail, cache management frequently has to be extended to multiple levels. For example, a web-based data center will have large storage caches equipped at both the front-end web servers (upper level) and the back-end storage servers (lower level). Figure 3.1(a) gives a sample architecture of such systems. How to effectively manage the aggregate cache space and improve the overall system performance has been studied extensively in the recent years. However, existing studies have focused on the multi-level caching problem [23, 60, 126, 129], while to our best knowledge no research has targeted coordinating the prefetching operations in a multi-level storage system. In contrast to the relative lack of studies it receives, prefetching has a crucial role on multi-level server systems. Many service applications hosted by such systems are read-intensive and possess heavily sequential access patterns. Examples include commercial or scientific data queries, web document processing, and multimedia hosting/streaming. Such applications benefit tremendously from file system prefetching.

In a multi-level system, prefetching is needed at each level to hide the latency of fetching data blocks from the lower layer. However, if prefetching is carried out independently at each layer, the system will not be able to make a coordinated use of the combined cache space. In particular, when a single-level prefetching algorithm is too conservative or too aggressive for a certain application workload, this mismatch will be magnified when multiple levels of prefetching are stacked up together.

On one hand, a conservative prefetching scheme, when applied to the lower level, may not be able to effectively use the cache space to hide I/O costs. Other researchers have recently observed that due to the low temporal locality at the lower level caused by higher-level caching, as well as the increasingly bottleneck-prone storage device performance, aggressive lower-level prefetching is especially appealing [75]. The lower level cache space may be better utilized when used as a staging area for higher levels to hide disk I/O costs, than as a backing store for demand paging. Also, aggressive lower-level prefetching helps handle the larger and more bursty requests caused by the batching effect of upper-level prefetching. Conservative prefetching at the upper level may use a small, fixed prefetch depth, or grow the prefetching depth very slowly. This in turn gives the lower level less encouragement for prefetching even with highly sequential accesses. Therefore, a slightly conservative single-level prefetching algorithm may need to be speeded up a lot when applied to multiple levels.

On the other hand, as people realize the growing appeal of prefetching [94], more aggressive prefetching strategies are likely to be adopted at each level. When multiple such uncoordinated software layers are stacked together, there is a compounding effect causing overly aggressive prefetching, which may significantly increase the burden of the storage devices, waste both the cache space and the I/O bandwidth, and degrade the application performance. Known problems associated with aggressive prefetching will be intensified when a system is expanded into multiple layers with uncoordinated prefetching: namely *cache pollution* (too eager prefetching taking space from more useful data) and *prefetch wastage* (prefetched data being evicted before they are used) [44]. Also, aggressive multi-level prefetching may accumulate many prefetched data blocks in the lower level cache even after such blocks have been passed up to the upper level, which is especially undesirable when the access pattern is mostly random or a mixture of sequential and random accesses.



(a) Multi-level storage system architecture

(b) Application access sequence. Dark blocks indicate sequential accesses and light ones random.

(c) Cache status snapshots corresponding to the three access points indicated in (b)

Figure 3.1: Sample architecture and behavior of uncoordinated multi-level prefetching

Finally, these problems related to aggressive prefetching can further be aggravated when the system employs a n-to-1 or n-to-$m(n > m)$ mapping between the clients and servers, requiring each server's space and bandwidth resources to be split between multiple clients.

Figures 3.1(b) and 3.1(c) demonstrate the problems with uncoordinated multi-level prefetching, using an adaptive prefetching depth, which increases by 1 at each sequential hit. In this example, the upper level cache is larger than the lower level one. After access point (i) the upper-level will prefetch blocks 3-4, triggering the lower level to prefetch blocks 5-8. With a limited lower-level cache size, prefetched block 5 will be flushed out of cache at point (ii) by the accesses of two random blocks. Therefore when block 5 is needed by the upper level, both levels will suffer a miss. Besides, after both access points (i) and (iii), redundant blocks are cached in both levels, while prefetched data have a lower chance of being requested again, at least at the lower level. In addition, the unnecessary prefetching of blocks 7-12 at the end of the sequential run will be extended to blocks 13-24 at the lower level.

The above example illustrates the deficiencies of multi-level independent prefetching, by showing one instance of overly aggressive prefetching caused by the amplifying effect from both leverls performing adaptive prefetching.

Similar to the observation made in Chapter 2, we discover that the multi-level prefetching problem is fundamentally similar to a branch of SCM, namely multi-echelon inventory control. The above deficiency caused by lack of coordination corresponds to the "bullwhip effect" that has been long studied in SCM. In Section 3.4 we present the mapping of concepts and an SCM-based, information sharing approach.

In real multi-level systems, however, upper and lower level systems are often designed and managed separately. Furthermore, one lower level might be connected to different types of upper level systems and vice versa. Therefore, a transparent solution is very desirable. Based on this consideration, we design a transparent coordination called PreFetching Coordinator (PFC), which acts like an "extension cord" that connects the existing prefetching algorithms at different levels, each working on top of an existing cache management strategy. Further, PFC enables coordinated prefetching across more than two levels, and potentially the stacking of different prefetching algorithms. Finally, PFC does not require any application hints or a-priori knowledge on applications' access pattern.

The main idea behind PFC's operation is to place an immediate layer of intelligence between the upper- and lower-level strategies for prefetching and cache replacement. By observing the upper-level requests and the lower-level prefetching behavior, PFC detects whether the current prefetching is too aggressive or too conservative, and tries to refrain or boost the lower-level prefetching activity, while at the same time avoiding caching prefetched data redundantly at multiple levels.

We designed a novel PFC algorithm that acts as a middleman between two adjacent levels of caching/prefetching, and is independent of the specific prefetching or replacement algorithms adopted at each level. PFC is adaptive, transparent to applications, and maintains the I/O interface between the multiple levels.

We evaluated both the information sharing method and PFC with a verified multi-level simulator, along with an accurate disk simulator, using multiple storage system and application access traces. In our simulator we implemented four well known prefetching algorithms used in real systems: P-Block ReadAhead (RA), Linux kernel prefetching, SARC, and AMP. Our experiments show that for all the algorithms, when they are applied to a two-level storage system, the addition of PFC can considerably improve the overall system performance (in terms of average request response time), by up to 60% and on average 14%. In particular, PFC is able to regulate the prefetching aggressiveness and achieve a performance improvement for all types of trace workloads: highly sequential, highly random, and mixed patterns. Besides lowering the request processing time, PFC enhances the overall system resource utilization by reducing redundant caching and controlling wasted prefetch when cache space becomes tight.

3.2 Related Work

Prefetching As discussed in Section 2.3, prefetching techniques for single-level systems have been widely studied. Many prefetching algorithms were proposed to answer the key questions of “what to prefetch” and “when to prefetch”. Gill et al. recently gave a quite comprehensive classification of existing prefetching approaches [44]. It indicates that although many sophisticated algorithms have been proposed to perform stride-based [40, 70, 32, 7] or history-based prefetching [74] to “guess” the best blocks to prefetch next, most commercial storage systems adopt simple schemes such as sequential prefetching. The reason is that sequential prefetching is able to provide good long-term prefetching accuracy for diverse workloads, without imposing the cost of extra I/O involved in maintaining and using the access history. While PFC is algorithm-independent, our discussion and evaluation are focused on sequential prefetching.

Although there are a wealth of studies on multi-layer cache management, existing work on multi-layer prefetching is quite limited. Research efforts that we are aware of are on multi-layer hardware prefetching for the CPU caches [20, 39], which use fixed, uncoordinated strategies at different cache levels. The DiskSeen [36] technique exploits knowledge of on-disk data layout to direct efficient file-level prefetching. The most related work to ours is STEP [75], which is motivated by the need to perform aggressive lower-level prefetching. STEP accurately detects sequential access patterns as well as disk thrashing patterns, and makes prefetching decisions accordingly. Like PFC, STEP optimizes the lower-level prefetching behavior with the awareness of upper-level prefetching or caching. However, it promotes aggressive lower-level prefetching, while PFC may moderate the lower-level activity both ways. In addition, STEP is itself a stand-alone lower-level prefetching algorithm, while PFC is a portable, generic optimization that can be applied to any single-level prefetching/caching algorithms. Finally, STEP was shown to improve the multi-level system performance significantly with sequential workloads while having no impact on handling random workloads. In contrast, our results show PFC brings considerable performance gain to both types of workloads.

Some other studies on multi-level systems utilize application hints to direct prefetching [21, 96]. PFC, on the other hand, does not require such hints or modification to the inter-level I/O interfaces.

Space coordination between prefetching and demand paging Besides the problems of "what to prefetch" and "when to prefetch", another important issue in prefetching is how to allocate the shared memory cache space among prefetched and demand paged data. There are a number of previous studies about managing prefetched data in a cache shared with demand paged data [18, 63, 93, 72]. Several other solutions alleviate the problem of *cache pollution* [100, 92] by carefully limiting the space that prefetched data can use. As a recent example, the SARC algorithm [45] uses two separate LRU queues for sequential and random data respectively, and adjusts their sizes according to the access pattern. As we show in this chapter, PFC can seamlessly work together with such techniques.

Multi-level cache management There have been many research studies in the contexts of demand paging and general cache management for multi-level storage systems. Previous research in different environments has noticed the weakness of LRU-like algorithms for lower level buffer cache in a hierarchy [34, 85] and pointed out feasible solutions for different systems [16, 124, 134]. This group of work focuses on improving the lower-level caching performance in reaction to the upper-level caching effect.

There has also been work on collaborative caching across multiple layers of storage. Chen et al. categorized existing work on multi-level buffer cache collaboration into two paradigms [22]: *hierarchy-aware caching* [23, 8, 134] and *aggressively-collaborative caching* [33, 126, 60]. The authors indicate that although aggressively-collaborative caching utilizes the aggregate buffer cache space more sufficiently, hierarchy-aware caching has the advantage of being transparent to the storage client software. Their empirical evaluation based on typical commercial storage system workloads reveals that if local optimizations are properly applied, the performance gain of aggressively-collaborative caching over hierarchy-aware caching is actually very limited. PFC can be viewed as a hierarchy-aware strategy for multi-level prefetching.

One category of collaborative cache management particularly related to our approach is *exclusive caching*, using mechanisms such as a DEMOTE operation [126], eviction-based data placement [23], or having the client side keeping track of the server cache status [8]. For single-level systems, the Free-behind technique [79], used in Solaris, tries to evict sequentially accessed blocks. In a sense, PFC implicitly performs exclusive caching by selectively bypassing the lower-level cache. Like the above approaches, this bypassing helps preserve the combined cache space. However, PFC is unique by performing prefetching-aware bypassing to actively throttle the prefetching aggressiveness.

Improving the cache hit ratio has traditionally been the goal of cache management research. In a more recent work, however, Yadgar et al. introduce a new algorithm called Karma [129], whose optimization goal is the overall I/O cost instead of hit ratios. Karma is shown to outperform existing multi-level caching solutions given certain I/O hints. Like Karma, PFC tries to minimize the overall I/O time. However, instead of using hints, PFC works through a feedback system based on the dynamic interplay of application access pattern, cache space distribution, and hardware speed.

Finally, most hierarchy-aware multi-level cache management schemes [23, 8, 134] also accommodate multi-client settings. The discussion and evaluation of this chapter is limited to single-client scenarios (which indeed represent a significant portion of real-world multi-level storage environments [22]). However, PFC can be easily extended to work with both multi-client systems and multi-stream workloads, since it takes a light-weight approach by adjusting prefetching parameters rather than explicitly performing cross-layer data placement.

3.3 Sample Sequential Prefetching Algorithms

While there have been more sophisticated strategies proposed, such as history-based prefetching [62, 117, 71], our discussion focuses on *sequential prefetching*, where a set of contiguous blocks beyond the ones requested will be prefetched. Sequential prefetching is used by most commercial systems as it achieves long-term prefetching effectiveness without making assumptions on the application access pattern or incurring extra I/O in making predictions [44]. For sequential prefetching, there are two common decisions made by a prefetching algorithm: "how much to prefetch" and "when to prefetch". Many prefetching algorithms used in actual systems today are adaptive and adjust the *prefetch degree (p)* dynamically according to the access pattern observed, prefetching farther beyond the requested blocks if the sequential access pattern is confirmed by hits on prefetched blocks.

Regarding the timing of prefetching operations, *synchronous* algorithms issue a prefetch request only when there is a cache miss, while *asynchronous* algorithms generally use a *trigger distance (g)* to start prefetching when there is a hit: the next batch of blocks will be prefetched when the block with a distance of g from the end of the set of prefetched blocks is accessed. Some asynchronous prefetching algorithms, such as RA and Linux (to be introduced below), do not have a trigger distance, but trigger prefetching on each hit and each miss.

Below we briefly describe several representative prefetching algorithms that are used in our study. We implemented these algorithms in our simulator and evaluated PFC's impact on their two-level performance.

RA The P-block Readahead prefetching algorithm (RA) is an extension of the OBL (One-Block Lookahead) [107] algorithm, by increasing the prefetch degree p from 1 to P. P can be either fixed or adaptive [31, 110]. In our experiments, we used a fixed degree of $P = 4$. Thus RA has a relatively conservative behavior compared with other algorithms for sequential workloads, but a rather aggressive behavior for random workloads.

Linux prefetching The Linux prefetching algorithm maintains for each file a *read-ahead group*, which contains all the blocks prefetched on the current file access and a *read-ahead window*, which contains both the current and the previous read-ahead groups. The algorithm determines that the file is accessed sequentially if the next access is within the read-ahead window and will prefetch another group with twice the size as the current read-ahead group. This way, sequential accesses will double the prefetch degree, until the read-ahead group size reaches a pre-defined maximum value, set to be 32 blocks in 2.6.x kernels. If the next access is outside the *read-ahead window*, the algorithm will resort to conservative prefetching and only prefetch a minimum number of blocks (by default 3) after the on-demand block.

Is has been shown that the Linux kernel prefetching has significant impact on buffer cache replacement algorithms [14]. Among the prefetching algorithms we have experimented with, the Linux kernel prefetching algorithm is the most aggressive one, with an exponentially growing prefetching degree, which is aggravated when performed at two or more levels. In addition, compared to other algorithms we found that Linux prefetching obtains considerable performance gain by maintaining per-file prefetching parameters.

SARC SARC [45] was developed at IBM and deployed in the IBM flagship storage controllers DS6000/8000. Unlike the other algorithms we examined, ‘ SARC is actually a combined algorithm doing both prefetching and cache management. It uses a fixed prefetch degree p and a fixed trigger distance g. To handle mixed workloads that contain both sequential and random accesses, SARC maintains two LRU queues, namely SEQ and RANDOM, for sequential and random data, respectively. It optimizes the use of the fixed cache space by equalizing the *marginal utility* of the two queues.

AMP The AMP algorithm [44] is proposed recently to coordinate prefetching in multiple sequential access streams and has been deployed by the new IBM DS8000 system released in Oct. 2007. It adjusts both p and g dynamically and coordinates the prefetching of multiple access streams. The design of AMP was based on the observation that the cache space is best utilized when the prefetch degree for stream i is equal to the product of the request rate of stream i and the average cache life. AMP increases p_i when the sequential access pattern is confirmed and reduces p_i when it detects overly aggressive prefetching (from the eviction of prefetched blocks that have not been accessed). The trigger distance g_i is reduced when p_i is reduced, and increased when the prefetched block is found to be waited on by an on-demand request, which indicates that the prefetching has not been triggered early enough.

3.4 SCM Mapping and Approach

3.4.1 Multi-level Prefetching → Multi-echelon Inventory Control

In Section 2.4 we present a mapping between a prefetching problem in modern storage systems to an SCM problem. The discussions, however, have been limited within single level systems. As we introduced in Section 3.1, in multi-level storage systems prefetching is typically performed individually at each level to hide the latency of fetching data blocks from the lower layer.

Similarly, a supply chain normally has multiple levels of entities as well, such as *grocery store* → *distribution center* → *warehouse*, where each entity submits batched replenishment orders to its supplier. The branch of research in SCM on planning these inventory and replenishment behaviors is called *multi-echelon inventory control*. Our aforementioned problem mapping process can be repeated to construct a multi-echelon inventory problem.

In the following Section 3.4.2, we present an existing SCM analysis that lends us insight into a multi-level prefetching performance problem and inspires our solution.

3.4.2 SCM-based Information Sharing Approach

Modern server systems often have a multi-level architecture, employing layered servers for application/web, database, and storage services [22]. Meanwhile, access-pattern-driven optimization is also done at many levels, from the front-end server to the disk scheduler. The problem is that in these systems, the original access pattern is altered by layers of caching and prefetching, and the observed access pattern deviates from the true application request pattern more and more as requests go down the hierarchy. Compared with the cache replacement problem, where filtered accesses and exclusive caching [126] are desirable features, for prefetching, the batched, long, sequential accesses induced by a higher-level prefetching operation is often misleading to lower-level prefetching algorithms.

While this information distortion problem has recently been addressed in storage systems [131], the similar situation in supply chains has been studied for much longer. The term *bullwhip effect* is used to describe the phenomenon of variations in demand getting amplified as we go down the supply chain, away from the end customers. One famous illustration of the bullwhip effect is the *Beer Game* [82], developed by the MIT Sloan School of Management in the 1960's, where players form a beer supply chain without communication other than receiving orders. It demonstrated the frustrating nature of individual supply chain entity management, with each entity trying to minimize back orders and inventory levels merely based on the demand observation.

In this research, we exploit one straight-forward solution from SCM: extending the visibility of customer demand as far as possible. This approach has been most successfully adopted by the Walmart distribution system [2], generating better inventory positioning and lower costs by enhancing the customer demand visibility. At Walmart, the point-of-sale (POS) data are collected from the cash registers, processed by corporate headquarters a few times a day, and used to guide its distribution center in interacting with both the stores and the suppliers.

We applied the above approach to attack the bullwhip effect in multi-level prefetching, by having an upper-level server explicitly label the appended prefetching requests in each request to its direct supplier, a lower-level server. Most dynamic prefetching algorithms will adjust the prefetch degree and trigger distance on each block access to the cache, hit or miss. With the SCM-based solution, these actions are only taken for those blocks in the original application request. While a prefetched block appended to the application request may still trigger other actions at the lower level, such as issuing another prefetch request, the lower-level trigger distance and prefetch degree will not be affected.

We implemented the above mechanism in a two-level storage simulator, where several existing prefetching algorithms have been implemented at each level. In Section 3.6.3, we present a case study where this simple approach is able to correct a severe performance problem caused by inter-layer information distortion.

Regarding the overhead of such a mechanism, we consider I/O interface change rather small and comparable to that of existing inter-layer coordination schemes (*e.g.*, [46]), and the size of the extra data added to the communication should not be a problem considering the disk system is typically the performance bottleneck [22].

3.5 Hierarchy-aware Prefetching with PFC

The approach introduced in Section 3.4 reveals the importance of information sharing and behavior coordination between different levels. However, in real systems a transparent approach is much more preferable to those modifying both upper and lower level systems. Therefore in this section we present PFC, which is a heuristic approach residing on the lower level system and optimizing system performance via analysis of upper level request patterns. Because PFC is completely transparent to upper level systems, we categorize it as a "hierarchy-aware" prefetching approach in contrast to "client-controled" approaches.

3.5.1 PFC Architecture

Although PFC is designed to be able to apply to systems with more than two levels, we focus on two-level systems (a common architecture in today's multi-level storage systems and data centers) in our discussion and experiments. For brevity, we refer to the upper (client) level as L1 and the lower (server) level as L2, for the rest of this chapter.

As mentioned in Section 3.1, the multi-level prefetching problem presents somewhat conflicting demands. The goal of PFC is to moderate the prefetching process to achieve a desired level of aggressiveness in order to improve the overall system performance and resource utilization. The major design question here is "where should PFC reside", L1, L2, or both? We decide to place PFC in L2, as an intermediate layer between the client and the server's native prefetching and cache management, for a more portable and algorithm-independent design. A previous study [22] on multi-level cache management reveals that it is not worthwhile to sacrifice the transparency of the L1/L2 interface to add "aggressively-collaborative" mechanisms at the client side. In contrast, "hierarchy-aware" mechanisms sitting at the server side that leverage the knowledge gathered on the upper-level cache to perform intelligent cache replacement are more feasible and can achieve similar performance gains. We believe this also applies to the multi-level prefetching scenario, and our empirical results from implementing and evaluating a client-side prefetching coordination scheme indicate the same. Due to the space limit, in this chapter we only discuss our proposed server-side PFC design.

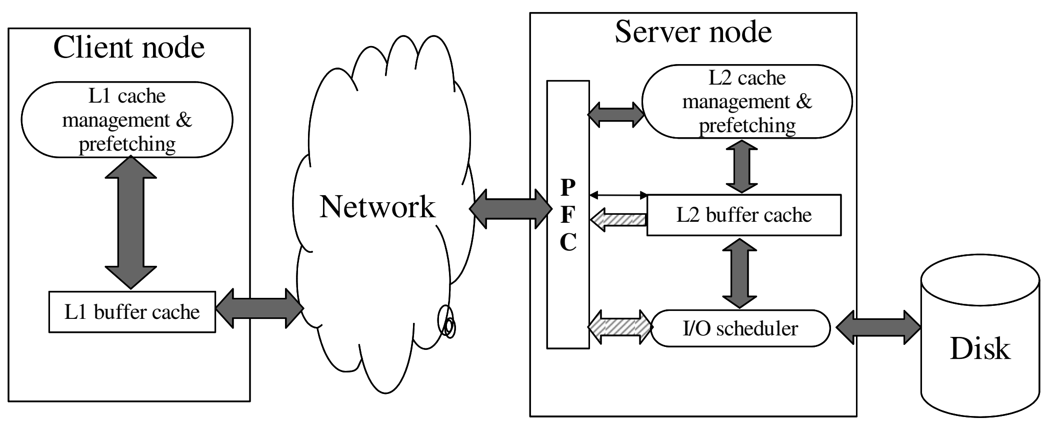


Figure 3.2: Software architecture for two-level systems using PFC.

Figure 3.2 illustrates the location and interfaces of the PFC module, and give an overview of how it functions (with more details and the algorithm discussed in 3.5.2). PFC resides at the server side as an intermediate gateway between the client node and the server-side I/O request processing. It intercepts the client requests (which may have included client-side prefetching), and relay data blocks between the client interface and the L2 I/O stack. It may query the L2 cache status and find out whether a certain block is currently cached. In general, PFC is aware of the existence of caching/prefetching both

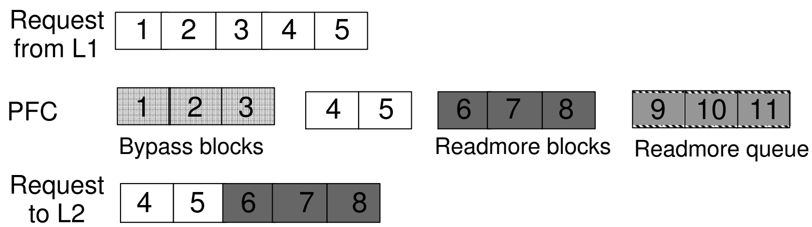


Figure 3.3: Sample PFC actions on L1 requests

above and beneath the L1/L2 interface, but unaware of the actual strategies used.

Based on the observed L1 requests and the L2 cache inventory regarding requested blocks, PFC detects whether the L2 prefetching may be too conservative or too aggressive. For example, it can infer the aggressiveness of L1 prefetching by looking at the request size, or that of L2 prefetching by checking how many blocks beyond those accessed by L1 have been stocked in the L2 cache. Depending on the situation, PFC may take one or both of the following two actions:

- *bypass:* bypassing the L2 cache and directly feed some or all of the requested blocks to L1. PFC does this by interacting directly with the L2 I/O scheduler, or drawing cached blocks from the L2 cache without notifying the L2 native caching/prefetching unit of a hit. This action serves two purposes: 1) slowing down L2 prefetching by "faking" an L1 access stream with weakened sequential pattern, and 2) performing exclusive caching on sequentially accessed blocks by avoiding caching them in L2.
- *readmore:* appending additional blocks to prefetch to the original L1 request. This action speeds up L2 prefetching when PFC decides the native L2 strategy is not aggressive enough.

3.5.2 PFC Algorithm

The goal of the PFC algorithm is to adaptively select a proper degree of aggressiveness in prefetching and make efficient use of L2 cache space. As mentioned in the previous section, PFC accomplishes this by activating one or both of two somewhat counter-acting operations: bypass and readmore. Figure 3.3 illustrates possible actions on a sample request issued by L1. In this example, the original request contains consecutive blocks 1-5. PFC may decide to bypass the first three blocks, and perform additional prefetching of its own for "readmore" blocks 6-8. This way, the request for blocks 1-3 are directly issued by PFC to the L2 I/O scheduler, while the L2 native caching/prefetching unit sees an altered request for blocks 4-8.

Algorithm 1: $PFC\_Process\_Req(req_u = [start_u, end_u])$

$req\_size = end_u - start_u + 1$;
avg_req_size = average request size so far.
/* If req_size is larger than two times of avg_req_size, exclude it from calculation of avg_req_size. */
$rm\_size = MAX(req\_size, avg\_req\_size)$;
$PFC\_Set\_Param(req_u)$;

$start_{pfc} = start_u + bypass\_length$;
$end_{pfc} = end_u + readmore\_length$;
process request $[start_u, start_{pfc} - 1]$ directly;
forward request $[start_{pfc}, end_{pfc}]$ to native L2 processing;

/*Insert new items into queues*/
if *bypass_queue or readmore_queue is full* **then**
    evict oldest items until required space is available;
insert $[start_u, start_{pfc} - 1]$ into *bypass_queue*;
$end_{rm} = end_{pfc} + rm\_size$;
insert $[end_{pfc}, end_{rm}]$ into *readmore_queue*;

Note that PFC's bypass action intercepts L1 requests from reaching L2, but may still access L2 cached data. In this example, blocks 1-3 will not be requested through the native L2 caching/prefetching modules. However, if some of these blocks are already cached in L2, PFC will serve L1 by reading them from the L2 cache rather than going to the disk. This way, the L2 cache may get a "silent hit" not registered with the native caching/prefetching algorithm.

The key decisions the PFC algorithm has to make, of course, are "when" and "how much" to perform bypass and/or readmore. Intuitively, the bypass blocks should be a prefix of the original request, as they are expected to be accessed first and ought to be cached closer to the application. At the same time, the readmore blocks should be the blocks immediately following the original request. The remaining challenge is to determine the trigger condition and the degree for each action.

To perform such dynamic decision-making, PFC manages two queues, the *bypass queue* and the *readmore queue*. These queues do not store real data blocks, but block numbers. Both are initially empty, and maintained with the LRU policy (the least recently inserted or re-accessed blocks are evicted when the queue is full). In our experiments, we set the maximum size of both queues to 10% of the L2 cache size. The queues help PFC detect the need to increase/decrease the bypass or readmore levels, by setting the key PFC parameters *bypass_length* and *readmore_length*, which are both initialized as 0.

The PFC request processing procedure is given in Algorithm 1, which takes the original L1 request, computes the PFC parameters by calling subroutine *PFC_Set_Param()* (Algorithm 2), and processes the request with optional bypass and readmore actions accordingly. Finally, some of the requested blocks are added to the appropriate queue if they are not already there. Here PFC treats the two queues differently. The bypassed blocks are added to the bypass queue. However for the readmore queue, rather than adding the readmore blocks PFC appended to the L1 request, it adds blocks in a *readmore window* following those readmore blocks to the readmore queue, since the purpose of this queue is to detect if accesses to blocks in it could be hits if *readmore_length* were larger. The size of this window is determined by a parameter *rm_size*, calculated in Algorithm 1 from the current and average request sizes.

Algorithm 2 describes how PFC set *bypass_length* and *readmore_length*. Basically, PFC monitors the request pattern, as well as the hit status of requested blocks in the L2 cache and both PFC queues, to determine whether it needs to increase/decrease the bypass/readmore activities.

One upfront step is to check whether the L1/L2 prefetching is already quite aggressive. PFC considers the former true if the L1 request appears large (longer than half of the average L1 request size), and the latter true if as many blocks as requested immediately beyond the requested range are already stocked up in the L2 cache. In these cases, PFC will choose to bypass the entire L1 request, and set *readmore_length* to 0.

If neither condition above is satisfied, PFC will perform more detailed checking to see whether any blocks requested are found in the L2 cache, the bypass queue, or the readmore queue. If none of the blocks have been bypassed earlier, PFC assume the L1 cache can store more and increases *bypass_length*. Otherwise, if accesses to previously bypassed blocks are misses in the L2 cache, PFC infers that the L1 cache space is tight and the blocks have been evicted prematurely, thus bypassing them was a wrong decision. In reaction, it will reduce *bypass_length*. The treatment of *readmore_length* is coarser: it will be increased to *rm_size* if the sequential access pattern anticipated is confirmed by having

Algorithm 2: $PFC\_Set\_Param(req_u = [start_u, end_u])$

$hit\_cache = hit\_bypass = hit\_readmore = false$;

/* Check against aggressive L1/L2 prefetching */
if *((req_size > avg_req_size) and (L2 cache is full))* **then**
    $readmore\_length = 0$;
endif
if *([end$_u$, end$_u$ + req_size] ∈ cache)* **then**
    $bypass\_length = req\_size$;
    $readmore\_length = 0$;
    return;
endif

/* Check hit status of L2 cache and PFC queues */
for $start_u \le x \le end_u$ **do**
    if $x \in cache$ **then** $hit\_cache = true$;
    if $x \in bypass\_queue$ **then** $hit\_bypass = true$;
    if $x \in readmore\_queue$ **then** $hit\_readmore = true$;
endfor

/* Adjust PFC parameters */
if $!hit\_bypass$ **then** $bypass\_length{+}{+}$;
if $!hit\_cache$ **then**
    if hit_bypass **then** $bypass\_length{-}{-}$;
    if $hit\_readmore$ **then**
        $readmore\_length = rm\_size$;
    else $readmore\_length = 0$;
endif

a hit in the readmore queue, and reset to 0 if otherwise.

From the algorithms given, it can be observed that random accesses are likely to be bypassed, except at the beginning of the run when *bypass_length* is zero or very small. This is desirable since the temporal locality of L2 accesses is expected to be low. A related issue is that in our current PFC implementation, the lower level maintains a single set of parameters. However, it is easy to extend PFC to maintain per-client or per-file contexts, in order to better handle multiple access streams.

3.6 Performance Evaluation

3.6.1 Simulator Overview

The simulator used in our trace-driven evaluation is extended from an existing two-level storage simulator that was used in several previous studies on multi-level cache management [134, 23, 22, 136, 137, 135], which has been validated against real systems and released to public.

We extended this base simulator in two areas. First, we added prefetching to both levels and implemented several prefetching algorithms, as well as our proposed optimization. Second, we made the simulator time-aware. The base simulator was designed to study cache replacement algorithms, for which it suffices to only consider the access sequence and ignore the actual timing of the requests. As existing research suggested [14], when prefetching is taken into the picture, one should examine the overall system performance rather than just on the cache hit ratios. To calculate the disk I/O time, we connected our simulator to the widely used disk simulator DiskSim [41]. We also implemented in the simulator an I/O scheduler that imitates I/O scheduling in Linux kernel 2.6. Finally, with the assumption that the network interconnection between L1 and L2 is unlikely the system bottleneck, we used a simple model [29] to compute the communication cost as $\alpha + \beta \times message\_size$, where α is a fixed startup latency and β determines the size-dependent cost. In our experiments, we set α as 6 ms and β as 0.03 ms/page, both measured through tests of TCP/IP data transfers between two computers in a LAN.

In this chapter, we assume that the system is composed of one upper level cache, one lower level cache, and a disk, a valid setting for many real multi-level systems [22]. However, our simulator can be easily expanded both horizontally (to include multiple nodes at each level) and vertically (to add more levels), by replicating nodes and disks.

3.6.2 Test Workloads

Our simulation uses three large real-system traces, representing a variety of typical multi-level system workloads and carrying different degrees of randomness in accesses. Below we briefly describe these test traces.

SPC traces SPC[1] is a widely used benchmark collection provided by the Storage Performance Council, that has been adopted by many previous studies on prefetching and multi-level cache management [44, 45, 75]. We selected two workloads from SPC, "OLTP", traces from OLTP applications running at a large financial institution, and "Web", web-search traces from a popular search engine. The OLTP trace is the most sequential one in our test workloads, with only 11% of requests being random accesses. The Web trace, on the other hand, is the least sequential, with 74% of accesses random.

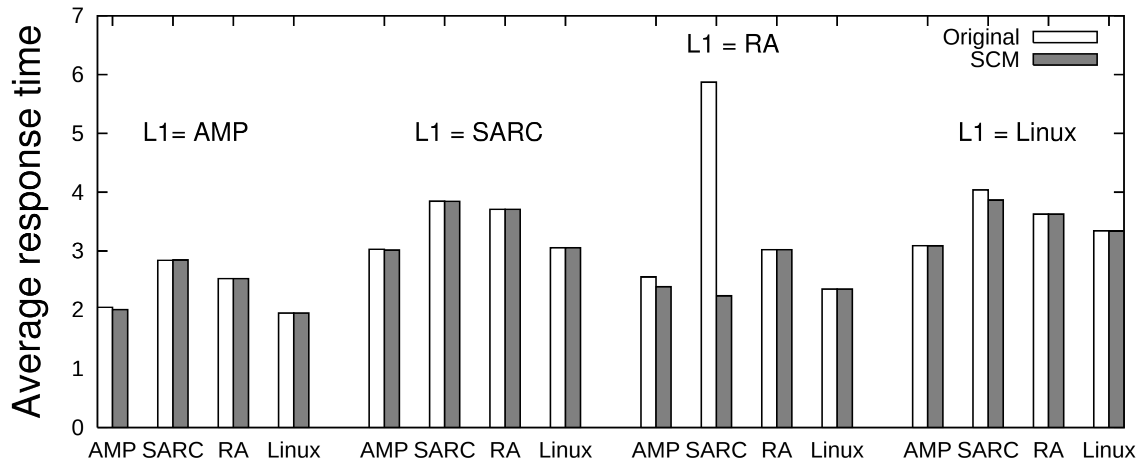
As our base simulator is not compatible with the newer version of DiskSim, we used a previous version (DiskSim 2), which has been used in several recent studies [74, 135]. The problem is DiskSim2 supports only older disk models with limited capacity. For example, the largest disk capacity allowed by DiskSim 2 is 9.1GB (with the Seagate Cheetah 9LP hard disk model used in our experiments). Due to this limitation, also to control the total simulation time, we used only the first 10GB of data requests from the SPC traces. That accounts for 10.8% of requests from the OLTP and 31.3% from the Web trace, resulting in a total footprint of 529MB and 8392MB, respectively.

Purdue Multi trace We also used one of the "Multi" traces collected by researchers at Purdue university in 2005 [14], from the concurrent execution of three applications: *csscope*, *gcc*, and *viewperf*. This trace accesses a total of 12,514 files, with a combined footprint of 792MB. This is a trace with mixed access patterns, with 25% of accesses being random. Unlike the SPC traces, where each trace record bears an application request timestamp, the Purdue traces were collected from running the workload benchmarks at the throughput allowed in a test system. We followed the way these traces were used in the Purdue researchers' work, by issuing the requests in a synchronous manner (only issuing the next request when the current one completes).

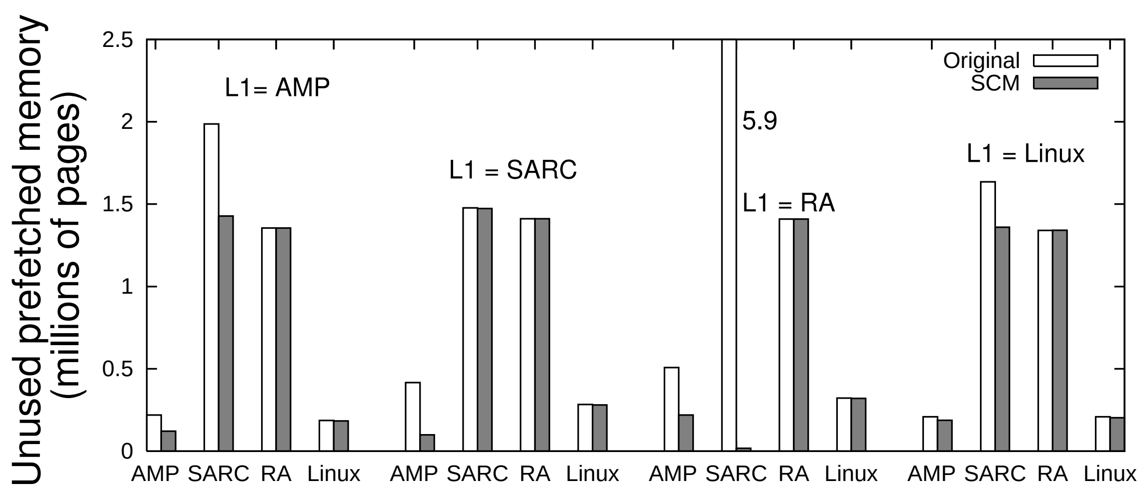
3.6.3 Evaluation Results of Information Sharing Approach

Here we present a case study where the SCM-based approach in extending customer demand visibility helped us to correct the bullwhip effect in multi-level data prefetching.

[1] http://traces.cs.umass.edu/index.php/Storage/Storage



(a) Response time



(b) Unused prefetched blocks

Figure 3.4: Multi-level prefetching with and without SCM approach

In this experiment, we tested all 16 combinations of the four prefetching algorithms implemented in the two-level simulator that we used, on the multi-application I/O trace collected by Purdue researchers [14]. Figures 3.4(a) shows the average response time of trace requests resulted from the algorithm combinations, where the bars are grouped by the L1 (upper-level) policy. Meanwhile, Figure 3.4(b) shows the number of data blocks that are fetched into the L2 (lower-level) cache but are evicted later without being used.

The white bars in the figures show the original algorithm performance and clearly one algorithm combination stands out: RA-SARC. After looking into the interplay between these two algorithms, we found the problems lies in the information distortion between the two levels. With this test, RA, the fixed read-ahead prefetching approach, uses a constant prefetching degree of 4. This means with every block request, the next 4 blocks will be prefetched, regardless of whether the access pattern is sequential. SARC, meanwhile, happens to detect the sequentiality of accesses with a default threshold of 4 contiguous blocks. Therefore, any application request, sequential or random alike, after passing through L1 will be perceived as sequential by L2 and encourage unnecessary prefetching.

Table 3.1: RA-SARC performance under different SARC sequentiality detection thresholds

SARC sequential threshold	3	4	5	6	7
Average response time (ms)	6.48	6.6	6.64	4.65	2.95

Table 3.6.3 further confirms this analysis. With the same RA configuration, the average response time drops back significantly when the SARC sequential access detection threshold is increased beyond 5. The reason that the threshold 6 produces a half-way improvement is because some of the random accesses cross the block border and access two blocks, and could still be detected as sequential with 4 additional blocks appended by L1 prefetching.

Table 3.2: Factors contributing to the performance gain of the SCM technique in the RA-SARC combination

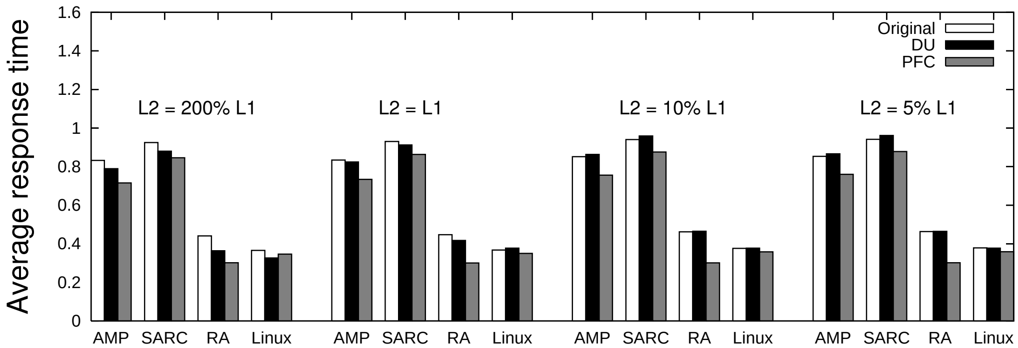
Metrics	Resp. time	L2 hit ratio	Unused prefetch	# of disk I/O	Amt. of disk I/O
Improvement	61.9%	187.2%	99.7%	−103%	69%

Back in Figure 3.4(a), we illustrate the difference made by our SCM-based solution, where the original application requested blocks are labeled in the L1 requests to L2. The bullwhip effect in the RA-SARC combination has been removed, and SARC actually outperforms all other L2 algorithms when RA is employed at L1. To give a more detailed view into this 61.9% performance improvement, Table 3.6.3 portraits the improvement generated by the SCM solution for the RA-SARC combination in four additional metrics. The L2 cache hit ratio is improved by more than 180%, as a result of the reduced cache pollution, and the amount of unused prefetch is cut in half. Also, the total amount of data requested from the disk has also been significantly lowered. However, the total number of disk request roughly doubles, due to the much reduced L2 prefetching aggressiveness. We found that one drawback of the block-labeling approach here is that when L2 prefetching parameter adjustment is limited to "real requests" from the application, the L2 decision making will suffer from the loss of information when parts of application requests are masked from L2 due to L1 cache hits. In our future work, we plan to study the benefit-cost tradeoff in demand information propagation.

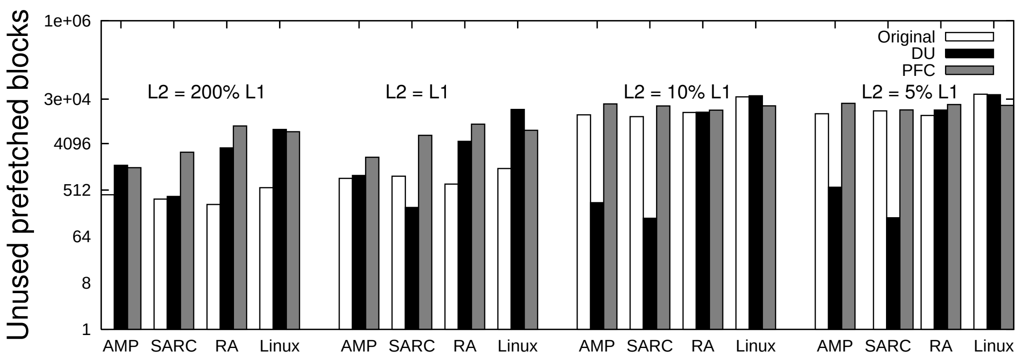
Even with this limitation, as can be seen in Figure 3.4(a) the SCM solution does not generate any negative performance impact to any algorithm combination, besides dramatically improving the RA-SARC performance. For several other algorithm combinations, it lowered the L2 prefetching agreesiveness significantly (thus saving disk bandwidth and energy), without hurting the request response time. When the L2 algorithm is RA or Linux, neither the average response time nor the unused prefetch is affected noticeably by the SCM solution. For RA, it is easy to explain as RA is oblivious to L1 access pattern. For Linux, it is because the sequentiality detection uses a more flexible window-based method, therefore the L2 prefetching is more resistent to L1 access pattern distortion or correction.

Finally, our results reveals that RA, a seemingly "dumb" algorithm not adopted by today's commercial systems, actually does a better job as an L1 algorithm than more sophisticated strategies such as SARC and Linux. From an SCM perspective, we believe this is because in most cases, its fixed and rather small prefetching degree generates less information distortion at L2 with sequential access streams.

3.6.4 Evaluation Results of PFC



(a) Average response time (OLTP, L1 = "H")

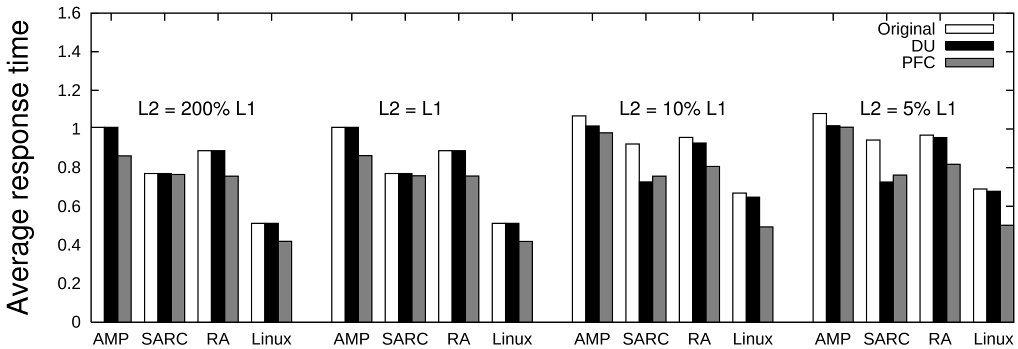


(b) Unused L2 prefetch (OLTP, L1 = "H")

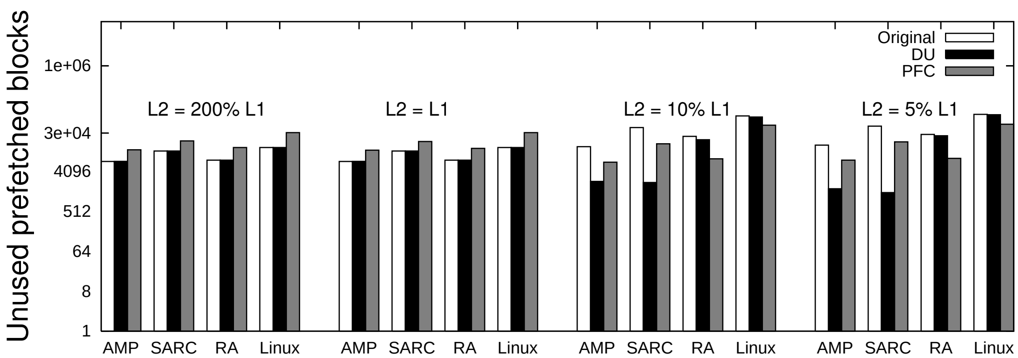
Figure 3.5: Impact of PFC on overall performance and unused prefetches for OLTP trace. Figures in the right column use log-scale at the y-axis.

Overall performance of PFC We evaluated PFC on the three trace workloads described above and the four existing prefetching algorithms discussed in Section 3.3. Each algorithm is applied to both L1 and L2. For every trace-algorithm combination, we tested different cache settings. The L1 cache size is set according to the trace footprint, with a "high setting" (H) that amounts to 5% of the total trace footprint, and a "low setting" (L) to 1%.[2] When the L1 cache size is fixed to H or L, we varied the L2 cache size by adjusting the L2:L1 size ratio, using four configurations: 200%, 100%, 10%, and 5%. This simulates different L2 base configuration, as well as the scenario where a single server node is simultaneously serving

[2]The L1 cache size may seem quite small with moderate trace sizes, compared to today's server configuration. However, such cache sizes are reasonable for analyzing cache behaviors, considering the increasing application concurrency on each node and the relatively small disk size used in this version of DiskSim.



(a) Average response time (Web, L1 = "H")



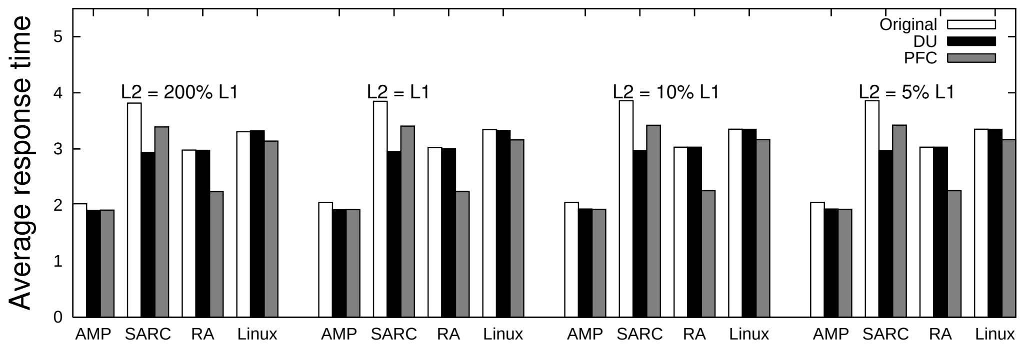
(b) Unused L2 prefetch (Web, L1 = "H")

Figure 3.6: Impact of PFC on overall performance and unused prefetches for Web trace. Figures in the right column use log-scale at the y-axis.

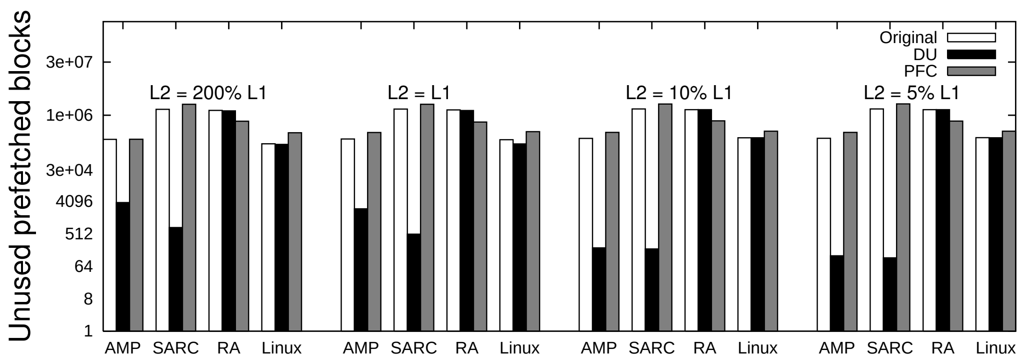
multiple client request streams. At both levels, LRU is used as the cache replacement policy, except for SARC, which comes with its own cache management strategy. Finally, all the discussion on hit ratio in this section is regarding the L2 cache, as we found PFC, as a server-side optimization, has little impact on the L1 cache hit ratio.

To compare PFC with non-prefetching-aware exclusive caching, we implemented DU [22], which marks blocks that have just been sent to L1 with the highest priority for eviction, assuming those blocks are to be cached by L1. Like PFC, DU is an L2 local optimization aware of the existence of upper-level cache activities.

Figures 3.5, 3.6 and 3.7 show the results of the tests using the "high" L1 cache size setting. Since PFC is designed to moderate multi-level prefetching in storage systems, we consider the most important metrics to be 1) the overall system performance (in terms of the average request response time) and 2) the unused prefetch (in terms of the total number



(a) Average response time (Multi, L1 = "H")



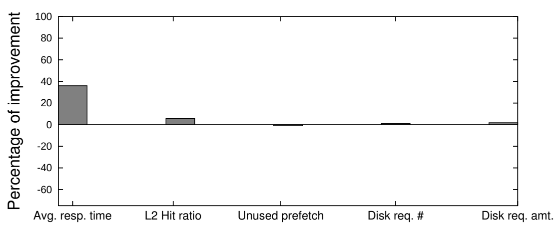
(b) Unused L2 prefetch (Multi, L1 = "H")

Figure 3.7: Impact of PFC on overall performance and unused prefetches for Multi trace. Figures in the right column use log-scale at the y-axis.

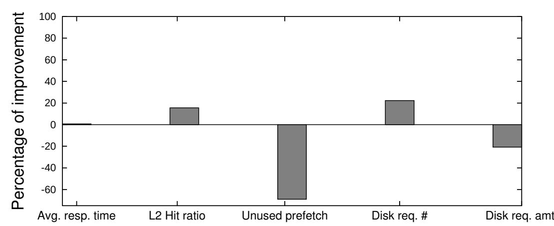
of blocks that are prefetched but not accessed when evicted or till the end of a test). The three figures in the left column plot the average response time, while the three in the right one plot the unused prefetch in log-scale. The "low" L1 cache setting tests yield similar results, and we omit the figures due to the space limit.

Table 3.3: Summary of PFC's improvement on the system overall performance.

Trace	Cache size	Prefetch Algorithm			
		AMP	SARC	RA	Linux
OLTP	200%-H	13.98%	8.49%	31.53%	5.23%
	200%-L	9.77%	10.88%	35.77%	9.50%
	5%-H	10.94%	6.74%	34.87%	5.26%
	5%-L	7.80%	11.41%	24.87%	15.10%
Web search	200%-H	14.66%	0.70%	14.83%	18.23%
	200%-L	12.54%	26.43%	16.42%	29.28%
	5%-H	6.56%	19.21%	15.57%	27.20%
	5%-L	5.76%	20.59%	14.80%	27.12%
Multi	200%-H	5.65%	11.13%	24.85%	5.03%
	200%-L	5.43%	8.29%	25.22%	5.18%
	5%-H	5.98%	11.30%	25.61%	5.52%
	5%-L	5.86%	10.39%	25.39%	5.87%



(a) Best case (51.9%): OLTP/Linux/L/5%



(b) Worst case (-4.7%): Web/AMP/L/200%

Figure 3.8: Case studies for tests where PFC obtained best and worst performance gain

PFC is shown to improve the average response time for all 96 test cases. The improvement is up to 35%, with an average of 14.6% over all cases. For the majority of the cases (around 77%), it also outperforms DU, which optimizes L2 space usage by evicting blocks passed to L1, but does not actively adjust the aggressiveness of L2 prefetching. PFC, on the other hand, may make the L2 prefetching more aggressive or more conservative based on the access pattern and cache status.

As can be seen from the three charts in the right column of Figures 3.5∼3.7, when the L2 cache size is large and the access pattern is highly sequential (OLTP, 200% and 100% L2:L1 cache ratios), PFC will actually aggravate L2 prefetching, resulting in higher numbers of unused prefetch. However, the overall performance is improved due to better L2 hit ratio, and the unused prefetch has a lower impact on the cache space utilization as there are not enough random blocks to compete for the L2 cache space. When the L2 cache size is relatively small, or when the accesses become more random (Web, 10% and 5% size ratios), in most cases PFC will slow down the L2 prefetching and reduce unused prefetch. In many of those cases, the L2 hit ratio is worse than in the original case, but PFC helps improve the average response time by reducing the number of disk requests and/or making shorter requests. Both of these help to lighten the disk workload and provide faster response to the application requests.

One final note on Figures 3.5∼3.7 is that PFC appears to maintain the relative performance of algorithms under most circumstances. This is appealing as PFC is intended to extend existing single-level prefetching algorithms found suitable for certain workloads to multi-level systems.

Table 3.3 summarizes the improvement on the average request response time with both the "high" and "low" L1 cache size settings. We can see that the most significant improvement comes from the RA tests (for all traces), where PFC's heuristics adds to the intelligence of the originally static RA algorithm. Also noticeable are the Linux tests with Web traces, where PFC improves performance significantly by regulating the L2 prefetching to avoid prefetching too much when two levels of aggressive prefetching are compounded. However, even with Linux, PFC may decide to prefetch more aggressively at L2, such as in the cases of OLTP, with "H" L1 cache setting and 200%/100% L2:L1 cache size ratios. In such cases, PFC generates more unused L2 prefetch, but improves the response time by over 5%. PFC also enhances performance considerably with the SARC and RA algorithms. For OLTP and large cache configurations, PFC makes the L2 prefetching more aggressive while for the other cases, it will suppress L2 prefetching. For random or mixed traces like Web or Multi, the reduced prefetching also translates to better L2 space utilization for random blocks, in addition to reduced I/O workload. In summary, PFC is able to make flexible, dynamic decisions, speeding up L2 prefetching in 9 test cases and slowing it down in 87 (collected from information not shown in the table).

Case studies To take a more detailed view into PFC's interaction with the native prefetching algorithms, in Figure 3.8 we plotted additional metrics for two test cases, where PFC obtained the most and the least performance gain (35% and 0.7% improvement on the average response time). These metrics include the L2 hit ratio, the total number of disk requests, and the total amount of disk I/O. For both cases, PFC decides to make the L2

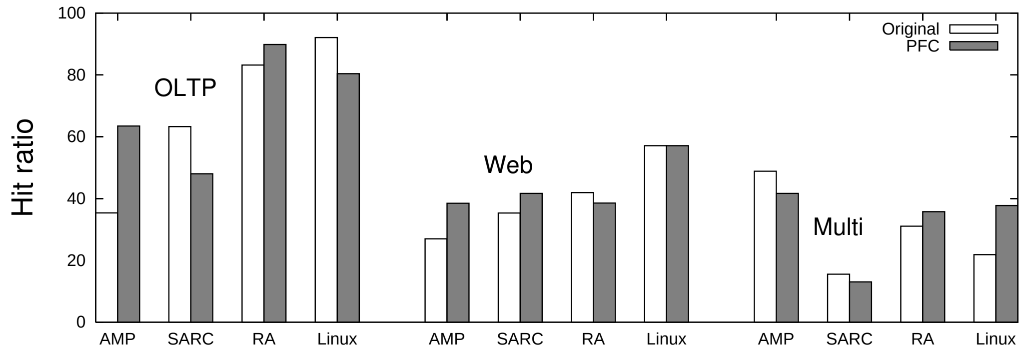
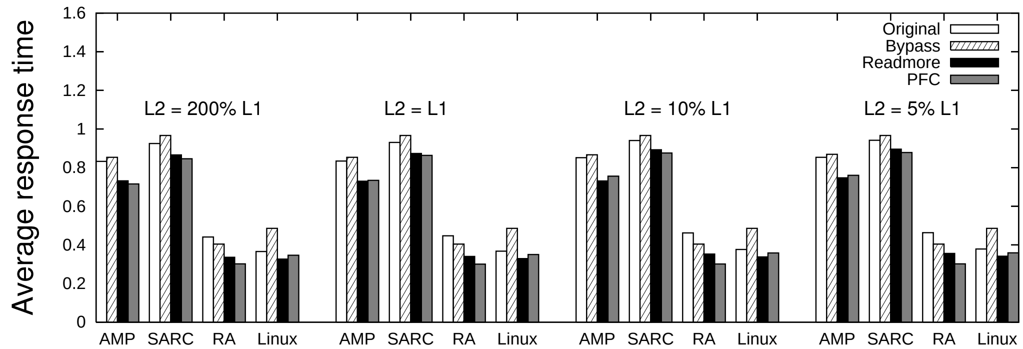


Figure 3.9: Average L2 cache hit ratio

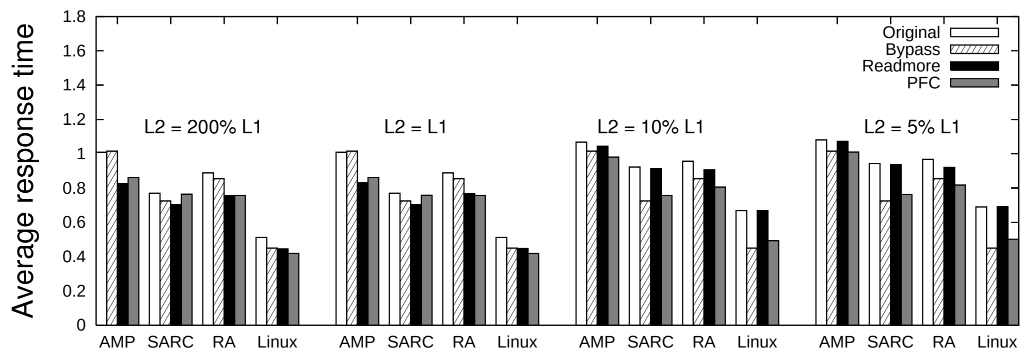
prefetching more aggressive, and as a result, the L2 unused prefetch is increased, so is the L2 cache hit ratio. However, the overall performance is impacted in quite different ways.

For the best case (Fig. 3.8(a)), a sequential trace like OLTP has already caused a high L2 hit ratio (88.9%). But our *readmore_queue* is still able to detect that the RA algorithm is not aggressive enough to catch up with the access rate. Therefore it adds *readmore* blocks and brings a 5.5% improvement in hit ratio which translates to a 35% improvement in response time. Fortunately most *readmore* blocks are used because of the sequential nature of the OLTP trace, so the amount of unused prefetch almost remains the same. For the worst case (Fig. 3.8(b)), PFC goes more aggressive compared with the best case. Although the L2 hit ratio is improved by nearly 20%, both the unused prefetch and the amount of disk request are increased significantly. As a result, the improvement in response time is much smaller than in the best case.

Such case studies reveal the fact that the impact of PFC on the L2 cache hit ratio can be far away from that on the overall system performance. To illustrate this, in Figure 3.9 we summarize the differences in L2 hit ratio with or without PFC, by showing the average L2 hit ratio for each trace-algorithm combination. Actually, for about half of the cases, PFC reduces (sometimes quite significantly) the L2 hit ratio, while achieving an overall performance gain. Such results agree with a previous finding by other researchers that when combined with prefetching, the cache hit ratio is no longer a reliable indication of the system performance [14]. Our observation is the deviation between the two is much more evident in a multi-level system.



(a) OLTP



(b) Web

Figure 3.10: Effect of combining the bypass and the readmore actions

Impact of individual PFC actions Finally, we examine the necessity of having both the "bypass" and "readmore" actions in PFC, using the OLTP and the Web traces. Figure 3.10 demonstrates the effect of enabling the bypass or the readmore action only. In the majority of the cases, combining the two counter-acting operations, PFC obtains a better performance gain than applying a single action only. One notable exception is for the AMP algorithm, where "readmore only" consistently outperforms the full PFC. This indicates that PFC is not prefetching aggressively enough for AMP, which agrees with our previous analysis.

3.7 Summary

In this chapter, we analyzed the multi-level prefetching problem and presented two coordination mechanisms, including an information sharing method and PFC, a hierarchy-aware optimization. They improve the performance of *existing* single-level prefetching algorithms when they are applied to multi-level systems. In particupar, PFC automatically and dynamically adapts to the higher-level access pattern and the lower-level cache status, and controls the aggressiveness of the lower-level prefetching. PFC makes no assumption on the native prefetching algorithm or the application workload, and does not modify the I/O interface between neighboring storage system levels. Our extensive trace-driven simulation with diverse trace workloads and cache configurations demonstrates that PFC can deliver a consistent improvement on the average request response time. In addition, PFC enhances the I/O resource utilization and potentially the system scalability, by regulating the lower-level prefetching and reducing wasted prefetch.

3.8 Future Work

There are several directions to pursue as future work. Some particularly interesting topics include, but are not limited by:

1. Enhancing PFC's decision making to better derive the upper-level prefetching behavior while maintaining its transparency.

2. Controlling the aggressiveness throttling and exclusive caching activities more flexibly and precisely.

3. Extending PFC to work with heterogeneous combinations of prefetching algorithms at multiple levels.

4. Optimizing storage stacks' global memory cache space utilization by adjusting prefetching behaviors of different layers in a coordinated way.

Chapter 4

Coordinated Job Data Staging, Scheduling, Replication and Recovery

4.1 Motivation and Contributions

PetaFlop (PF) computers are looming on the horizon for high-end computing (HEC). Reliability, availability and serviceability are of rising concern with the ever-increasing scale of supercomputer systems. Even with today's machines, fault tolerance is a serious problem. Table 4.1, from a recent study by Department of Energy (DOE) researchers, reports the reliability of several state-of-the-art supercomputers and distributed computing systems [56, 69]. With such frequent system down times, expensive resources are sitting idle while user jobs accumulate in wait queues. Thus, many petascale solicitations from federal agencies are calling for stringent availability requirements, e.g., *when averaged over one month, 90% of jobs submitted to the system should complete without having to be resubmitted as a result of a failure.* [1]

Data and I/O availability are integral to providing a non-stop, continuous computing experience to applications. Table 4.1 indicates that the storage subsystem is consistently

[1] High performance computing system acquisition: Towards a petascale computing environment for science and engineering—NSF program solicitation 06-573.

Table 4.1: Reliability statistics from several large-scale systems, including the top sources of hardware failures (top four lines are from [56]). To this list, we have appended MTBF numbers for the National Leadership Computing Facility at ORNL [69], which is No. 2 [81] in the Top500 [113] supercomputers as of this writing. MTBF/I stands for mean time between failures or interrupts.

System	# Cores	MTBF/I	Outage source
ASCI Q	8,192	6.5 hrs	Storage, CPU
ASCI White	8,192	40 hrs	Storage, CPU
PSC Lemieux	3,016	6.5 hrs	
Google	15,000	20 reboots/day	Storage, memory
ORNL's Cray XT4 "Jaguar" NLCF	23,416	37.5 hrs	Storage, memory

one of the primary sources of failure on large supercomputers. The situation is only likely to worsen in the near future, due to the growing relative size of the storage system forced by two trends: (1) disk performance increases slower than that of CPUs and (2) users' data needs grow faster than does the available compute power [50]. A survey of DOE applications suggests that most applications require a sustained 1 GB/sec I/O throughput for every TeraFlop of peak computing performance. Thus, a PetaFlop computer will require 1 TB/sec of I/O bandwidth. Current disk performance determines that a PetaFlop system needs to have tens of thousands of disk drives and hundreds of I/O nodes, as we can see in the Jaguar system discussed in section 1.1 An I/O subsystem of this size makes a large parallel machine itself and is subject to frequent failures. Trends from commercial and HPC centers suggest, that on average, 3% to 7% of disks, 3% to 16% of controllers and up to 12% of SAN switches fail per year. Further, it is estimated that the above numbers are 10 times the rates expected from vendor specification sheets [48, 49].

Currently, the majority of disk failures are masked by hardware solutions such as RAID [95]. However, it is becoming increasingly difficult for common RAID configurations to hide disk failures as disk capacity is expected to grow by 50% each year, which increases the reconstruction time. The reconstruction time is further prolonged by the "polite" policy adopted by RAID systems to make reconstruction yield to application requests. This causes a RAID group to be more vulnerable to additional disk failures during reconstruction [104].

Moreover, according to recent studies [61], disk failures are only part of the sources causing data unavailability in storage systems. RAID cannot help with storage node failures.

In next-generation supercomputers, thousands or even tens of thousands of I/O nodes will be deployed and will be expected to endure multiple concurrent node failures at any given time. Our experience with Jaguar shows that the majority of whole-system shutdowns are caused by I/O nodes' software failures. Although parallel file systems, such as Lustre [26], provide storage node failover mechanisms, our experience with Jaguar again shows that this configuration might conflict with other system settings. Further, many supercomputing centers hesitate to spend their operations budget on replicating I/O servers and instead purchase more FLOPS.

Another side to data unavailability affecting supercomputer service availability is the data staging and offloading problem. Users typically stage their data, submit their jobs to schedulers and offload result-data, after job execution, for post processing. However, this workflow is mired by numerous issues. First, with users' increasing data needs, input and output data sizes are growing faster than the size of center I/O systems. The upshot is that staging and offloading operations are themselves becoming large data transfer jobs, prone to failure. Second, these activities have to be conjoined with the compute job itself, so the data is available when the job is ready to be scheduled. Otherwise, individual user job turnaround time increases as jobs get requeued due to data unavailability. Further, there is already a significant waiting time at these facilities, especially for large jobs, requesting many nodes. Staging data well in advance is also not a solution as it wastes precious center resources in terms of scratch space, which could be used for running jobs or for storing data of more immediate jobs. In addition, with the likelihood of storage system failure (mentioned earlier), this could very well result in job resubmits. Similarly, delayed offloading from scratch system renders result data vulnerable to mandatory purging, resulting in resubmissions. Thus, data staging/offloading, coupled with transient storage system failure, is unnecessarily degrading supercomputer center performance.

These design constraints are widely recognized. So much so that it is explicitly called out and asked to be addressed in High Performance Computing (HPC) solicitations such as the NSF 06-573 mentioned above. Even so, leadership-class supercomputing facilities are stymied by common pitfalls when it comes to providing seamless data and service availability or reducing resubmission rates. What is needed is a concerted effort towards data staging/offloading, job scheduling and data recovery, which can significantly improve center operations.

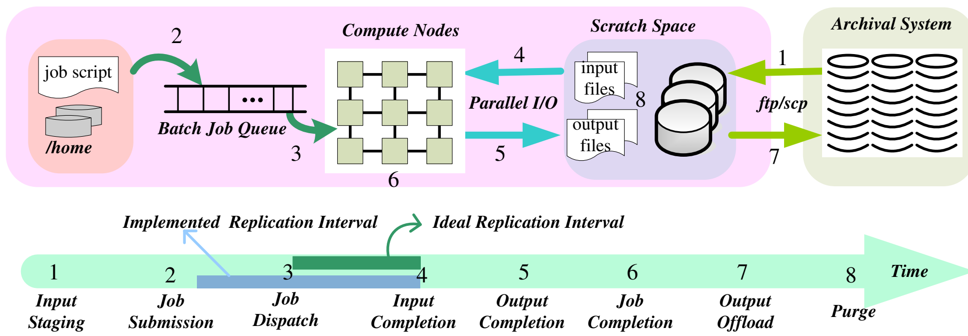


Figure 4.1: Event timeline with ideal and implemented replication intervals

To address the above issues on data availability for supercomputers, we propose and develop a coordinated solution consisting of several novel approaches. Thematic to our solution are two important but previous overlooked recognitions.

Firstly, as illustrated in the event timeline in Figure 4.1, supercomputing job's data is *transient* in nature. Users stage their job input data from elsewhere to the scratch space, submit their jobs using a batch script, and offload the output files to archival systems or local clusters. For better space utilization, the scratch space does not enforce quotas but purges files after a number of days since the last access. Moreover, job input files are often read-only (also read-once) and output files are write-once.

Secondly, supercomputing jobs' data should be protected by coordinating different components along users' workflow, instead of merely within the I/O subsystem in an isolated manner. In the timeline discussed above, job data typically has a remote source/sink which is relatively reliable and immutable. The "active period" of each piece of data is determined by the corresponding job's submission and execution, which are in turn managed by the job scheduler. These two components, in particular, could make the protection of job data much easier and more efficient.

The first part of our solution is to enhance the job scheduler and job scripts with the ability of *automatic scheduling of data staging/offloading activities* so they can better coincide with computing [132]. This approach involves the explicit definition of data staging and offloading in a job script, followed by its decomposition and submission to a zero-charge data transfer queue.

We further propose *temporal file replication*, wherein a parallel file system performs transparent and temporary replication of job input data by checking the job scheduling status and anticipating *future* data accesses [119]. This facilitates fast and easy file reconstruction before and during a job's execution without additional user hints or application modifications. Unlike traditional file replication techniques, which have mainly been designed to improve long-term data persistence and access bandwidth or to lower access latency, the temporal replication scheme targets the enhancement of short-term data availability centered around job executions in supercomputers.

We finally extend the parallel file system to be aware of the remote data source, and be able to transparently and efficiently recover job data lost due to I/O node failures [120, 132]. This approach takes advantage of the existence of an external data source/sink and the immutable nature of job input data.

Collectively, from a center standpoint, our integrated solution globally optimizes resource usage and increases its data and service availability. From a user job standpoint, it reduces job turnaround time and optimizes the usage of allocated time.

4.2 Background

4.2.1 Lustre Parallel File System

Our prototype data recovery system is based on the Lustre parallel file system [26], which is being adopted by several leadership-class supercomputers.

A Lustre cluster comprises of four key components: *Client*s that access and use the data, *MDS* (MetaData Server) that stores and provide metadata information; *OSS*s (Object Storage Server) that provide access to several underlying *OST*s (Object Storage Target); and finally *OST*s that actually manage the storage devices (e.g., RAID storage arrays). In each I/O operation a client firstly queries the MDS for the location of data, and then sends the actually data operation to the OSS/OSSs holding the data, which in turn sends corresponding commands to underlying OST/OSTs.

In contrast to distributed file systems such as the Google File System [42] and IBM's GPFS [103], parallel file systems like Lustre use a more regular data placement method called "striping". A striped file is divided into blocks of equal size, and distributed to a set of storage servers/devices in a *round-robin* fashion. Figure 4.2 illustrates how a 16MB file is striped over 4 OSSs (here the term *stripe count* is used to denote the number

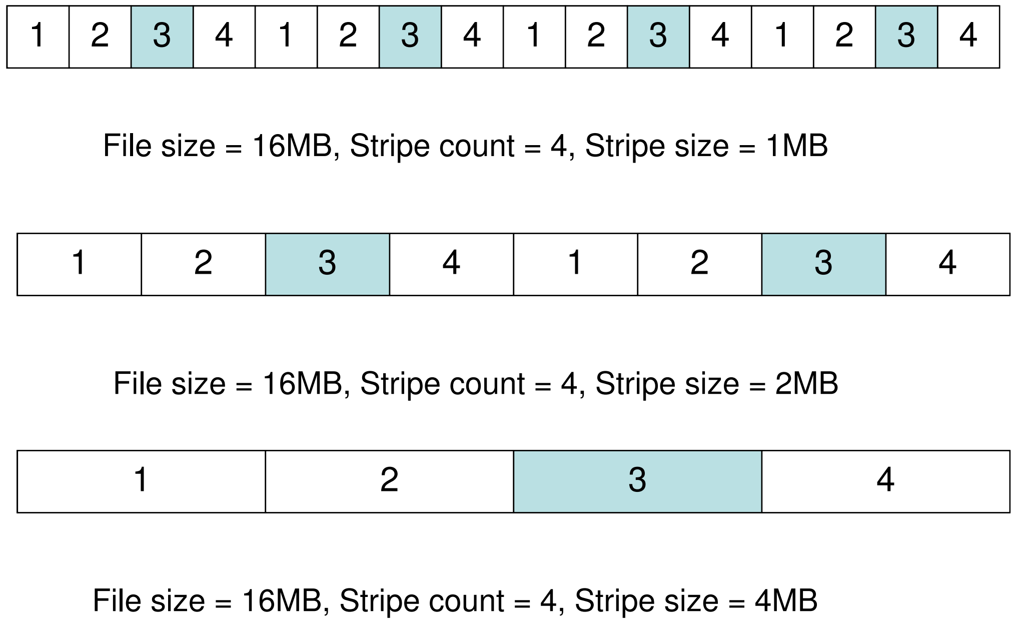


Figure 4.2: Round-robin striping over 4 OSSs

of OSSs in the striping set for a file) with different *stripe size*s, meaning the size of each equal-size block. It can be observed that larger stripe count brings about higher access parallelism (data from OSSs 1, 2, 3 and 4 can be accessed simultaneously), but also causes higher risk of data unavailability due to server failures. Stripe size also affects data access parallelism when only a small number of blocks are accessed. On the other hand, it is also related to data sequentiality on each server and the overhead of managing blocks. In Lustre, a default stripe count of 4 and stripe size of 1MB is used.

In different HEC systems the above software stack can exist in different forms. The Jaguar system [57] as described in Section 1.1 depicts a representative scenario for supercomputing centers, where special-purpose storage "appliances" (DDN S2A 9900 storage couplets) are used as OSTs, while separate servers are used as OSSs and MDSs. In this environment, the connections between OSSs and OSTs are flexibly configured for scalability and fault tolerance purposes. In a smaller cluster, an OSS typically controls a small number of directly attached disks or disk arrays as OSTs. In this case, a crash of the server hosting the OSS and underlying OST devices would cause the corresponding data unavailable to clients.

4.2.2 Job Scheduling

In the scientific computing community, HEC facilities, especially supercomputers, are precious resources and are typically shared among large numbers of users. This not only enables researchers across the world to access to cutting edge computing powers, but also significantly improves the utilization of such computing systems.

Shared accesses to scientific HEC facilities also introduce issues such as efficiency and fairness. Therefore, job scheduling systems are widely used, where users submit their computing programs as *tasks* or *jobs* into *queues*, waiting to be scheduled and executed. In a job a user typically indicates the number of compute nodes required and the expected run time. The scheduling algorithm has many optimization goals: maximize the throughput of the whole system, minimizing the average wait time, minimizing the worst case wait time for each individual job, and so forth. Different policies are used on different positions in the trade-off between these goals, including FCFS (first-come, first-served), SJF (shortest job first), and so on.

Scheduling on the level of jobs is typically non-preemptive. In other words, in "normal" cases a scheduled job remain in the system until it finishes execution. However, when an error occurs to the job, it is killed by the system, and the user would need to resubmit the job, with the suffering of the new wait time.

4.3 Related Work

Coordination of I/O and Computation: Coscheduling of data and computation has been thrust to the forefront due to the increasing need to stage large data at remote computation sites. Stork [66], a scheduler for data placement activities in a grid environment, is closely related to our approach. Stork, along with Condor [77] and DAGMan [30], is used to schedule data and computation together in the face of vagaries in the grid. In [99], the authors, using simulation analysis, argue that data-aware scheduling on the grid improves job response time. Similarly, Condor [77] and SRM [105] are coupled in [101] to schedule jobs on compute nodes, where data is available. However, all of the above are focused as part of an application workflow rather than a set of HPC center integrated services.

BatchAware Distributed File System (BAD-FS [10]) constructs a file system for large, I/O intensive batch jobs on remote clusters. BAD-FS addresses the coordination of storage and computation by exposing distributed file system decisions to an external, workload-aware scheduler. The authors argue that this approach helps data placement, replica creation and reduces data movement in wide-area networks. While the common goal is the coordination of jobs and their data, our approach is fundamentally different from this work. We attempt to inherently improve and optimize the job workflow in HPC environments without creating a new file system.

The importance of the timely offloading of result data, from execution sites to their destinations, is emphasized in IBP [97] and Kangaroo [111]. Both provide an offloading mechanism based on intermediary node staging. While IBP and Kangaroo address the scratch space purging problem by timely offloading of result data, they do not address the scheduling or coupling of this activity alongside computation. Moreover, these solutions are not yet prevalent in HPC settings and users still use simple transfer tools such as hsi [47], ftp or scp. Our research, on the other hand, addresses the scheduling of offloading alongside computation and, in addition, works with existing user tools.

Commercial solutions for cluster and supercomputer schedulers (such as Moab [83]) are beginning to realize the importance of the data staging requirement for jobs and its effect on job turnaround time. Moab provides the ability to set staging requirements on jobs such that the computation does not commence until staging is complete. This is similar to our approach. However, for these solutions to work, it is required that a storage manager be associated with the resource manager for the scheduler to pass the commands along. Moreover, it does not support offloading to the end user site. Further, unlike our approach, there is also no support for staged data availability (e.g., retries or recovery).

RAID: RAID [95] reconstruction is a commonly used technique to sustain access to data and recover redundancy levels in the presence of disk failures. Even staged, transient data on supercomputer parallel file systems are likely to be stored on RAID systems. However, RAID recovery is insufficient for the following reasons. First, even when hot spare disks are available for RAID reconstruction to kick in automatically, the latency of such native reconstruction methods is high (reconstruction time for a 160-GB disk takes on the order of dozens of minutes.) During this time, the application is subject to storage performance degradation, at best. The reconstruction time is proportional to drive size, load and the number of drives in a RAID group. This time is bound to get worse with storage systems on petascale machines. Further, RAID recovery is impaired when there are controller failures or multiple disk failures within the same group. When hot spares are not available, the reconstruction requires manual intervention by the system administrator and will typically take longer. Second, data reconstruction based on hardware RAID does not help when there are I/O node failures. Petascale systems will likely employ thousands of I/O nodes. There are two classes of parallel file systems, namely those whose architectures are based on I/O nodes/servers managing data on directly attached storage devices (such as PVFS [19] and Lustre [26]) and those with centralized, shared storage devices that are shared by all I/O nodes (such as GPFS [103]). For the former category, node failure implies that a partition of the storage system is unavailable. Since parallel file systems usually stripe datasets for better I/O performance, failure of one node may affect a large portion of user jobs. Moreover, unlike specialized nodes/servers such as metadata servers, token servers, etc., I/O nodes in parallel file systems may not be routinely protected through failover. I/O node failover does not help when the underlying RAID recovery is impaired (as mentioned above) as the data is seldom replicated. Our on-demand reconstruction for job input data mitigates these situations by exploiting the source copies of user job input data.

Replication: Data replication creates and stores redundant copies (*replicas*) of datasets. Various replication techniques have been studied [12, 27, 109, 122] in many distributed file systems [13, 42, 76]. Most existing replication techniques treat all datasets with equal importance and each dataset with static, time-invariant importance when making replication decisions. An intuitive improvement would be to treat datasets with different priorities. To this end, BAD-FS [10] performs selective replication according to a cost-benefit analysis based on the replication costs and the system failure rate. Similar to BAD-FS, our approach also makes on-demand replication decisions. However, our scheme is more "access-aware" rather than "cost-aware". While BAD-FS still creates static replicas, our approach utilizes explicit information from the job scheduler to closely synchronize and limit replication to jobs in execution or soon to be executed.

Erasure coding: Another widely investigated technique is erasure coding [15, 98, 127]. With erasure coding, k parity blocks are encoded into n blocks of source data. When a failure occurs, the whole set of $n+k$ blocks of data can be reconstructed with any n surviving

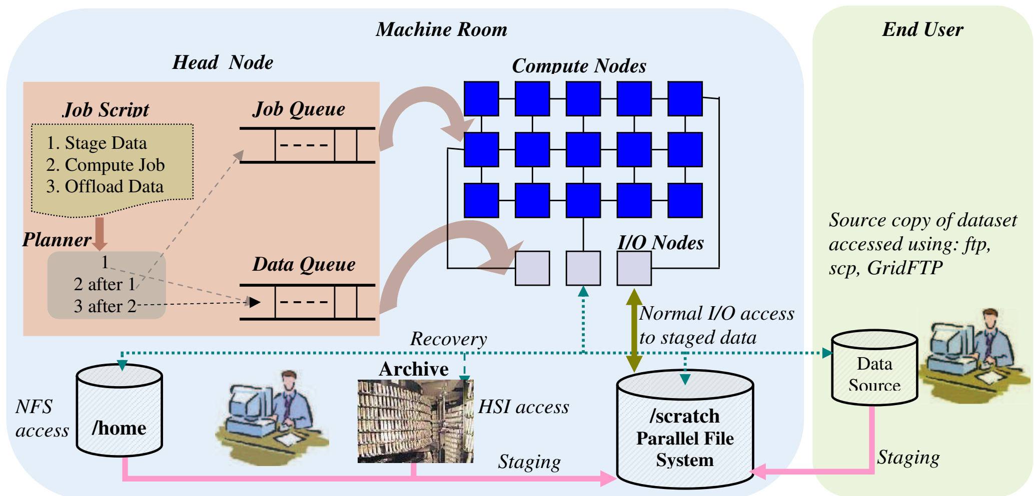


Figure 4.3: Architecture for coordinated data and job orchestration. Also shown are various I/O activities such as staging, regular I/O operations and on-demand recovery from data sources in the machine-room and beyond.

blocks through decoding.

Erasure coding reduces the space usage of replication but adds computational overhead for data encoding/decoding. In [121], the authors provide a theoretical comparison between replication and erasure coding. In many systems, erasure coding provides better overall performance balancing computation costs and space usage. However, for supercomputer centers, its computation costs will be a concern. This is because computing time in supercomputers is a precious commodity. At the same time, our data analysis suggests that the amount of storage space required to replicate data for active jobs is relatively small compared to the total storage footprint. Therefore, compared to erasure coding, our approach is more suitable for supercomputing environments, which is verified by our experimental study.

4.4 Automatic Scheduling of Data Operations

Our approach involves the explicit definition of data staging and offloading in a job script, followed by its decomposition and dependency setup. This infrastructure allows us to schedule and manage data jobs, resulting in better coordination with the computation.

Figure 4.3 presents an overview of our architecture.

4.4.1 Specification of I/O Activities:

HPC users are already acclimatized to using sophisticated data transfer tools for their staging and offloading operations. Many times, these operations are embedded as part of the job script. However, these operations are not explicitly scheduled like compute jobs. As a first step towards the scheduling of data transfers, we have enabled the specification of these operations using "STAGEIN" and "STAGEOUT" directives. These data directives can be embedded within the job script much like resource manager directives (e.g., "#PBS" [88]). Such a specification allows a parser to extract the data staging/offloading commands to compose individual "data jobs" that can then be scheduled. In our implementation, we have instrumented a standard PBS job script [88]. Below is a simple example of a job script with data directives to stage-in data from the HPSS archival system [28] (using the hsi interface [47]) and to stage-out data (using scp).

```
#PBS -N myjob
#PBS -l nodes=128, walltime=12:00
#STAGEIN any parameters here
#STAGEIN -retry 2
#STAGEIN hsi -q -A keytab -k my_keytab_file -l user\
``get /scratch/user/destination_file :  input_file''
mpirun -np 128 ~/programs/myapp
#STAGEOUT any parameters here
#STAGEOUT scp /scratch/user/output/ user@destination_host:/home/user/
```

Of course, more complex staging and offloading logic can be constructed as long they are prefixed with the appropriate directive tags. Staging and offloading can also specify retry attempts with the "-retry" option. For a compute job several other parameters such as expected run time, priority, etc., are specified in the job script as resource manager directives. Similar parameters can also be specified for a data activity so they can be propagated to the resource manager.

4.4.2 "Zero Charge" Data Transfer Queue:

Fundamental to the scheduling of data staging and offloading activities is the treatment of these operations as “data jobs”. To achieve this, we propose establishing a separate queue (“dataxfer”) for data stage-in and stage-out jobs (Figure 4.3). Doing so enables scheduler control over the staging and offloading activities. It is common practice for a supercomputer center to setup multiple queues to manage the diversity in user jobs. For instance, a computer center might setup different queues to handle all high-priority jobs, high throughput jobs (long running) or jobs requiring access to specific resources such as tapes. Multiple queues, in general, help service user jobs better. In our case, setting up separate queues for data and compute jobs is a step towards streamlining data operations by having them explicitly scheduled and managed. Further, it allows better coordination of staging and computing activities.

In center operations, data transfers are not currently charged (the network and I/O bandwidth are not chargeable resources yet), which is also the reason why manual staging operations incur no cost. Storage allocations, on the other hand, are beginning to be charged. However, if this is so, it is a cost that users need to pay in any case. Now, the center can setup a “zero charge” data queue, to which staging and offloading jobs can be queued. The data queue should be scheduled based on the same scheduling policy used to manage the compute queue. For instance, if the compute queue uses priority-based scheduling, it makes sense for the data of the high priority job to arrive before the data for other jobs. Our approach enables this since we propagate parameters about data jobs from the job script to the resource manager. Finally, queuing up and scheduling data jobs separately also has the added benefit that these activities could be charged, if need be, in the future.

To ensure that the dataxfer queue is not used by users for other computational jobs, we can limit access to the queue based on ACLs, which are supported by modern resource managers [83]. For instance, queue attributes such as host-based ACLs can be used to control the machines that can submit jobs to the dataxfer queue. Based on this, the scheduler on the head node can setup an ssh connection to a data job submission node (or I/O node) to submit stag-in and stage-out jobs to the dataxfer queue.

4.4.3 Planning and Orchestration:

We now have a way to specify and distinguish I/O activities such as staging and offloading alongside job scripts. Further, we have a dedicated data queue accepting transfer jobs. The next step is to devise a "plan" for execution. To this end, we have built a planner that accepts a PBS job script (shown above) and decomposes it into computing, staging and offloading jobs and their associated dependencies (see Figure 4.3). Let us say the compute, stage-in and stage-out jobs are stagin.pbs, compute.pbs and stageout.pbs, respectively. The stagin.pbs looks as follows:

```
#PBS any parameters here
retry looping construct here...
hsi -q -A keytab -k my_keytab_file -l user\
‘‘get /scratch/user/destination_file :  input_file’’
```

The stagout.pbs looks as follows:

```
#PBS any parameters here
scp /scratch/user/output/ user@destination_host:/home/user/
```

The compute.pbs looks as follows:

```
#PBS -N myjob
#PBS -l nodes=128, walltime=12:00
mpirun -np 128 ~/programs/myapp
```

The planner then sets up dependencies between these jobs and submits them to their respective queues using standard resource manager primitives. For example, an execution plan for the above three jobs might comprise the following steps.

```
staginJobid=‘qsub -q dataxfer stagein.pbs‘
computeJobid=‘qsub -W depend=afterok:$staginJobid -q batch compute.pbs‘
qsub -W depend=afterok:$computeJobid -q dataxfer stageout.pbs
```

Here, we have used standard PBS primitives to set up dependencies and have used "qsub" to submit the jobs to the respective queues (batch and dataxfer). By explicitly specifying I/O activities and scheduling them along side compute jobs, we have ensured that the data will have arrived when the job is ready to be scheduled. Likewise, upon job completion, the offload data job is scheduled to transfer the data to its destination, thereby avoiding getting purged, waiting for manual intervention. Note, however, that our approach only ensures the timely initiation of the staging activity and cannot guarantee its completion due to other end-to-end issues (network vagaries and other storage issues).

However, we do support retries of staging and offloading. If the staging activity failed, the compute job will not be scheduled. This way, the user, at least, does not waste his compute time waiting for data to become available.

The scheduling of both data and compute operations poses the question: When should a job in the data queue be executed to enable "just-in-time" staging? This can be addressed by having estimates of queue wait times (using tools like NWS [125]) at the time of job submission and data transfer times, both of which are used as a measure to decide when to initiate staging. To start with, data transfer estimates can be specified by the user in much the same way estimated job run times are specified. Estimates of data transfer times can also be obtained between any two sites using transfer predictions (our earlier work [116]). This way, we can more closely tie the staging with the compute job. This also reduces the early stagein of the dataset and its associated pitfalls. However, this is beyond the scope of this dissertation and will be addressed in future work. Finally, to address the potential unavailability of prestaged data, due to storage system failure, before the job is scheduled, we have developed recovery stubs that perform data reconstruction if necessary (discussed in the next section). Since these recovery stubs are a means to cope with center failure, they will incur no cost to users' allocated time. Thus, the coordinated scheduling helps improve center performance by mitigating the detrimental effects of failure.

4.5 Temporal File Replication

Supercomputers are heavily utilized. Most jobs spend significantly more time waiting in the batch queue than actually executing. The popularity of a new system ramps up as it goes towards its prime time. For example, from the 3-year Jaguar job logs, the average job wait-time:run-time ratio increases from 0.94 in 2005, to 2.86 in 2006, and 3.84 in 2007.

4.5.1 Justification and Design Rationale

A key concern about the feasibility of temporal replication is the potential space and I/O overhead replication might incur. However, we argue that by replicating selected "active files" during their "active periods", we are only replicating a small fraction of the files residing in the scratch space at any given time. To estimate the extra space requirement, we

Table 4.2: Configurations of top five supercomputers as of 06/2008

System	# Cores	Aggregate Memory (TB)	Scratch Space (TB)	Memory to Storage Ratio	Top 500 Rank
RoadRunner(LANL)	122400	98	2048	4.8%	1
BlueGene/L(LLNL)	106496	73.7	1900	3.8%	2
BlueGene/P(Argonne)	163840	80	1126	7.1%	3
Ranger(TACC)	62976	123	1802	6.8%	4
Jaguar(ORNL)	23412	46.8	600	7.8%	5

examined the sizes of the aggregate memory space and the scratch space on state-of-the-art supercomputers. The premise is that with today's massively parallel machines and with the increasing performance gap between memory and disk accesses, batch applications are seldom out-of-core. This also agrees with our observed memory use pattern on Jaguar (see below). Parallel codes typically perform input at the beginning of a run to initialize the simulation or to read in databases for parallel queries. Therefore, the aggregate memory size gives a bound for the total input data size of active jobs. By comparing this estimate with the scratch space size, we can assess the relative overhead of temporal replication.

Table 4.2 summarizes such information for the top five supercomputers [113]. We see that the memory-to-storage ratio is less than 8%. Detailed job logs with per-job peak memory usage indicate that the above approximation of using the aggregate memory size significantly overestimates the actual memory use (discussed later in this subsection). While the memory-to-storage ratio provides a rough estimation of the replication overhead, in reality, however, a number of other factors need to be considered. First, when analyzing the storage space overhead, queued jobs' input files cannot be ignored, since their aggregate size can be even larger than that of running jobs. In the following sections, we propose additional optimizations to shorten the lifespan of replicas. Second, when analyzing the bandwidth overhead, the frequency of replication should be taken into account. Jaguar's job logs show an average job run time of around 1000 seconds and an average aggregate memory usage of 31.8 GB, which leads to a bandwidth consumption of less than 0.1% of Jaguar's total capacity of 284 GB/s. For this reason, we primarily focus on the space overhead in the following discussions.

Next, we discuss a supercomputer's usage scenarios and configuration in more detail to justify the use of replication to improve job input data availability.

Even though replication is a widely used approach in many distributed file system implementations, it is seldom adopted in supercomputer storage systems. In fact, many popular high-performance parallel file systems (e.g., Lustre and PVFS) do not even support replication, mainly due to space concerns. The capacity of the scratch space is important in (1) allowing job files to remain for a reasonable amount of time (days rather than hours), avoiding the loss of precious job input/output data, and (2) allowing giant "hero" jobs to have enough space to generate their output. Blindly replicating all files, even just once, would reduce the effective scratch capacity to half of its original size.

Temporal replication addresses the above concern by leveraging job execution information from the batch scheduler. This allows it to only replicate a small fraction of "active files" in the scratch space by letting the "replication window" slide as jobs flow through the batch queue. Temporal replication is further motivated by several ongoing trends in supercomputer configurations and job behavior. First, as mentioned earlier, Table 4.2 shows that the memory to scratch space ratio of the top 5 supercomputers is relatively low. Second, it is rather rare for parallel jobs on these machines to fully consume the available physical memory on each node. A job may complete in shorter time on a larger number of nodes due to the division of workload and data, resulting in lower per-node memory requirements at a comparable time-node charge. Figure 4.4 shows the per-node memory usage of both *running* and *queued* jobs over one month on the ORNL Jaguar system. It backs our hypothesis that jobs tend to be in-core, with their aggregate peak memory usage providing an upper bound for their total input size. We also found the actual aggregate memory usage averaged over the 300 sample points to be significantly below the total amount of memory available shown in Table 4.2: 31.8 GB for running jobs and 49.5 GB for queued jobs.

4.5.2 Delayed Replica Creation

Based on the above observations about job wait times and cost/benefit trade-offs for replication in storage space, we propose the following design of an HPC-centric file replication mechanism.

When jobs spend a significant amount of time waiting, replicating their input files (either at stage-in or submission time) wastes storage space. Instead, a parallel file system can obtain the current queue status and determine a *replication trigger point* to create replicas for a given job. The premise here is to have enough jobs near the top of the queue, stocked up with their replicas, such that jobs dispatched next will have extra input data redundancy. Additional replication will be triggered by job completion events, which usually result in the dispatch of one or more jobs from the queue. Since jobs are seldom interdependent, we expect that supplementing a modest prefix of the queued jobs with a second replica of their input will be sufficient. Only one copy of a job's input data will be available before its replication trigger point. However, this primary copy can be protected with periodic availability checks and remote data recovery techniques previously developed and deployed by us [132].

Completion of a large job is challenging as it can activate many waiting jobs requiring instant replication of multiple datasets. As a solution, we propose to query the queue status from the job scheduler. Let the replication window, w, be the length of the prefix of jobs at the head of the queue that should have their replicas ready. w should be the smallest integer such that:

$$\sum_{i=0}^{w} |Q_i| > max(R, \alpha S),$$

where $|Q_i|$ is the number of nodes requested by the ith ranked job in the queue, R is the number of nodes used by the largest running job, S is the total number of nodes in the system, and the factor $\alpha(0 \leq \alpha)$ is a controllable parameter to determine the eagerness of replication.

One problem with the above approach is that job queues are quite dynamic as strategies such as backfilling are typically used with an FCFS or FCFS-with-priority scheduling policy. Therefore, jobs do not necessarily stay in the queue in their arrival order. In particular, jobs that require a small number of nodes are likely to move ahead faster. To address this, we augment the above replication window selection with a "shortcut" approach and define a threshold T, $0 \leq T \leq 1$. Jobs that request $T \cdot S$ nodes will have their input data replicated immediately regardless of the current replica window. This approach allows jobs that tend to be scheduled quickly to enjoy early replica creation.

4.5.3 Eager Replica Removal

We can also shorten the replicas' life span by removing the extra copy once we know it is not needed. A relatively safe approach is to perform the removal at job completion time. Although users sometimes submit additional jobs using the same input data, the primary data copy will again be protected with our offline availability check and recovery [132]. Further, subsequent jobs will also trigger replication as they progress toward the head of the job queue.

Overall, we recognize that the input files for most in-core parallel jobs are read at the beginning of job execution and never re-accessed thereafter. Hence, we have designed an *eager replica removal* strategy that removes the extra replica once the replicated file has been closed by the application. This significantly shortens the replication duration, especially for long-running jobs. Such an aggressive removal policy may subject input files to a higher risk in the rare case of a subsequent access further down in its execution. However, we argue that reduced space requirements for the more common case outweigh this risk.

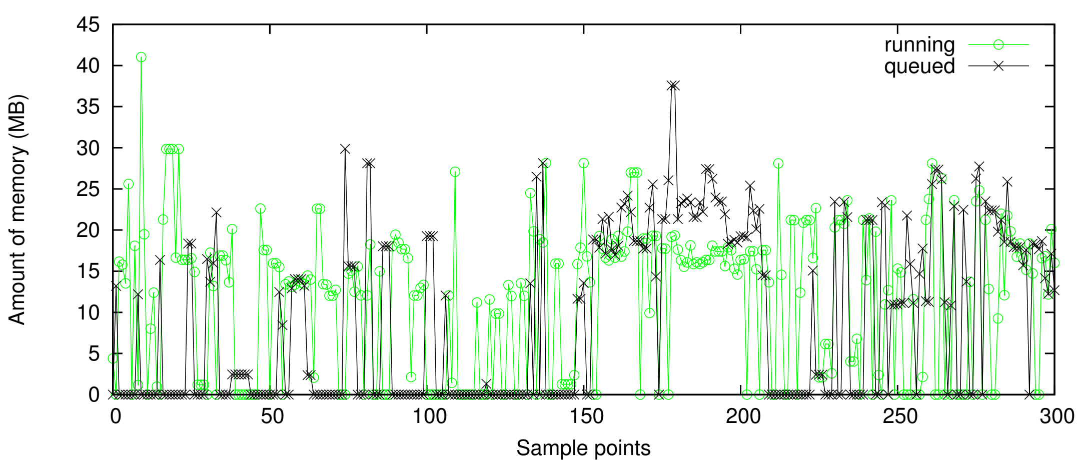


Figure 4.4: Per-node memory usage from 300 uniformly sampled time points over a 30-day period based on job logs from the ORNL Jaguar system. For each time point, the total memory usage is the sum of peak memory used by all jobs in question.

4.5.4 Implementation of Temporary Replication

As we mentioned in Section 4.1, our temporal replication mechanism is required to be coordinated with the batch job scheduler to achieve selective protection for "active" data.

In our target framework, batch jobs are submitted to a *submission manager* that parses the scripts, recognizes and records input data sets for each job, and creates corresponding replication operations at the appropriate time.

To this end, we extend our enhancement to the job scheduler described in Section 4.4 by setting up a separate queue, "ReplicaQueue", that accepts replication jobs. We have also implemented a *replication daemon* that determines "what and when to replicate". The replication daemon creates a new replication job in the ReplicaQueue so that it completes in time for the job to have another copy of the data when it is ready to run. The daemon periodically monitors the batch queue status using the *qstat* tool and executes the delayed replica creation algorithm described in Section 4.5.2. These strategies enable the coordination between the job scheduler and the storage system, which allows data replication only for the desired window during the corresponding job's life cycle on a supercomputer.

4.6 Transparent Data Recovery

Although the coordinated staging and temporary replication approaches described in previous sections reduce the vulnerability of job data, it is still possible that pre-staged data is unavailable before or during a job's execution due to I/O node failures. Furthermore, they modify the job scheduler to setup special data operation queues, which requires special permissions in some systems. Therefore, in this section we present a *transparent data recovery* framework which complements the above approaches by capturing I/O failures and efficiently recovering any file with remote sources.

To start with, this mode of recovery is made feasible due to the transient nature of job data and the fact that they have immutable, persistent copies elsewhere. Next, many high-performance data transfer tools and protocols (such as hsi [47] and GridFTP [11]) support partial data retrieval, with which disjoint segments of missing file data—specified by pairs of start offset and extent—can be retrieved. Finally, the costs of network transfer is decreasing much faster than that of disk-to-disk copy. Collectively, these reasons enable and favor on-demand recovery.

A staged job input data on the scratch parallel file system may have originated from the "/home" area or an HPSS archive [28] in the supercomputer center itself or from an end-user site. To achieve proactive data reconstruction, however, we need rich metadata about data source and transfer protocols used for staging. In addition, we need sophisticated recovery mechanisms built into parallel file systems. In the rest of this section, we will describe automatic recovery metadata extraction, failure detection and the instrumentation of a parallel file system as an instance of how to achieve this on-demand data recovery.

4.6.1 Extending Metadata for Recovery Hints

To enable on-demand data recovery, we first propose to extend the parallel file system's metadata with *recovery information*, which can be intelligently used to improve fault tolerance and data/resource availability. Staged input data has persistent origins. Source data locations, as well as information regarding the corresponding data movement protocols, can be recorded as optional recovery metadata (using the extended attributes feature) on file systems. For instance, the location can be specified as a uniform resource index (URI) of the dataset, comprised of the protocol, URL, port and path (e.g., "http://source1/StagedInput" or "gsiftp://mirror/StagedInput"). In addition to URIs, user credentials, such as GSI (Grid Security Infrastructure) certificates, needed to access the particular dataset from remote mirrors can also be included as metadata so that data recovery can be initiated on behalf of the user. Simple file system interface extensions (such as those using extended attributes) would allow the capture of this metadata. We have built mechanisms for the recovery metadata to be automatically stripped from a job submission script's staging/offloading commands, facilitating transparent data recovery. One advantage of embedding such recovery-related information in file system metadata is that the description of a user job's data "source" and "sink" becomes an integral part of the transient dataset on the supercomputer while it executes. This allows the file system to recover elegantly without manual intervention from the end users or significant code modification.

4.6.2 Failure Detection

The next step toward failure recovery is to have in-time detection of storage and I/O failures. We detect failures both before and during a job's execution.

Pre-execution Detection: To detect the failure of an OST, we use the corresponding Lustre feature with some extensions. Instead of checking the OSTs sequentially, we issue commands to check each OST in parallel. If no OST has failed, the probe returns quickly.

Otherwise, we wait for a Lustre-prescribed timeout of 25 seconds. This way, we identify all failed OSTs at once. This check is performed from the head node, where the scheduler also runs.

On-the-fly Detection: To access the data of a file stored in the OST, the application issues calls *via* the standard POSIX file system API. The POSIX API is intercepted by the Lustre patched VFS system calls. Due to the fail-out mode, both I/O node and data disk failures will lead to an immediate I/O error at the client upon file access (steps 3 and 4). By capturing the I/O error in the system function, we obtain file name and index of the failed OST or, in case of a disk failure, the location of the affected OST. In step 5, the client sends relevant information (file name, OST index) to the head node, which, in turn, initiates the data reconstruction. Hence, we perform online/real-time failure detection at the client for on-the-fly recovery during application execution, much in contrast to prior work on offline recovery that dealt with data loss prior to job activation [132].

4.6.3 Data Reconstruction

The first step towards reconstruction is to update the stripe information of the target file. It is maintained in the form of a "stripe list" of OSTs being used, as part of the metadata. In the process of finding another OST to substitute the failed one, efforts are made to pick a distinct set of OSTs for each file to sustain parallel I/O performance. The recovery manager also need to take care of the client side caching of the stripe list, which is obtained when a file is opened and used for subsequent read/write calls. This is done by the client that locates the failed OST through sending a message to the MDS to trigger an update of the master copy. The MDS also associates a dirty bit with the file, indicating the availability of updated data to other clients.

When the new stripe list is generated and distributed, storage space needs to be allocated on the substitute OST. As an object-based file system [80], Lustre uses objects as storage containers on OSTs. The process of allocating objects on the substitute OST is similar to that of creating a new file, except that the object IDs from the failed OST are used in order to correctly redirect requests to the stored data.

Finally the recovery manager obtains the byte ranges of lost data by generating an array of $\{offset, size\}$ pairs according to the total number of OSTs used by the target file, the position of the failed OST in the stripe list, and the stripe size. Using this information and the URIs to remote permanent copies of the file recorded as part of file metadata, it establishes a connection to the remote data source, using the protocol specified in the URI, to patch each chunk of data in the array of $\{offset, size\}$ pairs. These are then written to the object on the substitute OST.

4.7 Performance Evaluation

4.7.1 Experimental Setup

As discussed in earlier sections of this chapter, our solution of protecting supercomputing job data consists of several components, each involving certain data operations within the parallel file system or over the network, such as failure detection, data object creation, remote data patching and so forth. The overheads of performing individual data operations are assessed on real clusters in [120, 119, 132] and summarized in [118]. In conclusion, two testbeds are used in these experiments, a 40-node cluster at Oak Ridge National Laboratory (ORNL) and a 17-node cluster at North Carolina State University (NCSU). Each cluster has Linux servers with Lustre-patched kernel and Lustre 1.4.7 installed. With the focus on testing the server-side of Lustre, majority of the the cluster nodes are setup as I/O servers. In practice, such a group of Lustre servers will be able to support a fairly large cluster with 16:1-64:1 ratios seen in contemporary supercomputers. Three data repositories with different network latencies, connection bandwidths and partial data access characteristics are used as sources to patch pieces of a staged input file, including an NFS server at ORNL, an NFS server at NCSU and a GridFTP [11] server with a PVFS [19] back-end.

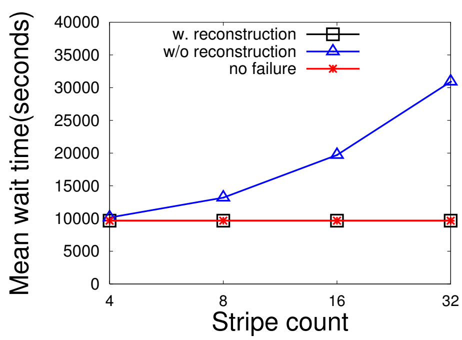
In this dissertation, we take these benchmarking results as input to a simulation study to evaluate the impact of our approach on the overall center performance in terms of average job wait time and wait time variances. The simulation also uses job submission/completion traces and job failure information collected at large supercomputer centers. In the rest of this section we present simulation results of different components of our solution. Since the benefits of automatic scheduling of data operations are mainly on facilitating users' operations and enabling data recovery from remote sources, we evaluate and present results of data scheduling coupled with pre-execution data recovery, followed by temporary data replication and on-the-fly data recovery.

4.7.2 Data Scheduling and Pre-execution Recovery Results

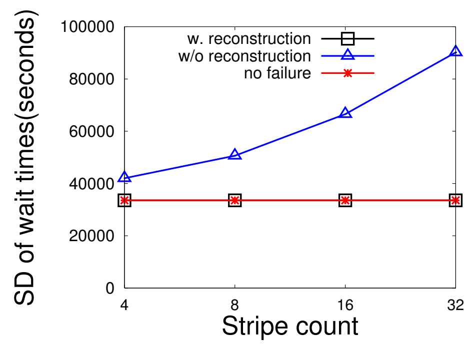
To evaluate the effect of our scheduling and recovery mechanism on the overall performance of a supercomputing center, we perform trace-driven simulations based on the operational data released from Los Alamos National Laboratory [68]. This trace contains failure records for 24 HPC systems and job submission/completion records for 5 of them. Out of these 5 systems, system 20 is the largest one with 512 nodes and 4 CPUs per node. Therefore, we chose to base our simulations on this system. Two traces from system 20 are used as input to our simulator. First is the *node failure trace* from system 20, which contains 2049 failure records over a period of 1349 days, each indicating specifications of a failure: the node number as well as the failure time and duration. Second is the *job trace*, which contains 489376 job submission and completion records over a period of 1073 days.

Next, we explain how we couple the failure trace with the job trace.First of all, due to a lack of disk failure data, we focus on I/O node failures. These are failures that could not be automatically recovered using schemes such as RAID. Since the traces did not indicate whether I/O nodes were included or which nodes were I/O nodes, we assume all the 512 nodes are compute nodes and append additional nodes as I/O nodes. In most cluster systems, I/O nodes often share the same configuration as compute nodes (except for secondary storage). Therefore we extrapolate the failure statistics observed from the failure trace to these additional I/O nodes. More specifically, system 20's node failure trace is used to calculate the average node failure rate and the average node failure repair time. We use those statistics to generate a set of failure events for each I/O node. Based on the job trace, we generate a set of job submission events, each of which contains the submission time, the execution time and the number of CPUs for a job. For parallel job scheduling, our simulator adopts the FIFO algorithm with backfilling, a popular choice in many supercomputing centers.

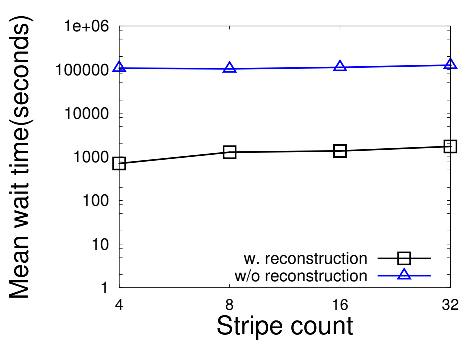
The job trace, however, is devoid of staged data information (e.g., file size, stripe size, stripe count) for each job. To this end, we have obtained a snapshot of the Lustre scratch space from a production supercomputer center at ORNL. This staged data trace contains details of every file staged in the scratch parallel file system. We have calculated the distributions of file sizes, stripe sizes and stripe counts. We play these two traces together, randomly assigning staged data sizes to jobs, and study the average waiting times of jobs in Section 4.7.2.



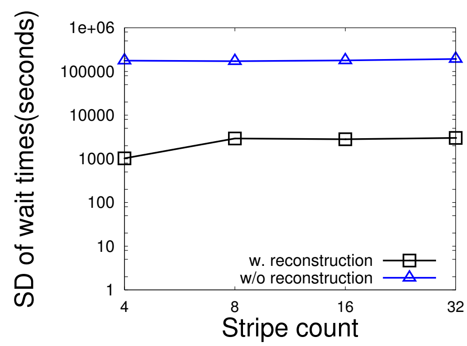
(a) Original trace, mean wait time for all jobs



(b) Original trace, standard deviation (SD) for wait time of all jobs



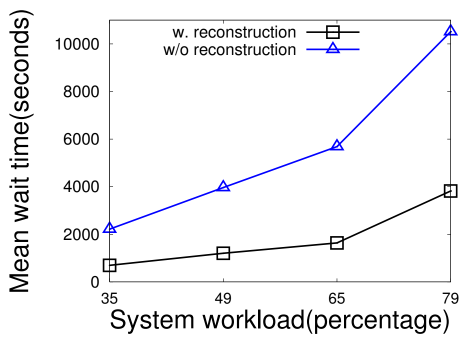
(c) Original trace, mean wait time for affected jobs



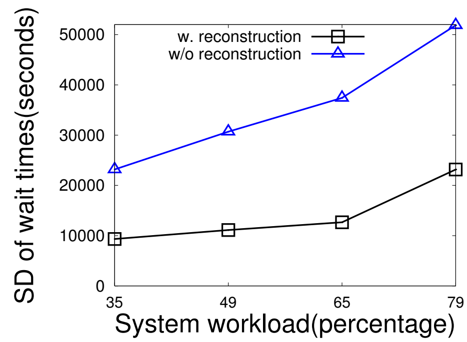
(d) Original trace, standard deviation (SD) for wait time of affected jobs

Figure 4.5: Simulation results with the original trace. Zero-wait jobs are omitted.

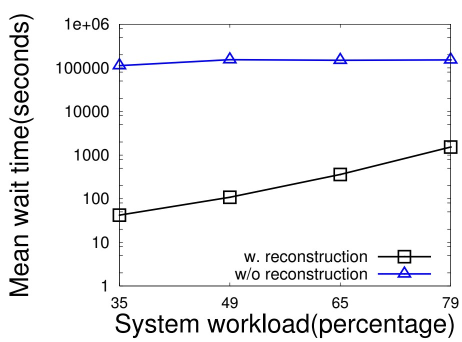
In this section, we describe our results obtained by using the reconstruction numbers as input to trace-driven simulation of job wait times. Our simulations compare system performance with and without the reconstruction mechanism for different stripe counts. We use two metrics to evaluate system performance: the mean value and standard deviation("SD") of wait times of jobs. The SD of wait times is vital to center users because it represents an expected level of service. With our approach, we expect to improve the SD of wait times by eliminating the difference in wait times of failure-afflicted jobs and other successful jobs. Without reconstruction, jobs encountering failure would be re-queued at the end of the queue. On the other hand, with our reconstruction, jobs that encounter an I/O node failure would only spend the cost of reconstruction as additional wait time. Moreover, the reconstruction cost can be hidden by proactive and periodic checking of the



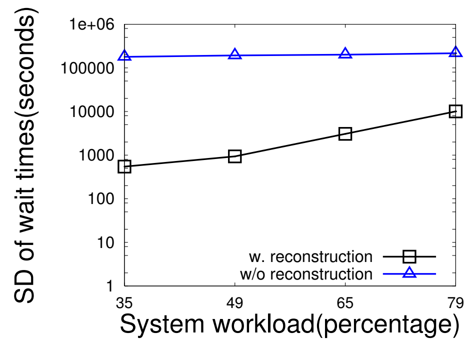
(a) 16 OSTs, mean wait time for all jobs under different system work load



(b) 16 OSTs, standard deviation (SD) of wait times of all jobs under different system work load



(c) 16 OSTs, mean wait time for affected jobs under different system work load



(d) 16 OSTs, standard deviation (SD) of wait times of affected jobs under different system work load

Figure 4.6: Simulation results with varying system work load

status of OSTs that host a job's data.

Our first set of simulation tests use the original LANL system 20 job trace combined with the staged data trace (see Section 4.7.1). In our experiments, we have tested different strategies of assigning a pair of $\{file\_size, stripe\_size\}$ values from the staged data trace to each job (from the LANL trace). These include using fixed values for all jobs, using larger file sizes and stripe sizes for longer running jobs, and randomly picking a pair of values for each job. However, we find no observable difference between the strategies. The reason being, file sizes and stripe sizes only impact the reconstruction cost, which is hidden from the affected job in most cases due to our approach of proactive checking of the file's availability. Therefore, we show the results of using randomly assigned file sizes and stripe sizes. As for stripe counts in the staged data trace, we find that more than 90% of the files are striped across 4 OSTs, according to the ORNL scratch space trace we obtained. In our simulations, we chose 4, 8, 16 and 32 for the stripe count values.

Figure 4.5 shows the simulation results using the LANL job trace. The job traces follow a binomial distribution of job wait times (*i.e.*, many short and few very long jobs exist in the queue). To address this, we filtered the trace to remove jobs that have a zero wait time under all test configurations. As can be seen from Figure 4.5, the higher the stripe count, the more OSTs the files are associated with, which means an OST failure will affect more jobs. This intuitive trend can be observed for the job schedule without reconstruction. With our new reconstruction mechanism, however, the mean value and SD of wait times of all jobs are not affected by the increase in the stripe count, indicating a scalable solution for potentially very large Lustre server groups. Further, our reconstruction mechanism achieves the same mean value and SD of wait times as the ideal values (without any OST failure) for all stripe count settings (as the two curves are on top of each other).

The second pair of charts in Figure 4.5 show the mean value and SD of wait times for the failure-affected jobs. There are two kinds of affected jobs. First are the scheduled and running jobs that access datasets located on the failed I/O node. These are now interrupted by the failure. Since we do not support online recovery yet, they will be rescheduled. The second set of jobs are yet to be scheduled, but have job input data located on the failed I/O node. If no reconstruction were applied, these jobs would have been moved to the tail of the job queue, waiting for their next turn to be dispatched. With reconstruction, however, they are scheduled immediately. For these affected jobs, the gains due to our new recovery mechanism are significant. The mean wait times are reduced from hundreds of thousands of seconds to tens of seconds.

The figure indicates that the mean value and SD of wait times for the affected jobs increases as the stripe count increases from 4 to 32. As mentioned earlier, a higher stripe count results in more jobs being afflicted due to an OST failure. Of those, affected running jobs that wait longer contribute to the mean wait time more than the affected jobs that are yet to be scheduled.

Also, Figure 4.5 indicates that the mean wait time of affected jobs is much shorter than that of the set of all jobs. However, the mean wait time of the affected jobs should be longer than that of the set of all jobs since they not only incur the same wait time as the normal jobs but they also need to wait for the recovery. One possible reason for these results is that the job submission and failure logs used for the simulation have some special properties that cause affected jobs to have short wait times. One thing to note is that the failure log used for the I/O nodes is correlated to CPU failures, which could have an unintended effect on simulation results. When failures occur in a supercomputing center, typically all or some nodes are shut down and the job queue may refuse new jobs. Therefore, the job submission behavior may be altered as a consequence of center management policies regarding failure handling. As a result, fewer jobs exist in the job log around the failure times recorded, causing short wait times.

In our experiments with system 20's job trace, we find that the system CPU utilization rate is only 35%. To test our mechanism under heavier system work loads, we constructed a series of concatenated job traces based on the original trace by shortening the time interval between job submissions. Figures 4.6(a)-(b) depict the results of the set of all jobs. These results indicate that as the system work load increases, the mean wait time and the standard deviation with our mechanism increases slower than without our mechanism. In other words, the benefit of applying our reconstruction mechanism will be even more significant with heavier system work loads. For the affected jobs shown in Figures 4.6(c)-(d), with heavier system work loads, more running jobs are interrupted by the failures. Consequently, they need much more time to be rescheduled, waiting in the longer queue. Thus, the mean wait time increases from tens of seconds to hundreds of seconds for the affected jobs with reconstruction. However, for the affected jobs without reconstruction, since the base is huge (around 100,000 seconds), the increase is not visible on the curve.

4.7.3 Temporary Replication Results

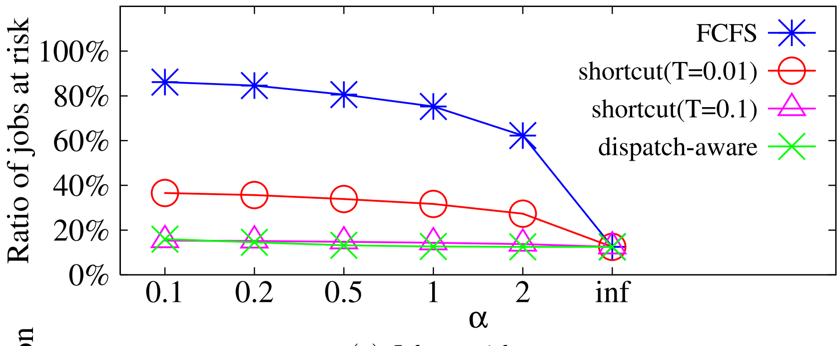
To evaluate the impacts of temporary data replication we use operational data from the ORNL Jaguar supercomputer with job logs collected from 04/2005 to 11/2007. Each job entry contains timing information (such as submission, dispatch, and completion times) and resource usage details (such as the number of cores requested and the peak memory usage per core). Note that for temporary replication we simulate on a different set of job traces as used in the evaluation of scheduling and recovery approaches in this section. Compared with the LANL job traces, Jaguar's job traces have a wider range in time (3 years), but do not have corresponding failure traces. We choose to use Jaguar's job traces because 1) it enables us to study the impact of the long term trends of the ratio between active data and the scratch space; and 2) temporary replication doesn't involve recovery operations at failure time (except for recreating replicas), making it meaningful to illustrate the number of jobs "at risk" without really simulating failures.

Similar to the traces used in Section 4.7.2, the Jaguar job log is devoid of information on job input data size. This metric is needed to determine the amount of storage space required for replicas. However, as discussed in Section 4.5.1 and supported by the memory usage pattern on Jaguar shown in Figure 4.4, we can safely estimate this information based on each job's peak aggregate memory usage.

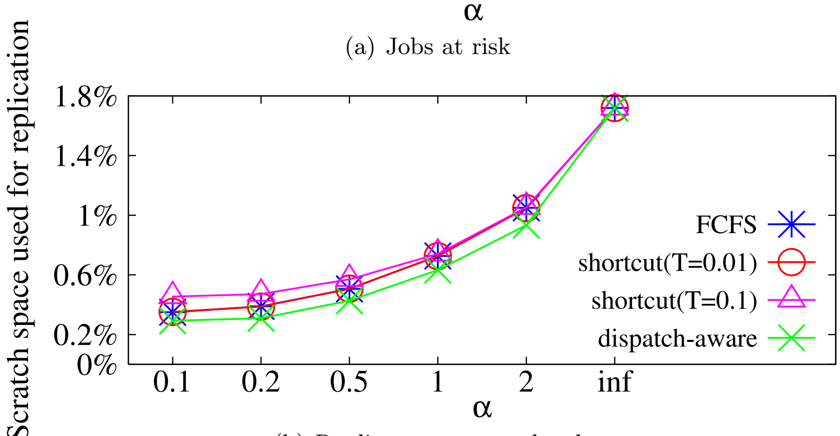
In our experiments, we replayed the job traces and simulated a job queue according to the submission and dispatch times of jobs. Several replication strategies, described in Section 4.5.2, were used to evaluate our dynamic replica creation algorithm to study the balance between enhanced data availability and increased space usage. The first one was FCFS, where the replication window is calculated based on the arrival time of jobs. The next two strategies used the shortcut approach, using a T value of 0.01 (FCFS with small threshold) and 0.1 (FCFS with large threshold), respectively. Finally, as a reference, we used an offline algorithm (dispatch-aware) that was aware of the actual dispatching order of jobs a priori and calculate the replication window accordingly. This algorithm indicates the best decision that can be made with the replication window approach.

Figure 4.7 compares the above algorithms using different α levels. The *alpha* value "inf" corresponds to an infinite replication window size, where all jobs' input datasets are replicated upon submission regardless of the replication strategy (hence the convergence). Figure 4.7(a) shows the percentage of jobs that are "at risk". A job was considered at risk if its replica was not ready when the job was dispatched. The replication time for each input dataset was calculated using the cost of a file copy, benchmarked on Jaguar via the command `cp`. Due to the small default Lustre stripe width, a large number of available OSSs, and the low average number of concurrent replication operations estimated in our simulation, we did not consider the impact of concurrent replications on the copy bandwidth.

With the basic FCFS replication strategy, a large fraction of jobs are at risk with small α levels, and even with a relatively large α level of 2, more than 60% of jobs will not have their replicas created early enough. The offline algorithm (dispatch-aware), in contrast, is highly insensitive to α. This reveals that to have an appropriate level of data



(a) Jobs at risk



(b) Replica storage overhead

Figure 4.7: Simulation results using a 3-year Jaguar log with replication covering jobs up to αS nodes; Infinity(inf) denotes replication for all Jobs.

redundancy, only a small portion of jobs in the queue needs to be selected for replication if we know the exact order of dispatched jobs. The dispatch-aware curve is almost flat due to the existence of jobs that are scheduled immediately upon submission, which leaves no time for data replication. Fortunately, these jobs tend to be debug or testing jobs that request a small number of nodes and run time, making them less vulnerable to failures and requesting a small resubmission cost. With the shortcut enhancement to FCFS, we see a very selective threshold of 0.01 will dramatically reduce the jobs at risk. A threshold of 0.1 will generate a fault tolerance level virtually identical to the offline dispatch-aware algorithm.

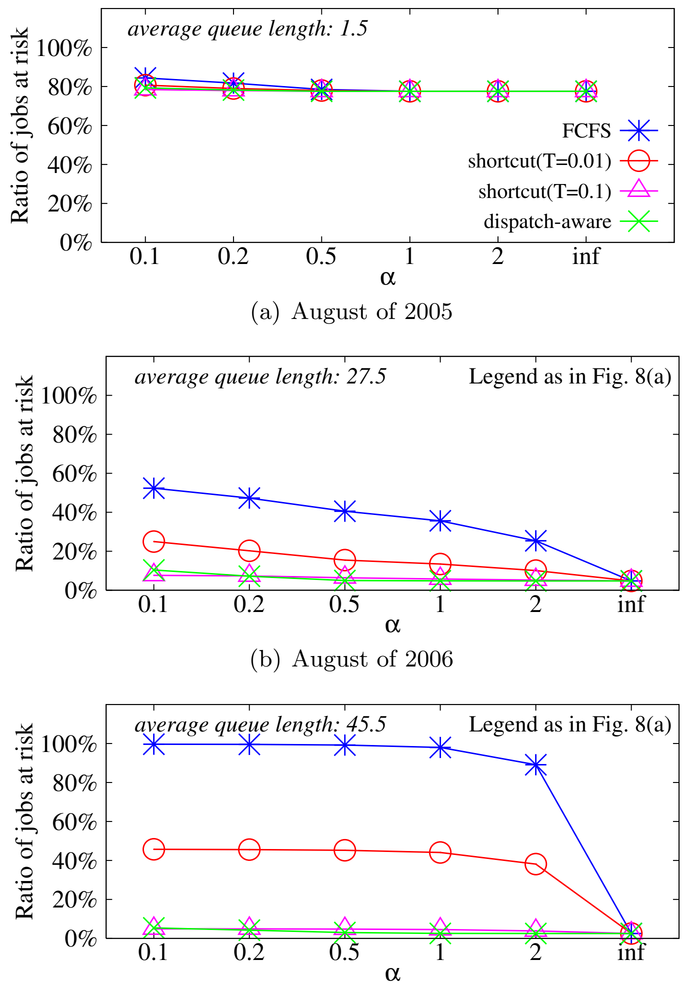
In Figure 4.7(b), we compare the space overhead of the above strategies by calculating the ratio of total scratch space used for storing job input file replicas. This ratio was averaged over 300 snapshots, regularly sampled during the 32-months-long period. For each snapshot, the total replica size was estimated as the sum of the peak aggregate memory usage of all jobs that have input data replicas created, including both running and waiting jobs. The results indicate that, overall, creating one extra copy of active jobs' input data consumes a very limited fraction of the entire scratch space on Jaguar. Even with the most aggressive replication setting (an infinite α level), the replicas will, on average, occupy less than 1.8% of the scratch space. The space overhead significantly reduces as the α value is lowered. The differences between different strategies are quite small since they mainly differ in the handling of small jobs. The dispatch-aware algorithm has an edge due to its optimal job selection.

In summary, as shown in Figure 4.7, we regard that FCFS with small threshold as an effective replication strategy for our traces. A threshold of 0.1 will generate close-to-ideal data redundancy, which, in turn, allows a small replication window of α levels such as 0.2 and 0.5. Such small windows will incur very low space overhead, lower than 0.5% of the scratch space.

Finally, Figure 4.8 demonstrates the behavior of the replication strategies under different job workload levels, by plotting the percentage of jobs at risk for the same month (August) in three consecutive years. As mentioned earlier, most supercomputers tend to get crowded once into production. For our trace, the average job queue length increases from 1.5 in 08/2005, to 27.5 in 08/2006, and to 45.5 in 08/2007. Under a light load, most jobs can be scheduled quickly, without a sufficient enough waiting period required for just-in-time replication regardless of strategies. As the system gets busier, different strategies begin to diverge. While the naive FCFS strategy produces a high at-risk rate, short cut with T=0.1 stays close to the offline dispatch-aware algorithm and is able to replicate most datasets in time. This result further justifies the advantage of temporal replication when the system is busier. A busy system implies longer wait times and consequently longer turnaround time for jobs that are required to be resubmitted due to input data unavailability as they get back at the end of the queue. However, the longer wait time provides enough room for just-in-time replication.

4.7.4 On-the-fly Recovery Results

We use the same traces as in Section 4.7.2 to evaluate the impact of capturing storage failures on-the-fly, i.e., during job execution, and recovering job data transparently from remote sources.



(a) August of 2005

(b) August of 2006

(c) August of 2007

Figure 4.8: The ratios of jobs at risk in the month of August in 2005, 2006 and 2007.

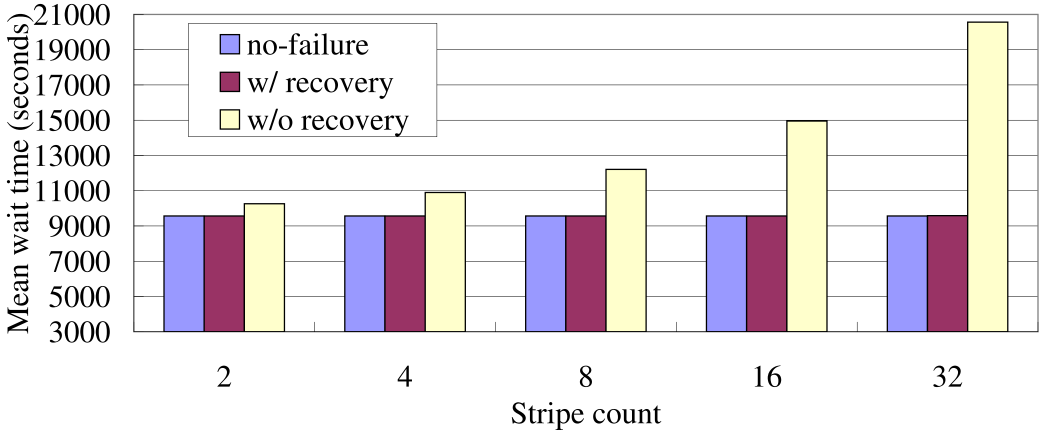


Figure 4.9: Mean wait time for all jobs

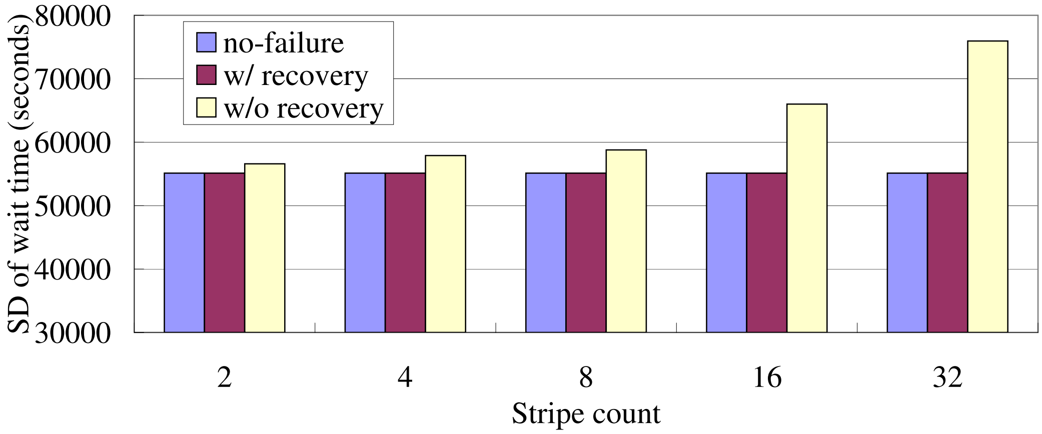


Figure 4.10: Standard deviation (SD) for wait time of all jobs

Our simulations compare system performance with and without recovery for different stripe counts. We use mean and standard deviation (SD) of job wait times to evaluate system performance. Without recovery, if a running job accesses an input file residing on a failed OST, the job exits upon I/O error and is requeued at the queue's tail. Similarly, disk or I/O node failures typically result in job exit before being requeued. With our recovery, jobs will coordinate with the head node to patch missing data and continue to run despite both failure cases.

Figures 4.9 and 4.10 depict the mean and standard deviation (SD) of the wait time for all jobs. Job wait times follow a bimodal distribution with many short and few

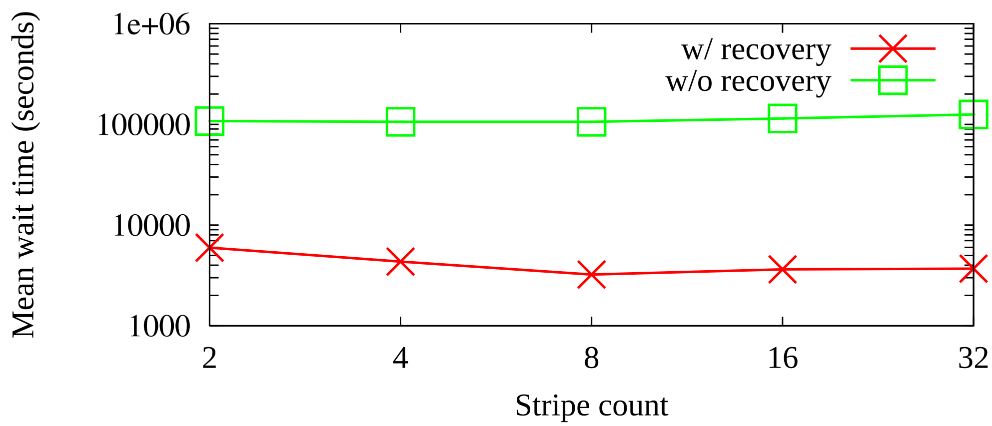


Figure 4.11: Mean wait time for affected jobs

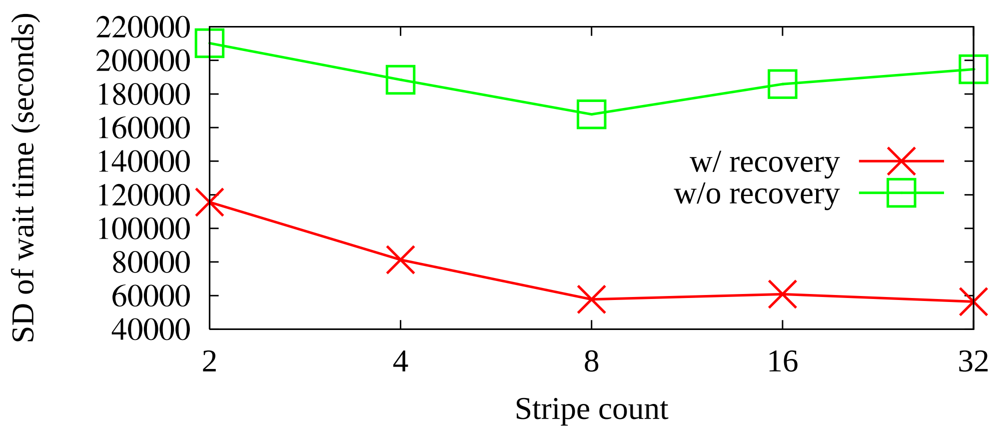


Figure 4.12: SD for wait time of affected jobs

very long jobs. To address this, we filtered results removing jobs that have a zero wait time under all test configurations. The higher the stripe count, the more OSTs the files are associated with. This means an OST failure will affect more jobs. In fact, the percentage of the affected jobs over all jobs increases from 0.14% to 2.09% when the stripe count increases from 2 to 32. This explains the rise of the curve without recovery with increasing stripe counts. With our recovery mechanism, in contrast, the mean and SD of wait times remain constant as stripe counts increase, indicating a scalable solution for potentially very large Lustre server groups. Furthermore, the recovery mechanism results in the same mean and SD of wait times as the ideal case (without any failure) for all stripe counts.

Figures 4.11 and 4.12 show the mean and SD of wait times for those jobs affected by failures. Since these jobs have non-zero wait times without our recovery, no job filtering is applied. Without recovery, each failed job will be requeued. On-the-fly recovery can handle both failure cases, up-front and mid-way, as mentioned previously. Failure-affected jobs result in slightly longer run times but finish without requeuing. For these affected jobs, gains due to on-the-fly recovery are significant. The mean wait times are reduced by more than an order of magnitude from over 100k seconds to thousands of seconds and show a falling trend (less noticeable due to the log-scale y-axis).

Our experiments indicate a system CPU utilization of 70.50% and 70.54% without and with the recovery, respectively. This 0.04% increase is due to the actual recovery of failure-affected jobs. Such an insignificant change can easily be amortized by contemporary HPC systems. This is further reinforced by the observation that under recovery, the same mean and SD of wait times are observed (indicated by the ideal values for all stripe counts).

4.8 Summary

In this chapter, we have identified the need to manage transient job input/output data efficiently and have assessed its direct impact on supercomputer center performance as we approach petascale environments. We have built techniques to schedule data jobs alongside their computational counterparts to allow them to be scheduled and replicated in a synergistic fashion. Further, we have proposed a novel way to reconstruct transient, job input data in the face of storage system failure. Our simulation study of job traces and staged data, supported by data recovery measurements from a real supercomputer, suggests that our technique reduces the average wait time of jobs in a center significantly. In particular, more significant benefits are seen for long jobs and as data is stored over a larger number of I/O nodes. From a user's standpoint, our techniques help reduce the job turnaround time and enable the efficient use of the precious, allocated compute time. From a center's standpoint, our techniques improve serviceability by reducing the rate of resubmissions due to storage system failure and data unavailability. Further, they enable the optimal use of center resources (e.g., precious scratch space) by the timely staging and offloading of job input and output data.

4.9 Future Work

On this topic, it is particularly interesting to investigate the automatic, timely offloading of HPC jobs' result data. A large portion of HPC jobs, especially simulation jobs have large amounts of result data. We plan to design a mechanism to offload the result data once the corresponding files are closed in the parallel file system, so as to save users' time and effort.

Chapter 5

Conclusions

With the rapid advances and growing popularity of supercomputing and cloud computing services, high-end storage systems are playing increasingly important roles in the overall computing experience perceived by users. The author's doctoral research is based on the observation of the key challenges faced by today's high-end storage systems, and the recognition of adding coordination to system management as a fundamental and effective solution to this problem.

Over the past decades, much effort has focused on improving the management of high-end storage systems, and many new mechanisms have been developed and deployed. In fact, many of them adopt the strategy of adding coordination to existing mechanisms based on previously overlooked interactions or interdependencies among system components or requests. Nevertheless, due to the fast development of new techniques and rapid emergence of new trends, many directions remain relatively unexplored.

Memory cache management is crucial to the performance of high-end storage systems with the ever enlarging processor-I/O gap. Chapters 2 and 3 have focused on adding coordination to the management of file system memory cache for data prefetching. In particular, we discovered the interesting connection between this problem and that of supply chain management (SCM). Both problems study how to stock up appropriate amount of items to be consumed in the future, based on predicted consumption rates and patterns. Both chapters explore issues that are relatively well recognized and studied in SCM, but overlooked in prefetching. The overall mapping from prefetching to SCM problems are presented in Chapter 2, which also leverages two rate-aware inventory control methods to develop new prefetching algorithms. Chapter 3, on the other hand, focuses on addressing the issue of information distortion among levels of caches and prefetching algorithms. Instead of new multi-level prefetching algorithms, two coordination mechanisms are proposed and developed, including a simple information sharing method and the PreFetching-Coordinator (PFC). Both mechanisms can be applied to different combinations of single-level prefetching algorithms, while PFC is also transparent to the upper-level system. These studies are the first to recognize the similarity between SCM and prefetching, leverage SCM techniques to improve prefetching performance.

Chapter 4 presents our work in adding coordination between a supercomputer's storage subsystem, the job scheduler and external data sources for the purpose of providing high data availability at low cost. With the basic observation that supercomputing job data has "life cycles" in accordance to jobs, we connect the job scheduler and the parallel file system to enable data to be scheduled, staged and replicated. Furthermore, the parallel file system is enhanced with metadata about the job data's remote source. Therefore, any lost data can be transparently patched, adding another layer of protection against storage failures. In contrast to disk-level fault tolerance techniques such as RAID, this solution works for whole-system failures. Compared with other system-level techniques such as data replication and erasure coding, this solution is more efficient in the supercomputing environment since it shortens the time that data stays in the system, allocates more resources to "active" data, and leverages the existence of immutable copy of data elsewhere.

In conclusion, the author's doctoral research has produced multiple novel techniques which add coordination to the management of high-end storage systems. The importance and impact of this strategy has been illustrated with experimental results from real computer systems and simulation studies. Many interesting future research topics have been discovered, such as applying SCM techniques to other perspectives of memory resource management, and adding coordination to different types of supercomputing data, and so forth. Several lessons have been learnt, including the importance of keeping coordination algorithms as simple as possible, and the difficulty of designing coordination mechanisms to work well for wide ranges of scenarios.

Bibliography

[1] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002.

[2] T. Andel. Manage inventory, own information. *Transportation and Distribution*, 1996.

[3] Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the i/o bottleneck: network storage with trapeze/myrinet. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 12–12, Berkeley, CA, USA, 1998. USENIX Association.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing, Feb 2009.

[5] Amazon spreading its wings in the cloud. http://aws.amazon.com/.

[6] Amazon spreading its wings in the cloud. http://hardware.silicon.com/storage/0,39024649,39426231,00.htm.

[7] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[8] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. *SIGARCH Comput. Archit. News*, 32(2):176, 2004.

[9] Rakesh Barve, Mahesh Kallahalla, Peter J. Varman, and Jeffrey Scott Vitter. Competitive parallel disk prefetching and buffer management. In *IOPADS '97: Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 47–56, New York, NY, USA, 1997. ACM.

[10] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Explicit control in a batch aware distributed file system. In *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation*, March 2004.

[11] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.

[12] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[13] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of Supercomputing*, 2004.

[14] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, 2005.

[15] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM Conference*, 1998.

[16] P. Cao and S. Irani. Cost-aware web proxy caching. Usenix Symposium on Internet Technologies and Systems (USITS), 1997.

[17] Pei Cao. *Application-controlled file caching and prefetching.* PhD thesis, Princeton, NJ, USA, 1996.

[18] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated

prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1):188–197, 1995.

[19] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[20] John B. Carter, Wilson C. Hsieh, Leigh Stoller, Mark R. Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, pages 70–79, 1999.

[21] Fay W. Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *Operating Systems Design and Implementation*, pages 1–14, 1999.

[22] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 145–156, 2005.

[23] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 269–282, Jun 2003.

[24] S. Chopra and P. Meindl. *Inventory Management and Production Planning and Scheduling*. Prentice-Hall, 2001.

[25] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[26] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. http://www.lustre.org/docs/whitepaper.pdf, 2002.

[27] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the ACM SIGCOMM Conference*, 2002.

[28] R.A. Coyne and R.W. Watson. The parallel i/o architecture of the high-performance storage system (hpss). In *Proceedings of the IEEE MSS Symposium*, 1995.

[29] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM.

[30] Directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman/.

[31] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 56–63, Washington, DC, USA, 1993. IEEE Computer Society.

[32] Fredrik Dahlgren and Per Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(4):385–398, 1996.

[33] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[34] Asit Dan, Daniel M. Dias, and Philip S. Yu. Analytical modelling of a hierarchical buffer for a data sharing environment. *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 21-24, 1991.

[35] Datadirect networks. http://www.datadirectnet.com.

[36] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX '07)*, pages 261 – 274, 2007.

[37] Facebook. http://www.facebook.com/.

[38] Facebook compnay timeline. http://www.facebook.com/press/info.php?timeline.

[39] J. Fritts. Multi-level memory prefetching for media and stream processors. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*, volume 2, pages 101–104, 2002.

[40] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 54–63, New York, NY, USA, 1991. ACM Press.

[41] G. Ganger, B. Worthington, and Y. Patt. The disksim simulation environment version 2.0, Dec. 1999.

[42] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[43] Phillip Gibbons. Data-rich computing: Where its at. In *Hadoop Summit and Data-Intensive Computing Symposium*, 2009.

[44] B. Gill and L. Bathen. Amp: Adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies(FAST '07)*, pages 185–198, 2007.

[45] B. Gill and D. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *Proceedings of the 2005 USENIX Annual Technical Conference(USENIX '05)*, pages 293–308, 2005.

[46] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17, Berkeley, CA, USA, 2008. USENIX Association.

[47] M. Gleicher. HSI: Hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/.

[48] J. Gray. Greetings from a file system user. http://research.microsoft.com/~Gray/talks/Fast_2005.ppt, 2005.

[49] J. Gray and C.V. Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft, December 2005.

[50] J. Gray and A. Szalay. Scientific data federation. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, 2003.

[51] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 13–13, Berkeley, CA, USA, 1994. USENIX Association.

[52] D. Gross and C. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, 1985.

[53] Diwaker Gupta, Sangmin Lee, and Michael Vrable. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, Berkeley, CA, USA, 2008. USENIX Association.

[54] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, third edition, 2003.

[55] Vegard Holmedahl, Ben Smith, and Tao Yang. Cooperative caching of dynamic content on a distributed web server. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 243, Washington, DC, USA, 1998. IEEE Computer Society.

[56] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *SC*, 2005.

[57] Jaguar system. http://www.nccs.gov/jaguar/.

[58] Jaguar's ranking in the top 500 supercomputers. http://www.top500.org/lists/2009/06/.

[59] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional share scheduling of operating system services for real-time applications. In *In Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 480–491, 1998.

[60] Song Jiang and Xiaodong Zhang. Ulc: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 168–177, Mar 2004.

[61] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage*, 4(3):1–25, 2008.

[62] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM.

[63] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *Proceedings of the third international symposium on Memory management*, 2002.

[64] L. Kleinrock. *Queueing Systems: Volume 2: Computer Applications*. John Wiley & Sons, 1976.

[65] B.O. Koopman. On distributions admitting a sufficient statistic. *Transaction of American Mathematics Society*, 1936.

[66] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS2004)*, 2004.

[67] Amit Vasant Kulkarni. Implementing and evaluating scm algorithms for rate-aware prefetching, 2008.

[68] Los Alamos National Laboratory. Operational data to support and enable computer science research. http://institutes.lanl.gov/data/fdata/, 2006.

[69] Oak Ridge National Laboratory. Resources - national center for computational sciences (nccs). http://info.nccs.gov/resources/jaguar, June 2007.

[70] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the International conference on parallel processing*, pages 28–31, 1987.

[71] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

[72] Chuanpeng Li and Kai Shen. Managing prefetch memory for data-intensive online servers. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.

[73] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive prefetching for concurrent sequential i/o. *SIGOPS Oper. Syst. Rev.*, 41(3):189–202, 2007.

[74] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 173–186, 2004.

[75] S. Liang, S. Jiang, and X. Zhang. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS'07)*, page 64, 2007.

[76] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.

[77] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[78] Z. Liu, B. L. Otto-Bliesner, F. He, E. C. Brady, R. Tomas, P. U. Clark, A. E. Carlson, J. Lynch-Stieglitz, W. Curry, E. Brook, D. Erickson, R. Jacob, J. Kutzbach, and J. Cheng. Transient simulation of last deglaciation with a new mechanism for bølling-allerød warming. *Science Magazine*, 325:310–314, July 2009.

[79] Jim Mauro and Richard McDougall. *Solaris Internals.* Sun Microsystems Press , 2001.

[80] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[81] H. Meuer, E. Strohmaier, H. Simon, and J. Dongarra. 29th top500 list of worlds fastest supercomputers released. http://www.top500.org/news/2007/06/23/29th_top500_list_-world_s_fastest_supercomputers_released, June 2007.

[82] The mit beer game. http://beergame.mit.edu.

[83] Cluster resources inc. http://www.clusterresources.com/, 2007.

[84] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[85] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 305–314, Berkeley, CA, USA, January 1991. Usenix Association.

[86] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[87] Nielsen announces september us online video usage data. http://en-us.nielsen.com/main/news/news_releases/2009/october/nielsen_announces.

[88] OpenPBS. Portable batch system. http://www.openpbs.org/.

[89] Oak ridge national laboratory. http://www.ornl.gov/.

[90] Oak ridge supercomputers provide first simulation of abrupt climate change. http://www.ornl.gov/info/press_releases/get_press_release.cfm?ReleaseNumber =mr20090716-00.

[91] John Ousterhout and Fred Douglis. Beating the i/o bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.*, 23(1):11–28, 1989.

[92] S. Devadas P. Jain and L. Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. In *Tech. Rep. CSG-462,M.I.T*, 2001.

[93] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference(USENIX '04)*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[94] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: an idea whose time has come. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[95] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference*, 1988.

[96] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[97] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[98] J. Plank, A. Buchsbaum, R. Collins, and M. Thomason. Small parity-check erasure codes - exploration and observations. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[99] K Ranganathan and I Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the High Performance Distributed Computing Symposium*, 2002.

[100] P. Reungsang, S. K. Park, S.-W. Jeong, H.-L. Roh, and G. Lee. Reducing cache pollution of prefetching in a small data cache. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, page 530, Washington, DC, USA, 2001. IEEE Computer Society.

[101] A Romosan, D Rotem, A Shoshani, and D Wright. Co-scheduling of computation and data on computer clusters. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM 2005)*, 2005.

[102] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceeding of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[103] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, 2002.

[104] B. Schroeder and G. Gibson. Understanding failure in petascale computers. In *Proceedings of the SciDAC Conference*, 2007.

[105] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Essential components for the grid. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, 2003.

[106] Edward Silver, David Pyke, and Rein Peterson. *Inventory Management and Production Planning and Scheduling*. John Wiley & Sons, 1998.

[107] A. Smith. Cache memories. In *ACM Computing Surveys (CSUR)*, volume 14, pages 473–530. ACM Press, 1982.

[108] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 377–390, Berkeley, CA, USA, 2008. USENIX Association.

[109] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[110] Myoung Kwon Tcheun, Hyunsoo Yoon, and Seung Ryoul Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, pages 306–313, Washington, DC, USA, 1997. IEEE Computer Society.

[111] D. Thain, J. Basney, S. Son, and M. Livny. The kangaroo approach to data movement on the grid. In *Proceedings of Tenth IEEE Symposium on High Performance Distributed Computing*, 2001.

[112] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, 1997.

[113] Top500 supercomputer sites. http://www.top500.org/.

[114] Takanori Ueda, Yu Hirate, and Hayato Yamana. The challenge of eliminating storage bottlenecks in distributed systems. In *STFSSD '09: Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*, pages 49–53, Washington, DC, USA, 2009. IEEE Computer Society.

[115] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2), 2000.

[116] S. Vazhkudai and J. Schopf. Predicting sporadic grid data transfers. In *Proceedings of the 11th IEEE Int'l Symposium on High Performance Distributed Computing (HPDC-11)*, 2002.

[117] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *J. ACM*, 43(5):771–793, 1996.

[118] Chao Wang. *Transparent Fault Tolerance for Job Healing in HPC Environments*. PhD thesis, North Carolina State University, 2009.

[119] Chao Wang, Zhe Zhang, Xiaosong Ma, Sudharshan S. Vazhkudai, and Frank Mueller. Improving the availability of supercomputer job input data using temporal replication. In *Computer Science - Research and Development*, pages 149–157, Berlin / Heidelberg, 2009. Springer.

[120] Chao Wang, Zhe Zhang, Sudharshan S. Vazhkudai, Xiaosong Ma, and Frank Mueller. On-the-fly recovery of job input data in supercomputers. In *ICPP '08: Proceedings*

of the 2008 37th International Conference on Parallel Processing, pages 620–627, Washington, DC, USA, 2008. IEEE Computer Society.

[121] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[122] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[123] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 17 – 33, 2008.

[124] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[125] R. Wolski. Dynamically forecasting network performance using the network weather service. *Journal of Cluster Computing*, 1, 1998.

[126] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[127] Jay J. Wylie and Ram Swaminathan. Determining fault tolerance of xor-based erasure codes efficiently. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 206–215, Washington, DC, USA, 2007. IEEE Computer Society.

[128] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Mc2: Multiple clients on a multilevel cache. In *ICDCS '08: Proceedings of the 28th International Conference on Distributed Computing Systems*, Beijing, China, 2008.

[129] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 169 – 184, 2007.

[130] Zhe Zhang, Amit Kulkarni, Xiaosong Ma, and Yuanyuan Zhou. Memory resource allocation for file system prefetching: from a supply chain management perspective. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 75–88, New York, NY, USA, 2009. ACM.

[131] Zhe Zhang, Kyuhyung Lee, Xiaosong Ma, and Yuanyuan Zhou. PFC: Transparent Optimization of Existing Prefetching Strategies for Multi-level Storage Systems. In *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS '08)*, 2008.

[132] Zhe Zhang, Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma, Gregory G. Pike, John W. Cobb, and Frank Mueller. Optimizing Center Performance through Coordinated Data Staging, Scheduling and Recovery. In *Proceedings of the Supercomputing 2007 (SC '07): Int'l Conference on High Performance Computing, Networking, Storage and Analysis*, 2007.

[133] Ming Zhong, Kai Shen, and Joel Seiferas. Replication degree customization for high availability. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 55–68, New York, NY, USA, 2008. ACM.

[134] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*, June 2001.

[135] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 177–190, New York, NY, USA, 2005. ACM.

[136] Qingbo Zhu, Francis M. David, Christo F. Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *HPCA '04: Proceedings of the 10th International Symposium on

High Performance Computer Architecture, page 118, Washington, DC, USA, 2004. IEEE Computer Society.

[137] Qingbo Zhu, Asim Shankar, and Yuanyuan Zhou. Pb-lru: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 79–88, New York, NY, USA, 2004. ACM.