UNIVERSITY OF CALIFORNIA, SAN DIEGO

RobuSTore: A Distributed Storage Architecture with Robust and High Performance

A Dissertation submitted in partial satisfaction of the requirement for the degree

Doctor of Philosophy

in

Computer Science

by

Huaxia Xia

Committee in charge:

Dr. Andrew A. Chien, Chair
Dr. George Papen
Dr. Joseph Pasquale
Dr. Paul H. Siegel
Dr. Geoffrey M. Voelker

2006

The Dissertation of Huaxia Xia is approved, and it is acceptable in quality and form for publication and microfilm:

Chair

University of California, San Diego

2006

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

I would like to thank everyone who helped me during my many years of graduate study in UCSD.

First of all, I would like to thank my advisor Professor Andrew A. Chien for his constant support and motivation. Not only did Andrew give me invaluable advice and instruction on my research, he also helped me to improve writing skills, trained my leadership skills through various lab activities, and encouraged me throughout my academic program. His diligence, preciseness and passion to research have been and will be a great influence on me for many years to come. It has been a great honor to have the opportunity to work with him and learn from him. Furthermore, I would like to thank Professor Joseph Pasquale, Professor Geoffrey Voelker, Professor Paul Siegel, and Professor George Papen for serving on my committee, and helping me with my dissertation.

I would like to thank my fellow graduate students and colleagues in CSAG Lab. I thank those who worked together with me on the OptIPuter Projects, Justin Burke, Frank Uyeda, Xinran Wu, Nut Taesombut, and Eric Weigle for their invaluable discussion and help on my research of RobuSTore. Xin Liu and Alex Olugbile gave me a pleasant experience in the MicroGrid Project. I would also like to thank Troy Chuang, Adam Brust, Ju Wang, Luis Rivera, Richard Huang, Ryo Sugihara, Dionysios Logothetis, Jerry Chou, and Yang-Suk Kee for their valuable discussion on my research and help on my thesis writing and presentation.

Finally, I would like to thank my family for their support, encouragement and love. Special thanks go to my wife Yuanfang Hu. She has always been the first proofreader of my papers and the first audience of my presentations. Her love has made me capable of weathering all the ups and downs throughout the ordeal of my doctoral study.

Part of Section 5.2 appears in UCSD Technical Report CS2004-0798, Evaluation of a High Performance Erasure Code Implementation, by Frank Uyeda, Huaxia Xia and Andrew Chien. The dissertation author was the primary researcher and co-author of this report.

Part of Chapter 6 is published as "RobuSTore: Robust Performance for Distributed Storage Systems" by Huaxia Xia and Andrew Chien in the proceedings of 14th NASA Goddard - 23rd IEEE Conference on Mass Storage Systems and Technologies, 2006. The dissertation author was the primary researcher and author of this paper.

VITA

1998	Bachelor of Engineering, Tsinghua University, Beijing
1998-2000	Research Assistant, Department of Computer Science and Technology, Tsinghua University, Beijing
2000-2006	Research Assistant, Department of Computer Science, University of California, San Diego
2002	Master of Science, University of California, San Diego
2006	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

1. “RobuSTore: Robust Performance for Distributed Storage Systems”, Huaxia Xia and Andrew Chien, 14th NASA Goddard - 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST2006), College Park, Maryland, May 2006.

2. “Validating and Scaling the MicroGrid: A Scientific Instrument for Grid Dynamics”, Xin Liu, Huaxia Xia, and Andrew Chien, Journal of Grid Computing, vol. 2, no. 2, 2004.

3. “The MicroGrid: Using Emulation to Predict Application Performance in Diverse Grid Network Environments”, Huaxia Xia, Holly Dail, Henri Casanova and Andrew Chien, Workshop on Challenges of Large Applications in Distributed Environments (CLADE’04), Honolulu, Hawaii, June 2004.

4. “New Grid Scheduling and Rescheduling Methods in the GrADS Project”, Many authors including Alex Olugbile, Huaxia Xia, Xin Liu and Andrew Chien from UCSD, Workshop for Next Generation Software, Santa Fe, New Mexico, April 2004, also appears in International Journal of Parallel Programming, 33(2-3), 2005.

5. “Experience with the design of a communication protocol for PC clusters”, Jichao Zhang, Weimin Zheng, Di Chang and Huaxia Xia, 4th International Workshop on Advanced Parallel Processing Technologies, Ilmenau, Germany, September 2001.

6. “Optimization of PVM Based on a User-level Communication Protocol”, Jichao Zhang, Di Chang, Weimin Zheng and Huaxia Xia, 4th International Workshop on Advanced Parallel Processing Technologies, Ilmenau, Germany, September 2001.

7. “High-performance PVM Based on Fast Message Passing”, Huaxia Xia and Weimin Zheng, Journal of Software (China), vol. 12, no. 1, 2001.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Grid Computing and Distributed Storage

Professor Andrew A. Chien, University of California, San Diego

ABSTRACT OF THE DISSERTATION

RobuSTore: A Distributed Storage Architecture with Robust and High Performance

by

Huaxia Xia

Doctor of Philosophy

University of California, San Diego, 2006

Professor Andrew A. Chien, Chair

Emerging large-scale scientific applications involve access to distributed large-scale data collections, and require high and robust performance. However, the inherent disk performance variation in distributed shared systems makes it difficult to achieve high and robust performance with traditional parallel storage schemes. We propose RobuSTore, a novel storage architecture, which combines erasure codes and speculative access to tolerate the performance variation. RobuSTore uses erasure codes to add flexible redundancy then spreads the encoded data across a large number of disks. Speculative access to the redundant data enables application requests to be satisfied with only early-completed blocks, reducing performance dependence on the behavior of individual disks.

To demonstrate the feasibility of the RobuSTore architecture, we design a system framework to integrate erasure coding and speculative access mechanisms, and discuss the critical choices for the framework.

We evaluate the RobuSTore architecture using detailed software simulation across a wide range of system configurations. Our simulation results affirm the high and robust performance of RobuSTore compared to traditional parallel storage systems. For example, to read 1 GB data from 64 disks with random data layout, RobuSTore achieves an average bandwidth of over 400 MBps, nearly 15x that achieved by a baseline RAID-0 scheme. At the same time, RobuSTore achieves standard deviation of access latency of only 0.5 seconds, less than 25% of the total access latency, which improves about 5-fold comparing to RAID-0. To write 1 GB data to 64 disks, RobuSTore achieves average bandwidth of 180 MBps, five times faster than RAID-0 even if RobuSTore writes 300% redundant data. RobuSTore secures these benefits at moderate cost of about 2-3x storage capacity overhead and 50% network and disk I/O overhead.

Chapter 1. Introduction

Existing and emerging large-scale scientific applications and data-intensive applications require dramatically higher levels of performance from distributed storage systems. These applications involve access to extremely large data collections and sharing of these data collections for collaboration amongst hundreds of widely distributed users. Distributed storage systems with both high and robust performance are critical to these applications. Throughout, we use the term *robust* to mean low variation in data-access latency.

This dissertation proposes a distributed storage architecture called RobuSTore, which combines erasure coding and speculative access mechanisms to deliver high and robust storage performance. Its erasure coding mechanism encodes the original data into fragment blocks with symmetric redundancy, allowing flexible data reconstruction from subsets of the fragment blocks over the distributed storage system. The speculative access mechanism utilizes fully the available disk bandwidths to read/write redundant fragment blocks from/to heterogeneous distributed disks. Combining these two mechanisms in RobuSTore achieves both efficient disk bandwidth utilization and symmetric data redundancy for high and robust disk performance.

To demonstrate the feasibility of RobuSTore architecture, we design a system framework that integrates the erasure coding and speculative access mechanisms, and then evaluate the system's performance using detailed software simulation across a wide range of system configurations. Our simulation results demonstrate the high and

robust performance of RobuSTore when compared to traditional parallel storage systems.

The remainder of this chapter is organized as follows. Section 1.1 explains the performance requirements from recent data-intensive applications and the necessity and the possibility of using distributed storage systems. Section 1.2 follows with the technology challenges faced by traditional parallel mechanisms in distributed environments. Section 1.3 presents the thesis statement. Section 1.4 outlines the contributions of this dissertation and Section 1.5 summarizes with a roadmap of the entire dissertation.

1.1. Motivation

Existing and emerging large-scale scientific applications and data-intensive applications require dramatically higher levels of performance from distributed storage systems. These applications are increasingly common; they are emerging in virtually every area of science, engineering, and commerce, including biology [1], high-energy physics [2], geology [3], astronomy [4], and many other fields. These applications usually involve massive data collections with objects as large as 10 gigabytes and collections larger than tens of petabytes. For example, the BIRN Project [1] in Medical School of UCSD involves constructing custom indices in a collection of 1,000,000 10GB-size images that total ten petabytes of data. These applications usually have widely distributed data sources and owners, and up to thousands of users, which

implies enormous network bandwidth requirements. The workloads are usually read-dominated, and often require fast, even real-time access.

High performance is essential for these applications to access their large data objects. These objects may be as large as 10s of gigabytes, so transfer rates of hundreds of MBps or even multiple GBps are required to achieve interactive, real-time data accesses. Furthermore, the whole data set, a collection of these large data objects, may come to tens of petabytes, and requires high data-access bandwidth to be shared among multiple distributed users. For example, accessing 1 TB data takes 11 days if the access bandwidth is 1 GBps; it takes three years if the access bandwidth is 10 MBps, which is typical speed for many present-day NFS file servers.

Robust performance is important for both user interaction and resource scheduling. With robust performance, an interactive user can predict the access latency for each access and have comfortable access experiences. Furthermore, resources in a distributed environment are usually shared by many users, so resource scheduling is important for achieving better resource utilization and for guaranteeing the quality of service. Robust and thereby predictable performance is a key for a successful resource scheduling.

The applications also require the storage systems to be distributed. Though widely-used parallel storage systems can provide high levels of performance, they are not good candidates for these data-intensive applications. Parallel storage systems typically have centralized implementations, and consequently are not able to handle large amount of requests from thousands of users efficiently, due to limited network

bandwidth to the central storage site. Also, they fail to provide efficient mechanisms to support widely distributed data sources and owners. Instead, a distributed storage system is a more promising approach. The enormous network bandwidth requirements can be easily satisfied by accessing many widely distributed storage sites in parallel, even if each storage site has only limited network bandwidth. Moreover, distributed storage systems have the advantages of higher data reliability and higher availability. With carefully designed redundancy mechanism, these systems are able to tolerate effectively such single-site failures as power failures, disk failures, or network failures. However, one major challenge in distributed storage systems is how to deliver high-quality data-access performance, which is the focus of this dissertation.

Recent technological advances in networks, microprocessors, and hard drives make intensive data sharing across distributed storage servers possible. First, the emergence of high bandwidth wide-area networks (WAN) provides the basic capability for large data transfers in widely distributed storage systems. Over the past few years, low-cost optical transmissions and the dense wavelength division multiplexing (DWDM) technique have enabled the construction of private WANs with private 10 Gbps (or even 40 Gbps) connections [5]. Indeed, individual fibers have the potential to carry hundreds of 10 Gbps communication channels, called *lambdas*, thereby providing terabits of bandwidth in each fiber bundle. Numerous research infrastructures with private, high-speed lambdas have been deployed (OptIPuter [6], National LambdaRail [7], Netherlight [8], and the Global Lambda Interchange Facility (GLIF) [9]).

Furthermore, rapid increases of CPU processing power make complicated calculation overheads in distributed storage systems affordable. Moore's Law, which postulates that the number of transistors on a CPU chip doubles every 18 months, is still valid in today's industry, as microprocessors pass one billion transistors.

Finally, the exponential growth of hard drive capacity enables us to exploit data redundancy to improve the robustness and disk performance of distributed storage systems. According to Kryder's Law [10], the data density of hard drives doubles roughly every 13 months, while the price of single drives remains almost constant, producing dramatically cheaper data capacity. We can therefore aggregate thousands of disks in a distributed storage system, access them in parallel, and exploit the abundant total disk bandwidth.

Hence, the needs of applications make it desirable, and technology advances make it possible to achieve high and robust performance in distributed storage systems. It should be noted, however, that there are many other important concerns for data-intensive applications such as reliability, availability, security, integrity, and so forth. However, these other topics are not the focus of this dissertation which, instead, focuses on high and robust performance in large-scale distributed storage systems.

1.2. Challenges

One major challenge for distributed storage systems is heterogeneity. Individual disks distributed at different sites may exhibit dramatic performance

differences due to the federated and evolving nature of such infrastructures. First, each site may have different disk types, which may lead to different performance. For instance, an ATA disk may deliver a peak bandwidth of around 10-60 MBps, while a SCSI disk can deliver a peak bandwidth of up to 160 MBps; if some sites use RAID-like parallel disks, hundreds of MBps, or even multi-GBps, of bandwidth can be achieved. With disk manufacturers introducing new generations of products in rapid sequence, many federated storage systems incorporate different disk models.

Next, performance may vary by as much as 100-fold even for the same disk type. This is because a modern hard drive is a hierarchical system consisting of a complex collection of caches and rotating disks, whose access latencies may differ dramatically depending on cache status, disk layout, physical contiguity, and disk head seeking distance. Furthermore, in addition to disk performance variation, network performance variation may further exacerbate the situation. Finally, since distributed storage systems are usually shared by many users, the dynamic competitive workloads lead to dynamic network and disk access behaviors. Interleaved access streams can incur additional seeks that cause as much as a 100-fold variation in performance.

The property of heterogeneity and a requirement for robust performance present new challenges for distributed storage system design. Traditional parallel access mechanisms do not work well in such a heterogeneous environment. With data striping and replication mechanisms, traditional parallel file systems can aggregate disk performance by careful layout scheduling and control. However, these approaches do not perform well in the face of uncontrolled performance variation,

even if replication is used. This is because in such systems, each block in a file is unique and independent of the others, and parallel access has to wait for the arrival of copies of all blocks to get all the data. Simple replication helps a little, but only relaxes the completion requirements slightly. In general, the slowest disks in these systems limit the overall access time.

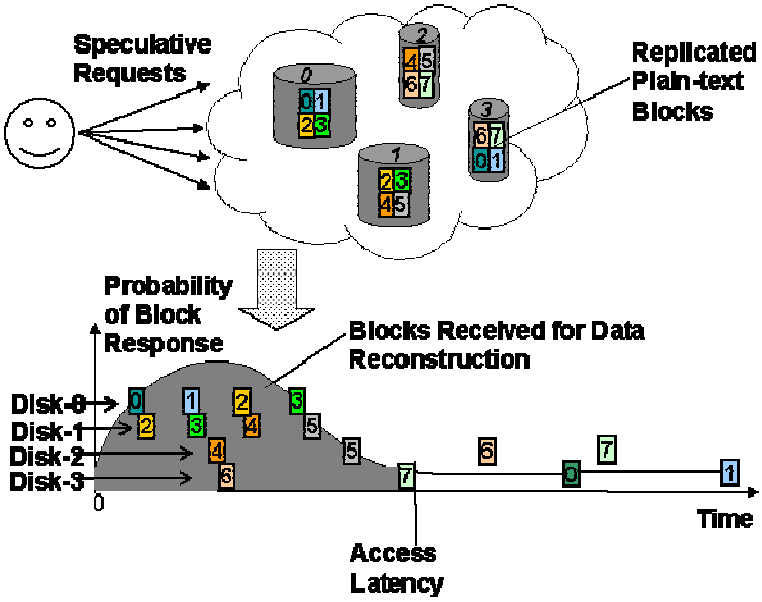


Figure 1-1. Conventional Parallel Storage. The four disks have different performance, with two replicas of eight blocks.

Figure 1-1 shows an example depicting the challenge for traditional parallel file systems. Data on each file consists of eight blocks, labeled from block 0 to block 7. These eight data blocks are replicated to 16 blocks, and are striped and distributed across four disks, as shown in the upper part of the figure. A speculative access is

completed when all blocks from 0 to 7 arrive. Since the disk's performance variation leads to different block arrival times, the first copy of block 7 arrives much later than the copies of all the other blocks. Therefore, even if multiple copies of other blocks have already arrived, the client has to wait for block 7, which leads to much longer data access latency. Even worse, if block 7 from disk 3, for example, gets lost or delayed, the client would have to wait for the arrival of the second copy of block 7 from disk 2, which implies a high variation of access latency.

These new challenges in heterogeneous storage environments demand a set of mechanisms to tolerate high variation in disk access in order to achieve robust access latency and high performance for accessing large storage.

1.3. Thesis and Approach

Our research studies the feasibility of delivering robust and high performance in distributed storage systems by exploiting erasure codes and speculative access. The thesis of our study is as follows: Robust and high storage performance can be achieved in distributed high-speed network environments by using erasure codes to generate symmetric data redundancy and by using speculative access to aggregate efficiently the performance of multiple storage devices.

Subsidiary theses required to substantiate this result include:

(1) Erasure codes can be carefully designed to deliver high encoding and decoding throughput, limited only by processor memory bandwidth.

(2) High access bandwidth can be achieved in distributed storage systems for large reads and writes.

(3) Robust access latency can be achieved in distributed storage systems, with access latency variation less than 20%.

(4) Achieving the above benefits can incur but moderate overhead costs.

We designed RobuSTore storage architecture to realize the idea of combining erasure codes and speculative access in storage. RobuSTore uses erasure codes to add symmetric data redundancy and stripes the encoded data blocks across a large number of distributed disks. With such layouts, clients can speculatively retrieve the data blocks and reconstruct the original data using the fast-returning blocks. As a result, RobuSTore can reduce performance dependence on the slow disks so that it can efficiently aggregate large number of distributed storage devices to deliver robust and high-access performance.

We evaluated RobuSTore architecture under a wide range of system configurations, via both theoretical analysis and detailed simulation. These studies show how RobuSTore architecture is able to increase absolute data access performance and reduce performance variation in distributed storage environments. We also study the overheads introduced by RobuSTore, including storage space overhead, disk access overheads, and network bandwidth overheads. Our simulation results show that RobuSTore architecture is a promising distributed storage system design.

1.4. Contributions

The primary contributions of the dissertation are:

- The RobuSTore idea that combines erasure codes and speculative parallel disk access to improve data access performance and to reduce disk performance variation.
- Design of a system framework that realizes the RobuSTore idea and enables systematic exploration of the design space.
- Evaluation of the robustness of RobuSTore architecture for a wide range of system configurations, and demonstrating reductions in access latency variation as much as five-fold.
- Evaluation of the performance of RobuSTore architecture under a wide range of system configurations, and demonstrating read/write bandwidth improvements as great as 15-fold.
- Studying RobuSTore overheads and showing that only moderate storage capacity overhead (~2-3x) and network bandwidth and disk I/Os overhead (50%) are required to secure all the benefits from the RobuSTore design.

1.5. Dissertation Roadmap

This dissertation presents the RobuSTore idea of combining erasure codes and speculative access to achieve high and robust performance in distributed storage systems. Chapter 2 describes the background of distributed and parallel file systems,

and the fundamentals of erasure codes. Chapter 3 presents the problem definition, the thesis statement, and a brief description of our approach. Chapter 4 gives a high-level overview of the RobuSTore scheme, and presents the idea of combining erasure codes and speculative access.

Next, in Chapter 5 we study several critical system design choices. First, we discuss the usage of erasure coding in distributed storage systems. We compare different erasure coding algorithms and conclude that LT codes are best suited for RobuSTore. We also present an implementation of improved efficient LT codes in Chapter 5. Further, we study how to integrate LDPC codes in RobuSTore, explore the choices of striping the coded blocks to distributed disks, and examine mechanisms to improve read, write, and re-write performance. Finally, we discuss several higher-level system design issues in large-scale distributed storage systems, including security management and I/O request-cancellation mechanism.

Chapter 6 evaluates RobuSTore by comparing it against three conventional parallel storage schemes using detailed software simulation, showing that RobuSTore improves access bandwidth by up to 15 times and access robustness by up to five times, with only a moderate 50% overhead of disk access.

Finally, Chapter 7 summarizes the major conclusions for this work and outlines a number of promising future research directions.

Chapter 2. Background and Related Work

This chapter provides background to help the reader understand the remainder of the dissertation better, and also presents related work. We provide an overview of storage systems in Section 2.1, followed by an introduction to erasure-coding algorithms in Section 2.2. In Section 2.3 we briefly discuss the usage of erasure codes in storage systems.

2.1. Storage Systems

2.1.1 Hard Disk Drive Structure and Behavior

A hard disk drive (HDD, also hard drive or hard disk) is a non-volatile data-storage device that stores data on a magnetic surface layered onto hard disk platters. Hard disk drives have dominated secondary storage since soon after the introduction of the first hard drive by IBM in 1955. The basic architecture of hard drives has been almost unchanged since then. A typical hard disk drive consists of a spindle on which magnetic *platters* spin at a constant RPM, and read-write *heads* move along and between the platters on a common *arm*. The surface of each platter contains data which are organized into a hierarchy of *tracks*, and *sectors*, as depicted in Figure 2-1. The tracks on all the platter surfaces that have the same track number make up a *cylinder*. Cylinders in a modern disk are usually grouped into multiple *zones* based on their distance from the center of the disk. A track in an outer zone has more sectors than a track in an inner zone.

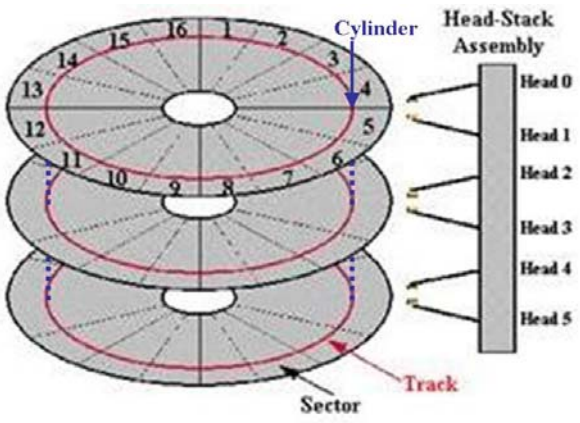


Figure 2-1. Hard Disk Drive Organization.

A disk access is a complex operation. First, the file system's software checks the main memory to see if the required data is there. Modern file systems maintain a free main memory to cache the disk data, as a so-called *filesystem cache*. The filesystem cache is generally all unused main memory, which could be around 1 to 16 GB in a present-day workstation. If the data are not in the filesystem cache, the file system will attempt to acquire the *data bus*, and then send the request to the disk controller.

The *disk controller* checks the *hard disk cache* first. Hard disk caches are usually 2 to 16 MB, so it is usually useful for frequent repeated access or prefetch. If data are missed again in the hard disk cache, the controller will put the request in a request queue. The controller schedules the request processes following some *scheduling algorithm*.

Once the request is finally scheduled and hits the disk, the arm should be moved to the proper track. This operation to move the arm to the desired track is

called a *seek*, and the time is called *seek time*. A modern hard disk usually has an average seek time of about 10ms. After that, the rotation of the platter eventually places the requested sector under the head; the data is then read and transferred back to the file system. The time for the requested sector to rotate under the head is the *rotation delay*. For example, a commodity 7200 rpm hard disk has a rotation delay of 0 to 8 ms.

The next component of disk access, *transfer time*, is the time it takes to transfer the data from the platter to the disk controller buffer. The actual data transfer time depends on data size, rotation speed, and the recording density of a track. The transfer rate of a modern hard disk is typically 30 to 140 MBps.

Therefore, the overall access time can range from almost zero, if the data are in the cache, up to 20 ms per sector. Even if the cache is not considered, different disk layouts may incur different access performance. For example, if all the requested data sectors are contiguous on a disk, then the access bandwidth will be the peak disk transfer rate of 30 to 140 MBps; if the requested data sectors are all scattered around due either to poor disk layouts or to interleaving access from competitive applications, the access bandwidth could be less than 0.1 MBps.

2.1.2 Distributed Storage Systems

A distributed file system is a file system that supports the sharing of files and resources in the form of persistent storage over a network. The first file servers were developed in the 1970s; Sun's Network File System (NFS) [11] became the first

widely used distributed file system after its introduction in 1985. Notable distributed file systems other than NFS include the Andrew file system (AFS) [12] and the Common Internet File System (CIFS) [13].

Existing distributed file systems typically assume slow (10 ~ 100 Mbps) wide-area networks, so caching and prefetching are the primary foci, and single-server performance is sufficient. Generally, little effort is spent on aggregating the performance of multiple servers. LAN storage systems, including NFS[11], Sprite[14], and LOCUS [15], usually depend on file caching and metadata caching to achieve acceptable performance. Sprite uses write buffer to avoid disk operations for temporary files and to write many small files in a buck write. NFS accelerators like Prestoserve [16] and NetApp Filer [17] exploit high-speed NVRAM to improve write performance of NFS. All these systems assume moderate network bandwidth, so single-server performance is generally acceptable. Wide-area file systems such as AFS and Coda [12], xFS [18], and Pangaea [19] spend much more effort enabling partitioned networks or disconnected execution (lazy update), and assume slow networks. None of these systems addresses the aggregation of multiple storage devices for high performance; IO bandwidth achieved by one client is usually bounded by the speed of a single-storage server.

A few distributed storage systems use RAID-like striping to improve the access performance. These systems separate data into multiple plain-text blocks, and stripe these blocks across multiple storage servers. For example, Zebra [20] increases throughput by striping the log-structured new data across multiple servers, delivering

up to 4-5 times the bandwidth of NFS. Another example is pNFS [21, 22], which is an extension to the latest version of NFS, NFSv4 [23]. It is an on-going research project to support data striping and parallel accessing in NFSv4. However, these systems are all based on simple plain-text-based data striping and replication. This simple scheme does not work well in heterogeneous shared environments, which we will demonstrate in Chapter 6.

2.1.3 Parallel Storage Systems

Due to the limited single disk capability, parallel access to multiple disks is required to get high performance. RAID and cluster file systems are two classes of parallel storage systems.

RAID (Redundant Array of Independent Disks) was proposed by Patterson, Gibson and Katz in 1988 [24]. It achieves high performance by aggregating multiple disks of a single server. There are different combinations of disks, or "RAID levels", targeting different performance and fault-tolerance goals. The most commonly used RAID levels are 0, 1, and 5. In RAID-0, data are split into many plain-text blocks and striped across multiple disks. RAID-1 uses an extra set of mirrored disks to improve fault-tolerance. A scheme combining RAID-0 and RAID-1 is called RAID-0+1, which has two mirrored disk sets and striped data in each set. RAID-5 adds parity check blocks during the block-level striping to provide good fault-tolerance with moderate storage space overhead. These RAID levels are depicted in Figure 2-2.

The performance improvement of RAID is limited because RAID can only aggregate a small number of disks in a single server. Especially, each server usually has only limited network bandwidth of 100 MBps ~ 1 GBps, which is not sufficient for large-scale shared accesses.

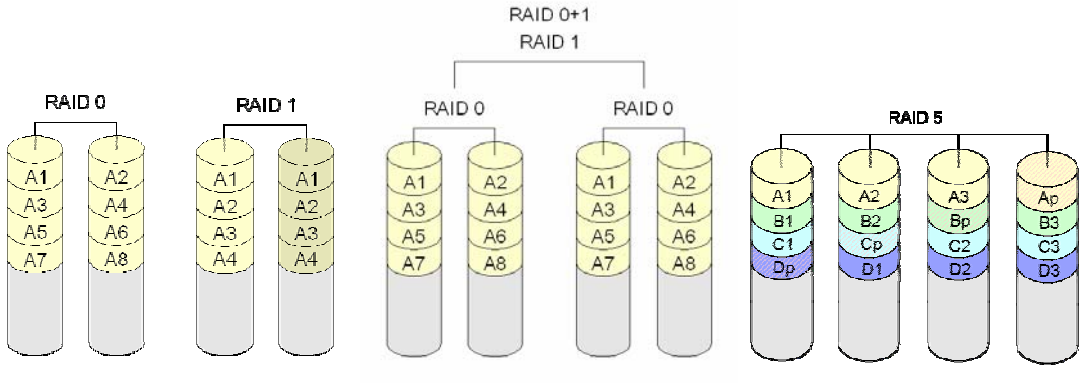


Figure 2-2. RAID Levels.

Cluster file systems address the issues of performance and capacity of single server by aggregating disks from multiple servers in one or multiple local clusters. Popular cluster file systems include Vesta [25], PIOFS [26], Paragon [27], Galley [28], PVFS [29], GPFS [30], Lustre [31], Panasas [32], etc. These file systems use similar data striping mechanism as in RAID-0: they split original data into many plain-text blocks and stripe them to multiple disks or servers. They achieve significant performance improvement by using techniques such as parallel I/O, optimized disk layout, intelligent and coordinated caching, prefetching, write-back techniques, and collective I/O, etc.

However, cluster file systems are not able to satisfy the application requirements for robust and high performance in shared environments. First, the limited network bandwidth of the storage cluster is the bottleneck for multiple remote accesses, since all the disks are at the same site and share the same external network. Second, these systems use a simple parallel scheme that is not fit for a distributed shared environment. While disk performance is both heterogeneous and dynamic in a distributed shared environment, most cluster file systems assume uniform arrays of storage devices in a system-area network (SAN) or local-area network (LAN) environment. Some recent systems such as Lustre [21] and Panasas [32] can utilize heterogeneous disks by allocating more storage objects on faster or larger disks; however, this is done by manual static configuration and cannot handle dynamic disk performance. Experimental results in Chapter 6 will demonstrate these disadvantages of simple parallel file systems.

2.1.4 Peer-to-Peer Storage Systems

Similar to RobuSTore, some peer-to-peer (P2P) file sharing systems (Kazaa [33], BitTorrent [34]) improve access performance by fetching from massively replicated data copies in parallel. The P2P accesses avoid the bandwidth bottleneck of centralized servers. However, the massive replication is unstructured and expensive in terms of storage space overhead. Furthermore, these systems focus on the shared Internet where access networks limit per-node bandwidth to 1-10 MBps.

2.1.5 Summary of Storage Systems

In a distributed environment, each remote server might be a file server with a single disk, a server with a disk array, or even a parallel storage system. To make the discussion easier, we refer to remote filer servers as *disks*. It should be kept in mind that disks' performance may have a high variation range, from less than one MBps up to hundreds of MBps.

2.2. Erasure Coding Algorithms

Erasure codes are a set of coding algorithms providing block-based redundancy. A wide range of error-correcting codes, ranging from Hamming [35] to Reed-Solomon [36, 37] to Viterbi [38], can be used to provide reconstruction flexibility. Recently, attention has focused on advances in erasure codes, a class of codes which tolerate and efficiently recover block-level data loss, a natural match for block and packet-based data-handling systems [39-42]. These codes have found widespread use in a range of innovative storage systems for achieving high levels of data availability and reliability [43-45].

In the following sub-sections, we first describe the basic concepts of erasure codes and introduce the terminologies. Then in 2.2.2 we describe several popular erasure codes, including parity codes, Reed-Solomon Codes, Tornado codes, Luby Transform Codes, and Raptor codes.

2.2.1 Basic Concepts and Terminologies

Erasure codes are a large set of coding algorithms that use a software-based approach to add data redundancy for reliable data transfer. Erasure codes can be used to transmit information reliably from a sender to a receiver; the sender first encodes the original message word into a code word with redundancy symbols and transmits the code word over the transmission channel. A receiver can reconstruct the original source message once it receives a sufficient number of symbols.

In general, an *erasure code* transforms a message of K symbols into a message with N (N>K) symbols in such a way that the original message can be recovered from a subset of those symbols. The ratio of K/N is called the *rate* of the code, denoted R (0<R<1); the ratio of the redundant data is *degree of data redundancy*, denoted as

$$D = N/K - 1$$

$$= 1/R - 1$$

A special case is the digital fountain codes, which can transform K-symbol messages into a practically infinite number of code symbols. These codes can be used with a flexible rate as needed, and hence are called *rateless codes*. The rateless codes are also referred to as *digital fountain* codes.

We need to distinguish between the concepts of word length and data length. In coding theory, a *word* is the whole message to be encoded or decoded, which consists of a number of *symbols*. The number of symbols in a word is called the *word length*. In practice, a symbol could be a bit, a byte, or a large data block, so the actual data length may be significantly different from the word length. We use erasure codes

for large data accesses; in our case, a symbol is a one MB or larger data block. In the later chapters, we also refer to a symbol as a *data block* and refer to a word as a *data segment*; accordingly, we refer to a code symbol as a *coded block*.

According to the reconstruction efficiency, erasure codes can be categorized into optimal erasure codes and near-optimal erasure codes. If any K symbols from the output N symbols are sufficient to recover the original message, the erasure code is an *optimal erasure code*. Unfortunately optimal codes are costly in terms of memory usage and CPU time when N is large, and so near-optimal erasure codes are often used. A *near-optimal erasure code* requires $(1+\varepsilon)K$ blocks to recover the message, where $\varepsilon>0$. *Reception overhead* is ε if $(1+\varepsilon)K$ encoding symbols are required to reconstruct the original K symbols. A code with lower reception overhead requires a smaller number of symbols for decoding; however, it usually has a higher cost in CPU time.

Next, we will briefly describe several optimal codes and near-optimal codes.

2.2.2 Optimal Erasure Codes

An optimal erasure code with rate R transforms a message of K symbols into K/R symbols in such a way that any K suffices to decode the original message; that is, the reception overhead is zero. Parity codes and Reed-Solomon codes are the two most popular optimal erasure codes.

Parity Codes

A parity code includes the original symbols and one parity symbol. The parity symbol is the XOR of all the original symbols, indicating whether the number of ones in the corresponding bits is odd or even. Parity codes are used in many communication protocols and hardware designs as the simplest error-detecting mechanism. They are also used as simple erasure codes in redundant arrays of independent disks (RAID) [24] to recover single-disk failure.

Reed-Solomon Codes

Reed-Solomon codes were invented in 1960 [36] and are widely used in mass storage systems to correct the burst errors associated with media defects, such as in CDs and DVDs. In the codes, the data symbols are represented by the coefficients of a polynomial over a finite field. The polynomial is then evaluated at numerous points over the field, and these values are sent as the block of the encoded message. The number of points evaluated is larger than the degree of the polynomial, so that the polynomial is over-determined; the coefficients can therefore be recovered from subsets of the plotted points.

The major disadvantage of Reed-Solomon Codes is its high computation overhead. Standard algorithms for decoding Reed-Solomon codes require quadratic time to K, which is too slow for even moderate values of K. Therefore, most Reed-Solomon code implementations use $K < 255$ and can only achieve decoding bandwidth up to tens of MBps.

2.2.3 Near-Optimal Erasure Codes

Because of the high computation cost of optimal codes, people often employ near-optimal erasure codes that have positive reception overhead, i.e., $(1+\varepsilon)K$ output blocks are required to reconstruct the message of K blocks for $\varepsilon > 0$. The most popular instances of near-optimal erasure codes are low-density parity-check (LDPC) codes [46, 47]. An LDPC code is a linear code that has a parity check matrix with a small number of nonzero elements in each row and column. It is often expressed as a sparse bipartite graph with each edge corresponding to one nonzero element of the matrix. During encoding, each code symbol on the right side of the graph is calculated as the XOR of its neighboring symbols on left side of the graph, as shown in part (a) of 0. Thus, the encoding time is proportional to the number of edges on the graph. During decoding, if we know the value of a code symbol and all but one of its corresponding original symbols, we can calculate the unknown original symbol as the XOR of the code symbol and the known original symbols, as shown in part (b) of Figure 2-3.

Popular LDPC codes include Tornado Codes [40], LT Codes [48], and Raptor Codes [49]. We briefly describe these codes below.

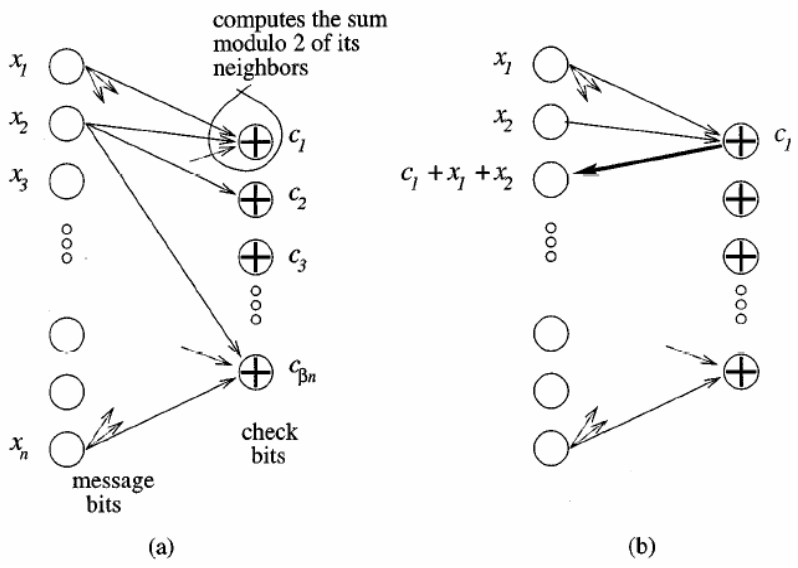


Figure 2-3. LDPC Codes. (a) A graph defines a mapping from message bits to check bits. (b) Bits x_1, x_2, and c_1 are used to solve for x_3.

Tornado Codes

Tornado codes were first described in 1997 [40]. They are erasure block codes based on irregular spare graphs, and were the first erasure codes with linear-time encoding and decoding algorithms.

A Tornado code $C(B_0, B_1, \ldots, B_m, A)$ is a cascade of bipartite graphs $B_0, B_1, \ldots, B_m, A$. The graph Bi has $K\beta^i$ input symbols (from B_{i-1}) and produces $K\beta^{i+1}$ check symbols, where $0<\beta<1$. At the last level, a conventional optimal erasure code is used. The cascade is ended with an erasure-correcting code A of rate $1-\beta$ with $\beta^{m+1}k$ message symbols. The total number of check symbols produced by this sequence is

given by Kβ/(1-β). The final code word is composed of the original message symbols and all the check symbols. The overall code rate is 1-β.

The edge degree distributions of these graphs are carefully chosen so that a simple belief propagation decoder, that sets the value of a graph node only if the values of all its neighbors are known, achieves a high probability of successful decoding.

LT Codes

Luby Transform (LT) codes were proposed by Michael Luby in [48] as the first full realization of the digital fountain approach. LT codes are the first rateless erasure codes from which the data can generate a potentially limitless number of encoding symbols.

LT codes only include one level of irregular bipartite coding graphs, with an unlimited number of parity nodes on the right side. Each encoding symbol is generated independently of all other symbols by the following process:

(1) Generate a random number d from the degree distribution $\mu(d)$.

(2) Choose uniformly at random d distinct input symbols as neighbors of the encoding symbol.

(3) Set the value of the encoding symbol as the XOR of the d neighbors.

Note that the final code word only consists of the parity symbols; the original symbols are not included, although some parity symbols have a degree of one and hence are copies of the corresponding original symbols.

A good degree distribution is essential to LT code design to achieve good decoding performance. In order to balance minimal redundancy with the production of enough edges to keep the decoding successful within an established probability, Luby proposed the robust Soliton distribution $\mu(d)$, which is defined as: first, define $R = c\ln(k/\delta)\sqrt{k}$ for some suitable constant c>0 and δ>0,

$$\rho(i) = \begin{cases} 1/k & \text{for i=1} \\ 1/i(i-1) & \text{for i=2, ..., k} \end{cases}$$

$$\text{and } \tau(i) = \begin{cases} R/ik & \text{for i=1, ..., k/R-1} \\ R\ln(R/\delta)/k & \text{for i=k/R} \\ 0 & \text{for i=k/R+1, ..., k} \end{cases}$$

Then the robust Soliton distribution is:

$$\mu(i) = (\rho(i) + \tau(i))/\beta, \text{ where } \beta = \sum_{i=1}^{k} (\rho(i) + \tau(i)).$$

Raptor Codes

Raptor codes are the latest class of digital fountain codes proposed by Amin Shokrollahi [49] for reliable multicasts. They are an extension of LT codes with linear time encoding and decoding. The key idea of Raptor codes is to relax the condition that all input symbols need to be recovered. If LT codes need to recover only a

constant fraction of their input symbols, then the decoding graph need only have *O(K)* edges, allowing for linear-time encoding. All input symbols still can be recovered by concatenating traditional erasure-correcting codes with LT codes. A graphic presentation of a Raptor code is given in Figure 2-4.

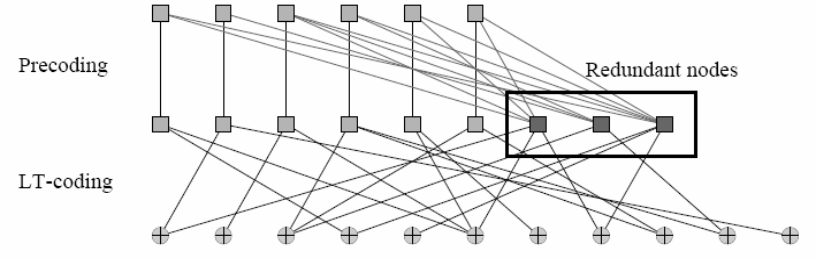


Figure 2-4. Raptor codes. The input symbols are pre-encoded by traditional erasure codes, then encoding with LT codes.

The encoding process consists of two steps. First the input symbols are pre-encoded using a traditional erasure code to get m intermediate symbols; then an appropriate LT code is applied to the intermediate symbols to get N output symbols. The decoding process first decodes the output symbols to get any K intermediate symbols, and then uses the traditional code to get K input symbols.

For a given integer K, and any real $\delta>0$, Raptor codes produce a potentially infinite stream of symbols so that any subset of symbols of size $K(1+\delta)$ is sufficient to recover the original K symbols with high probability. The encoding time is proportional to $O(n\ln(1/\delta))$; the decoding time is proportional to $O(k\ln(1/\delta))$.

2.3. Erasure Codes in Storage Systems

A number of distributed storage systems use erasure codes to improve data reliability and availability. Weatherspoon et al. [50] analyzed the impact of erasure coding and replication on availability, concluding that erasure-coded systems have mean time to fail many orders of magnitude higher than replicated systems with similar storage and bandwidth requirement. Numerous storage systems, such as Oceanstore [43], Frangipani [51], Total Recall [45], PASIS [52], Koh-i-Noor [53], Typhoon [54], and [55] have exploited erasure codes for data reliability and availability. While some of these systems increase performance opportunistically by exploiting data redundancy, for none of them is this a focus.

Collins and Plank's [56] work is most closely related to RobuSTore. Their work uses Reed-Solomon codes or LDPC codes to improve access bandwidth in wide-area storage systems. They concluded that Reed-Solomon codes improve the storage access bandwidth more significantly than LDPC codes. However, they assumed slow shared networks with bandwidth less than 10MBps, and a small number of blocks ($N \leq 100$). In contrast to Collins and Plank's work, RobuSTore is designed for high bandwidth dedicated networks and explores a much wider array of design choices in data coding, redundancy, layout, and access. In such environments, Reed-Solomon codes cannot provide required high bandwidth due to their high computation cost. Further, Collin and Plank only studied configurations with different N and different block selection methods, while a wide range of other configuration parameters also

have important impacts on access performance, including block size, network latency, degree of data redundancy, and so on. In this dissertation, we study the choice of erasure codes in more general situations and explore a range of configuration parameters using detailed simulation.

Chapter 3. Thesis Statement

This chapter outlines the research context, defines the research problem and presents the thesis statement.

3.1. Context

Emerging large-scale data-intensive applications require distributed storage systems with robust and high performance. These applications involve massive data collections and real-time accesses to data objects of size larger than gigabytes. Therefore, high storage performance is essential for these applications to access the large data objects; robust performance is important for real-time accesses and resource scheduling. Our research focuses on new data access mechanisms to deliver robust and high performance from a large collection of distributed disks. Although there are a range of other critical requirements for storage systems, such as reliability, security, integrity, manageability, and adaptability, they are not the foci of this dissertation.

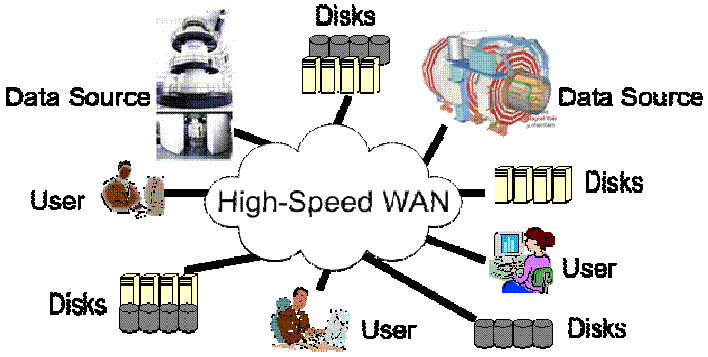


Figure 3-1. Distributed Applications and Shared Storage Systems.

Because a single disk has limited bandwidth, parallel access aggregating multiple disks is essential for high performance. However, in heterogeneous dynamic environments, doing this efficiently is an open research question. An efficient mechanism should be able to adaptively access the remote disks and tolerate the variation of individual disk performance. Such tolerance and adaptability are lacking in existing local parallel file systems, so applying the same parallel mechanisms in distributed storage systems will not achieve robust and high performance.

Our approach is to integrate erasure codes and speculative access into distributed file systems to tolerate the performance variation. We can encode the original data using erasure codes to introduce symmetric redundancy in data blocks and then distribute and retrieve the coded blocks among a large number of distributed disks. To realize our idea efficiently, we need to explore the wide-range configuration space in data coding, data striping, and data retrieving, and to develop a design guideline for system implementation and configuration.

3.2. Problem Definition

The problem we are solving is how to achieve robust and high storage performance in distributed shared systems for large data accesses. Our workloads are derived from data-intensive scientific applications, and are often dominated by write-once and read-only accesses of hundreds of MB to tens of GB each. Update operations are rare in these applications.

The main challenge is to effectively tolerate performance variation of the disks. In distributed shared storage systems, the disks may be shared among hundreds

of widely distributed applications and users. Each application may involve many collaborative users located nation-wide or even world-wide who share same set of data collections. Therefore, workloads on each disk or network channel are dynamic, incurring dynamic access performance to the disks.

We assume that a distributed storage system includes abundant resources of disk space, network bandwidth, and CPU processing capability, so that we may trade these resources for robust and high performance. We believe in the near future it will be easy to aggregate thousands of disks in a large-scale scientific project by combining a few clusters of hundreds to thousands of nodes. A large number of disks provide abundant total storage capacity and total disk bandwidth. We further presume that both client-end networks and wide-area networks have high bandwidths. Single network interface card (NIC) can already provide up to 10 Gbps for clients; and higher network bandwidth can be achieved by clustering multiple NICs. In wide-area networks, DWDM technology enables each individual fiber to carry hundreds of 10 Gbps communication channels, providing terabits of bandwidth per fiber and dramatically more bandwidth per fiber bundle. Finally, the industry continues to increase per-chip CPU performance following Moore's Law. Therefore, it is reasonable to trade some of these resources for robust and high performance in data accesses.

While abundant resources are available in a distributed storage system, they are not free and should be used efficiently. Since the resources are shared among hundreds of widely distributed applications and users, each storage system may be used by multiple applications simultaneously. Abusive resource allocation will hurt

the access performance of other applications or users. Therefore, although we have abundant hardware resources, they are not infinite or free.

3.3. Thesis Statement

Our research studies the feasibility of delivering robust and high performance in distributed storage systems by exploiting erasure codes and speculative access. My thesis is as follows:

By using erasure codes and speculative access, RobuSTore can efficiently aggregate large number of distributed storage devices to deliver robust and high storage performance in distributed high-speed network environments. Specifically, well-designed erasure codes can provide symmetric data redundancy with high encoding/decoding throughput. By exploiting this redundancy, speculative access can fully utilize the bandwidth of multiple heterogeneous disks, with tolerance to their performance variation. The combination of erasure codes and speculative access allows writing/reading data to and from a large number of distributed disks efficiently.

Subsidiary theses required to substantiate this statement include:

(1) Erasure codes can be designed to deliver high encoding and decoding throughput.

(2) High-access bandwidth can be achieved in widely distributed storage systems.

(3) High-access robustness can be achieved in distributed storage systems.

(4) Achieving the above benefits entails moderate overhead costs.

To prove the thesis, we need first show the feasibility of the RobuSTore idea. We discussed the theoretical advantages of using erasure codes and speculative access, and then designed RobuSTore's storage architecture to realize the idea of combining them together in storage systems. We further discussed the choices for RobuSTore components, giving a guideline for real RobuSTore implementation and configuration.

Furthermore, we proved the soundness of using erasure codes for high-speed storage. We compared different erasure codes using theoretical analysis and real experiments. Based on these studies, we decided that LT codes are best fitted for our RobuSTore storage architecture. Further, we explored the coding parameters and decided the best configuration using both theoretical analysis and software simulation. We improved LT codes and delivered a highly efficient implementation which achieves 600 MBps decoding bandwidth on a 2.8 GHz ADM Opteron processor.

Finally, we designed a range of simulation experiments to show the advantages and the costs of RobuSTore. Our detailed software simulation shows that the RobuSTore architecture can efficiently aggregate the performance of hundreds of distributed disks on large reads and writes. Comparing to simple striping and parallel access such as RAID-0, RobuSTore achieves up to 15 times bandwidth improvement and 5 times robustness improvement. RobuSTore uses about two to three times the storage space to achieve such improvements. On data accesses, it uses two to three times the I/O accesses for writes and about 1.5 times for reads.

Chapter 4. RobuSTore Approach

In this chapter, we present the RobuSTore idea of combining erasure codes and speculative access. We then explain why this approach is feasible and how it can improve access performance. We first present the key RobuSTore idea in Section 4.1. In Section 4.2, we present the design of the RobuSTore architecture followed by an explanation of the detailed access procedures in Section 4.3.

4.1. Key RobuSTore Idea

The key idea of RobuSTore is to combine erasure codes and speculative access to aggregate the dynamic heterogeneous disk performance in distributed environments efficiently, producing robust and high performance for large data accesses. RobuSTore uses erasure codes to add symmetric data redundancy, and stripes the encoded data blocks across a large number of distributed disks. With such layouts, clients can speculatively retrieve the data blocks and reconstruct the original data using the fast returned blocks. As a result, RobuSTore reduces performance dependence on the disks that are slow to respond. As a result, RobuSTore can efficiently aggregate large number of distributed storage devices to deliver robust and high access performance.

At a high level, our approach is trading storage bandwidth, network bandwidth, and computing for low and robust access latency. In the scenarios of data-intensive scientific applications and, storage bandwidth and network bandwidth are relatively plentiful and low and robust access latency is more difficult and more important to achieve, as described in Chapter 1.

4.1.1 Using Erasure Codes for Symmetric Data Redundancy

Erasure codes introduce symmetric data redundancy. An erasure code transforms data of *K* blocks into *N* (N>K) coded blocks. The coded blocks contain symmetric data information in such a way that the original data can be recovered from a flexible subset of those blocks. Optimal erasure codes such as Reed-Solomon Codes are perfectly symmetric and can use any K-coded blocks to reconstruct the original data. Near-optimal erasure codes allow the reconstruction using any (1+ε)K-coded blocks, which are still much more flexible than plain-text replication.

The flexibility provided by symmetric redundancy allows adaptive read access to heterogeneous disks. We spread the coded blocks to a large number of distributed disks. Since a read client can use any coded blocks to reconstruct the original data, it can read more blocks from a faster disk and less blocks from a slower disk. This is analogous to the usage of erasure codes for reliable communication; we use erasure codes to tolerate data loss in communication, while in storage systems we can think of the data blocks as lost blocks if they are on slow disks and cannot be retrieved in a short time. This redundancy reduces the dependence of the large-read request on the performance of individual disks.

Furthermore, rateless erasure codes such as LT codes provide a flexible degree of redundancy, which allows adaptive write accesses to heterogeneous disks. Rateless codes can generate a practically infinite number of coded blocks; statistically, any subset of these blocks of a certain size will provide the same level of data redundancy.

Therefore, if a write client wants to write N coded blocks to the disks, it may encode the original data into N' ($N'>N$) coded blocks, and adaptively spread them to many disks until N of the N' blocks are successfully written to disks.

To give a quantitative sense of how much flexibility erasure codes provide, we theoretically analyzed the number of blocks required for data reconstruction in both erasure-coded schemes and plain-text replicated schemes. Assume we have a K-block file and use four times its storage space. If using a plain-text replicated scheme, each block has four copies and at least one copy for each block should be retrieved to reconstruct all the data completely. The probability of successful reconstruction with M random replicated blocks is:

$$P(M) = \sum_{i=1}^{N} (-1)^{N-i} \frac{\binom{K}{i}\binom{4i}{M}}{\binom{4K}{M}}$$

In contrast, if we encode the K blocks into 4K blocks using a typical LT code in which the average encoded-node degree is about five, we can reconstruct the original data from M random coded blocks with the following probability:

$$P_c(M) = \sum_{i=1}^{K} (-1)^{K-i} \binom{K}{i} (\frac{i}{K})^{5M}$$

See the Appendix for the full analysis. In practice, about 3K blocks are needed in a replicated scheme versus about 1.5K blocks in an erasure-coded scheme (see Figure 4-1).

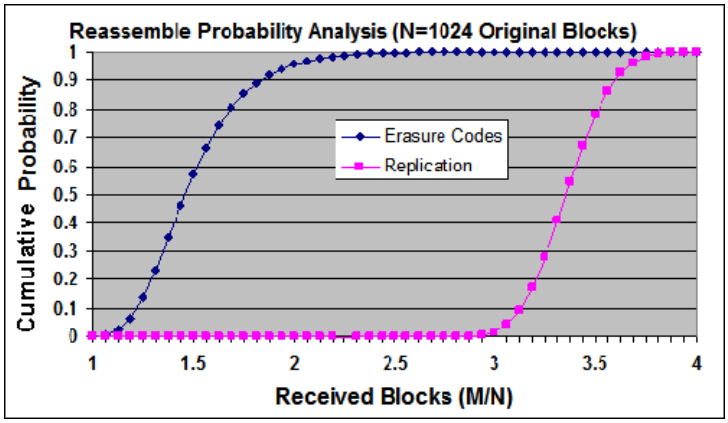


Figure 4-1. Cumulative Probability of Reassembly of Original Blocks. Assume there are N=1024 original blocks and 4096 coded or replicated blocks.

4.1.2 Using Speculative Access to Tolerate Performance Variation

Although erasure codes provide the access flexibility, we need a mechanism to utilize this flexibility to tolerate performance variation of the distributed disks. RobuSTore does this using speculative access. The basic idea of speculative access is to request more data blocks than needed from a large number of disks, to wait for the requests to be processed in parallel by the disks, and then to cancel the requests once enough blocks have been confirmed as completed. The speculative access for writes and reads are slightly different. To write a K-block data and use a factor of λ ($\lambda>1$) spaces, writing clients would first encode the data into N' blocks where $N'>\lambda K$, exploiting the rateless feature of the erasure codes. They would then send requests to many disks and transfer coded blocks to them in parallel, then cancel the ongoing writing once $N=\lambda K$ blocks have been confirmed as written success. To read the data, reading clients would request all the blocks from the disks, then continue receiving

them in parallel until enough blocks have been received. Benefiting from the decoding flexibility, clients can then reconstruct the original data using the sets of early-returning blocks.

In distributed environments with high-speed networks, speculative access improves performance significantly with only minor overhead. The major advantage of speculative access is that it only sends requests once to each disk during each access, which minimizes the impact of long latency in distributed environments. Without predicting or monitoring the performance of each individual disk, clients can take advantage of symmetric redundancy from erasure codes and simply "read all then cancel" or "write all then cancel". However, due to the long latency between the clients and the disks, some data bytes may already be on-the-fly at the time when the clients send out cancel signals, in either the disk-accessing or network–transferring phases. The actual disk and network accesses will be more than what is needed. Fortunately, we have abundant network and disk bandwidth, and the overhead only corresponds to one round-trip time.

4.1.3 An Example

By combining erasure codes and speculative access, RobuSTore can tolerate late-arriving blocks and reduce the dependence of a request on any individual disk, and hence achieve robust and high performance. Figure 4-2 provides an example depicting this advantage in read access. In the example, an eight-block of data is encoded into 16 blocks which are spread across four disks. We assume the data

reconstruction needs eight coded blocks, although the number could be slightly larger if we use near-optimal erasure codes. Read clients first send requests to all four disks for all the blocks. The disks then transfer the data blocks back to the clients at different speeds. Once the clients receive eight blocks, they cancel the rest of the accesses, reconstruct the original data, and complete the access with a high-average bandwidth. Furthermore, if any of these first eight blocks are lost or delayed due to any reason, the clients only need to receive one more block and complete the overall access with only slightly longer latency.

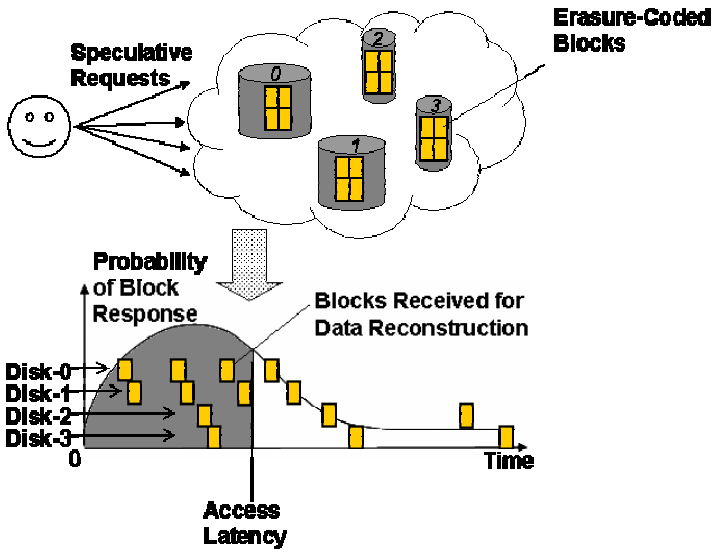


Figure 4-2. Advantage of Using Erasure Codes and Speculative Access. The four disks have different performance; Assume eight original blocks are encoded into sixteen coded blocks.

4.2. RobuSTore Framework

To realize the RobuSTore idea, we designed a RobuSTore framework which includes three key components: client, metadata server, and storage servers, as shown in Figure 4-3.

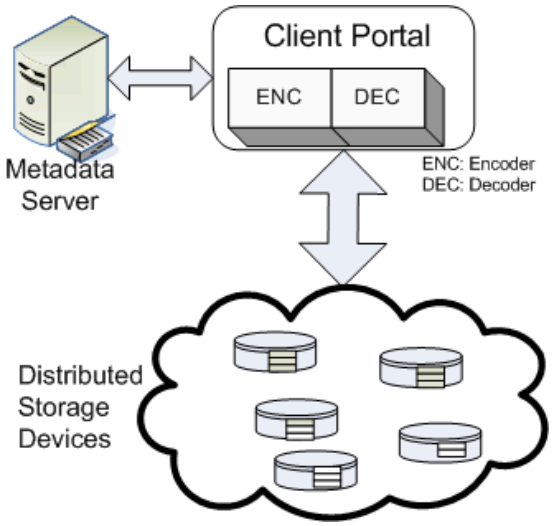


Figure 4-3. RobuSTore System Framework.

Clients perform the many distributed file system functions of accessing metadata, planning layout, encoding data, sending requests to storage servers, and decoding and assembling completed responses. First, when a client receives an application request, it queries from metadata server for information of data layout and related disks (disk address, performance, and load, etc). Next, based on this information and the quality of service specified by the application, the client will select a set of disks to do the access. For this purpose, the client also need to include

an access scheduler which selects amongst the candidate disks and blocks from which they access, and a Layout Planner which selects the devices' degrees of redundancy. If the request is a write, the client then encodes the data and sends them to the disks; if it is a read request, the client reads coded data blocks from the disks and decodes the data. The encoding and decoding modules (ENC/DEC) are implemented on the client side, which has several advantages: this provides the maximum leverage for client accesses to benefit from the order-flexibility in the returning responses (sort of an end-to-end argument), decouples the encoding/decoding cleanly from the design of storage and metadata servers, and enables easier and broader experimentation and configuration.

Metadata Servers maintain both data information and storage server information. Data information includes data name, data size, data location, encoding algorithms, owner, access rights, file locks, and so forth. Storage server information includes storage capacity, expected performance, recent load, connectivity, availability, and so forth. Metadata servers get the data information from client: each time when a client writes or updates some data, it reports the latest data information to metadata servers. The storage server information may come from multiple sources. Some static information, such as disk capacity and peak performance, is known when the storage servers register to the metadata server and join the storage systems. Dynamic storage information may come from the client accesses and periodic queries to the storage servers.

The metadata server implementation could be centralized or distributed, depending on the system's scale and the performance requirements. A central metadata server minimizes update and synchronization costs at some penalty in scalability. However, because metadata access is primarily needed on open/close, a well-designed metadata server can support a large-scale system. For example, in the Lustre parallel file system, a single metadata server can support a cluster with thousands of disks. On the other side, a distributed metadata server has the potential to support more disks and users with faster responses, while it also involves higher management costs for synchronization, load balancing, and so on.

Storage Servers provide data storage at block level (erasure-coded blocks, presumed to be larger than disk blocks). Servers may be single disks or disk arrays, and each implements local admission and access control. Servers typically have variable performance due to heterogeneity in hardware, data layout, or load.

Each server may have its own admission control mechanism. An admission controller controls access to the storage server in order to avoid disk overload and to guarantee the quality of services. Admission controllers are optional. They are useful when the storage system has high workloads and are required to guarantee the quality of services.

4.3. RobuSTore Access Procedures

4.3.1 Basic Operation Interfaces

Basic RobuSTore operation interfaces include:

- `open(filename, read/write,QoS_options);`

This function accesses the metadata server, opens the file, gets disk map information, and plans an access schedule based on the application QoS requirements. The function returns a file descriptor which includes information like data location, coding algorithm, coding parameters, and data offset, etc. An extended discussion is in Appendix B.

- `write(fdescriptor,data,length);`

This function encodes the data and spreads the coded blocks across the selected disks until enough blocks have been completely written.

- `read(fdescriptor,buffer,length);`

This function requests coded blocks from the selected disks, receives blocks, decodes the data until all the original data has been decoded successfully, and cancels the read requests after completion.

- `close(fdescriptor);`

This function registers the data structure and location with the metadata server if the data is written or modified and releases the file lock.

To show the relationship between the basic operations and the RobuSTore's components, we will explain the detailed processes of data write, data read, and data modification in the subsections below.

4.3.2 Write Access

The left side of Figure 4-4 depicts the write processes. In step 1, clients first access the metadata server, open the file, and plan layouts based on disk map information and application QoS requirements. The clients then allocate the needed storage on the servers, encode the data appropriately using specified coding parameters to generate redundant coded blocks, and transfer the blocks to the selected servers in parallel to the encoding, shown as step 2 and step 3 in the figure. Once enough data blocks are committed to the servers, the clients register the data structure and location with the metadata server, release the file lock, and complete the write access.

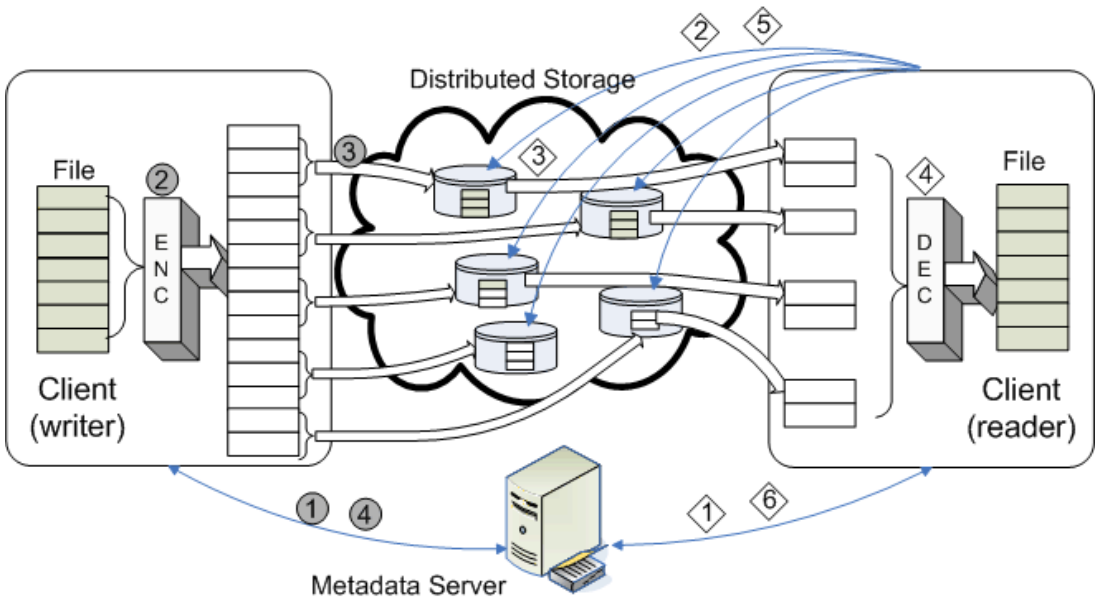


Figure 4-4. Write and Read Processes in RobuSTore. (Write in dark circles, Read in white diamonds)

4.3.3 Read Access

The right side of Figure 4-4 depicts the read process. The numbers with white squares in the figure show the steps in sequence. Similar to the write access process, read accesses start from queries to the metadata server, from which clients obtain storage server information, data structure and location information, and any required locks. To read, the clients request all coded blocks from servers and decode the received blocks in parallel. When enough blocks have been received, the decoding finishes and the original data are reconstructed. At the same time, outstanding requests to the storage servers are cancelled. Finally, the close function notifies the metadata server, releasing read locks and bandwidth reservations on the storage servers.

4.3.4 Update Access

Although update operations are rare in most data-intensive scientific applications and are not our focus in this dissertation, a neat mechanism is still needed to deal with these operations. In RobuSTore, if optimal erasure codes are used, then any minor modification may cause the change of almost all the coded blocks; however, if near-optimal erasure codes are used, the change to one original block only affects a limited number of coded blocks. For example, in our implementation of LT codes with 1024 original blocks and 4096 coded blocks, the average degree of original block is about 20. In order to change one original block, we need to update at most 20 coded blocks, which is about 0.5% of the total encoded data.

The complete update process is as follows. The clients first get data location information from the metadata server. They can then examine the coding graphs and figure out which coded blocks should be updated. Next, they regenerate those coded blocks, and spread them out to remote disks (not necessary for the disks that store the old coded blocks). Finally, the clients notify the metadata server about the updated blocks and notify the disks to delete the obsolete coded blocks.

4.4. Summary

In summary, we proposed the RobuSTore idea of combining erasure codes and speculative access in distributed storage environments, explained its potential for achieving robust and high performance by giving a high-level analysis, and showed its feasibility by presenting the design framework and access procedures.

Chapter 5. Design of RobuSTore: Critical Choices

There are many choices for the design, implementation and configuration of a RobuSTore system, including choices for erasure codes, speculative access and admission control. These choices are critical because they have major effects on the efficiency and performance of the RobuSTore systems. We discuss the choices in this chapter. Section 5.1 explains why these choices are critical. Section 5.2 discusses the erasure codes choices. Section 5.3 discusses the issues related to speculative access, including data layout preparation and request cancellation. Finally, Section 5.4 explains the admission control.

5.1. Introduction

The RobuSTore system framework enables flexible choices of the design space for the RobuSTore components. Different choices have significant impact on the system performance, so it is important to make proper choices.

The critical design choices for RobuSTore involve choices around erasure codes, speculative access, and admission control. Choices for erasure codes are critical because every RobuSTore access involves erasure encoding or decoding. The properties of erasure codes have a direct impact on access performance and the resource overhead (CPU, network, and disks). Choices for speculative access are also critical for RobuSTore performance. Since speculative access interact with erasure codes, their choices affect access latency and system loads. Furthermore, since

RobuSTore systems are shared by many users and applications, it is important to have proper admission control mechanism to provide efficient sharing and guarantee the quality of service. We discuss these major design issues below.

5.2. Choices of Erasure Codes

Since RobuSTore accesses are tightly related to erasure codes, it is essential to understand the erasure codes' behavior, parameters, and performance, and choose erasure codes with the best algorithm and the best configuration for RobuSTore. In this section, we discuss the choices of erasure codes in several aspects. First, we explore the different erasure codes, analyze which codes are the best for RobuSTore, and explain why we choose LT Codes. We then examine the different LT Codes parameters and describe our improvement and implementation of LT Codes.

5.2.1 Coding Algorithm Selection: LT Codes

To deliver high and robust performance on thousands of hard drives, we need erasure codes with low reception overhead, low computation overhead, and long code words. Low reception overhead means high coding efficiency, i.e., only a small number of coded blocks are enough to reconstruct the original data. Low computation overhead allows high bandwidth data encoding and decoding so that we can use an ordinary computer with moderate-speed CPU as a RobuSTore client. A code with a long code word can generate a large number of coded blocks, which brings two

benefits to RobuSTore. First, it allows RobuSTore to stripe the encoded data blocks across many disks and to retrieve them from many disks in parallel. Furthermore, long code words allow splitting the original data into more finely grained blocks, which brings more data access flexibility. These three features cannot be optimized at the same time. It is therefore important to choose proper coding algorithms and proper coding parameters.

As described in Chapter 2, there are many different erasure codes, including optimal erasure codes, poorly-optimized erasure codes, and near-optimal erasure codes. We analyze the properties of using each of them in RobuSTore as following.

Optimal erasure codes achieve perfect coding efficiency, but they have a high CPU overhead when using long code words. A rate R optimal code transforms the original K-block data into N=K/R blocks in such a way that any K-coded blocks suffice to decode the original data. This optimal coding efficiency implies that the information about every original block is mingled into at least N-K coded blocks, since otherwise we can find K coded blocks that are not sufficient to reconstruct the original data. Hence, on encoding, every coded block should be generated by computing at least K(N – K)/N original blocks on average, and on decoding, the reconstruction of every original block will need to compute at least K(N – K)/N coded blocks on average. Considering N=K/R, the encoding time is at least:

$$
\begin{aligned}
& N \bullet K(N-K)/N \\
&= K(N-K) \\
&= \frac{1-R}{R} K^2
\end{aligned}
$$

and the decoding time is at least:

$$K \bullet K(N-K)/N$$
$$= R \bullet K(N-K)$$
$$= (1-R)K^2$$

Both encoding time and decoding time is quadratic in K (and thus also quadratic in N). Hence, the encoding and decoding bandwidth is inversely proportional to K. For example, we implement an instance of Reed-Solomon codes and test its performance of encoding and decoding 16 MB data, which is shown in Table 5-1.

Table 5-1. Coding Bandwidth of Reed-Solomon Codes. Tested on 2.4GHz Intel Xeon.

K (# original blocks)	N(# coded blocks	Encode Bandwidth (MBps)	Decode Bandwidth (MBps)
32	64	13.7	15.9
16	32	26.8	31.3
8	16	53.3	60.8
4	8	112.2	99.5

At the other extreme, pure replication is the simplest form of erasure codes with poor efficiency. Replication has a low run-time overhead (a copy) and permits efficient random access to sub-blocks of an object. However, replication is highly

inefficient in environments with dynamic disk performance, since many storage replicas are required to tolerate disk performance variation. For example, if there are K original blocks that are replicated 1/R times (so the total block number is N=K/R), and we randomly read the N blocks, the expected number of blocks we need on average to have at least one copy for each of the K original blocks is:

$$f(1) = 1$$

$$f(2) = f(1) + K/(K-1)$$

$$f(3) = f(2) + K/(K-2)$$

…

$$f(K) = f(K-1) + K = K(1/K + 1/(K-1) + \ldots + 1/2 + 1) \approx K\ln K$$

With limited client-side network bandwidth and limited shared disk bandwidth, the cost of $K\ln K$ is significantly high.

Near-optimal erasure codes make a good trade-off between reception overhead and computation overhead; they require only a few more than optimal coded blocks for reconstruction, but can usually support long code words with low CPU overhead. In near-optimal erasure codes, each original block is only associated with a small number of coded blocks, and vice versa. For example, LDPC codes use sparse bipartite coding graphs in which each coded block is the parity of a few original blocks. The computation cost for encoding or decoding each block is proportional to the node degrees of the corresponding coded blocks, which are usually proportional to $\ln K$.

Among all the different LDPC codes, we selected the Luby Transform (LT) codes [48] for RobuSTore. While there are many different LDPC codes that approach the Shannon limit of channel capacity and have reasonable encoding/decoding complexity, such as Tornado codes, LT codes and Raptor codes, for our purposes LT codes [42, 48, 49, 57] have a number of advantages. First, they are *rateless*, which allows redundancy to be decoupled from other system-design issues, such as the number of storage servers used, and also allows adaptive writing. Second, LT codes use only one level of bipartite structure and block-XOR operations, so that they can be implemented with high coding throughputs. Third, their structure allows the coding process to be overlapped with data I/O, effectively eliminating the critical path time of coding.

5.2.2 LT Codes Parameters

Given the LT coding algorithm, its performance is affected by many coding parameters, including word length (K), code word length (N), coding graph density, and data redundancy (D=N/K-1). We analyze the parameters in theory below.

First, large K and N are required for flexibility in speculative access. If there are a large number of blocks, we can spread them to many disks and read them from many disks in parallel, and hence aggregate the bandwidth of many disks to achieve high performance. Furthermore, with the large number of coded blocks, we can put multiple blocks on each disk, which enables the reading of a different number of blocks from each disk using speculative access. If the average remote disk bandwidth

is 20 MBps, which is typical in contemporary storage systems, we need to access about 64 disks to saturate a client network with a 10 Gbps (1.2 GBps) network. Since we use block-based access in RobuSTore, if the maximum performance variation of the disks is q, we should have at least q blocks in each disk to utilize the performance of both fast and slow disks. Therefore, in a storage system with tens or hundreds of times the performance difference between the disks, we need hundreds or thousands of coded blocks.

On the other hand, it takes higher coding overhead to use larger K and N. To understand this, we analyzed the encoding and decoding cost of LT codes. First, each coded block in LT codes is the parity of multiple original blocks. Assuming the average degree of coded block is d_e, we need d_e–1 XOR operations to recover each block. Second, a successful decoding requires at least that all the original blocks be covered by the received coded blocks. Since the coding graph is randomly generated, each coded block can only cover d_e random original blocks. The minimum number of coded blocks required for reconstruction is:

$$(1+K/(K-1)+K/(K-2)+\ldots+K) / d_e = K\ln K / d_e$$

To achieve good reception overhead, d_e should be close to $\ln K$. This means that a large K should correspond with a denser coding graph with a large encoded node degree d_e.

The data redundancy (D=N/R–1) is another important parameter for erasure codes. It decides the storage space overhead and the read flexibility. However, LT codes are rateless and the coding algorithm is independent to redundancy, so we delay

the discussion on data redundancy until Section 5.3.2 and give only the conclusion below: the data redundancy is at most the maximum performance variation of the disks; a reasonable choice is the ratio between the peak disk performance and the average disk performance.

Considering these factors, we chose K=128~1024 and N=512~4096. In the next subsection, we further explore the code word length by implementing the LT codes and running them across different Ks and Ns.

5.2.3 LT Codes Improvement

LT codes are non-optimal codes with many good features. We have described the algorithms of LT codes in Chapter 2. They are simple, fast, and rateless. However, the original LT codes are optimized for communication; they do not fit perfectly in storage systems.

The original LT algorithms have several drawbacks when used in storage. First, the coded blocks are completely independent, so the codes cannot guarantee decodability with any finite number of coded blocks; instead, they only provide a significantly high probability of success for decoding with a certain number of coded blocks. In communication, this is acceptable since the sender can keep sending new coded blocks until the receiver can reconstruct the original data. In storage, however, the writing and reading are asynchronous operations, so it is the writer's responsibility to guarantee the decodability. Second, the coding graph has irregular coverage. The edges are randomly distributed among the original blocks, which makes some original

blocks have a low node degree and become a bottleneck for both performance and reliability. Third, the original decoding algorithm does XOR operations greedily whenever a new coded block is received. Many XOR operations are actually not needed.

To use LT Codes in storage systems, we need the codes to have guaranteed decodability, robust reception overhead, and low CPU and memory cost. Therefore, we made the following improvement and optimization of the original algorithms:

(1) We guaranteed the decodability by checking the coding graph. In our improved encoding algorithm, we first generated the bipartite graph without doing block XOR operations and checked the decodability of the coded blocks. If the original file could not be reconstructed, a new bipartite graph would be regenerated until we have guaranteed decodability.

(2) We covered the original blocks uniformly. On encoding, instead of randomly selecting original blocks as neighbors of coded blocks, we chose original blocks in order to make all original blocks have same node degree, or, at most, different in one. We implemented this using the following *pseudo-random selection* technique. We first generated a random permutation of the original blocks, then used them one-by-one when generating the coded blocks; a new permutation was generated whenever all the original blocks in the last permutation were used.

(3) We reduced CPU and memory cost by doing lazy XOR instead of greedy XOR. On decoding, we did memory XOR operations only when we could decode a block. This eliminated any operations to generate intermediate data that would not

help to decode the data blocks. In addition, this method also leveraged memory and cache locality at the system level to reduce memory hits.

(4) We reduced memory cost by using instruction-level optimization. We optimized the memory XOR operations for efficient usage of registers and cache on the processor. More specifically, we used variables with great care so that the number of registers needed in each loop body would not exceed the total number of registers available in processor; we used a long operand to reduce the loop times; further, we used striping for XOR on large memory buffers so as to maximize cache usage.

5.2.4 LT Codes Performance

We implemented LT codes with the improved features as above, and tested the performance with different coding parameters.

Figure 5-1 shows the reception overhead and its standard deviation for different numbers of data blocks (K) with different parameters C and δ. Reception overhead decides the disk and communication overhead; a small reception overhead implies less data to be retrieved from remote storage devices. Although different Ks correspond to different sweet spots of C and δ, it is not hard to find good parameters to achieve reception overhead in 0.3-0.5. For example, when K=1024, C=1 and δ=0.1, the reception overhead is about 0.5 and the corresponding standard deviation is about 0.08 which is about 5% of the total received blocks.

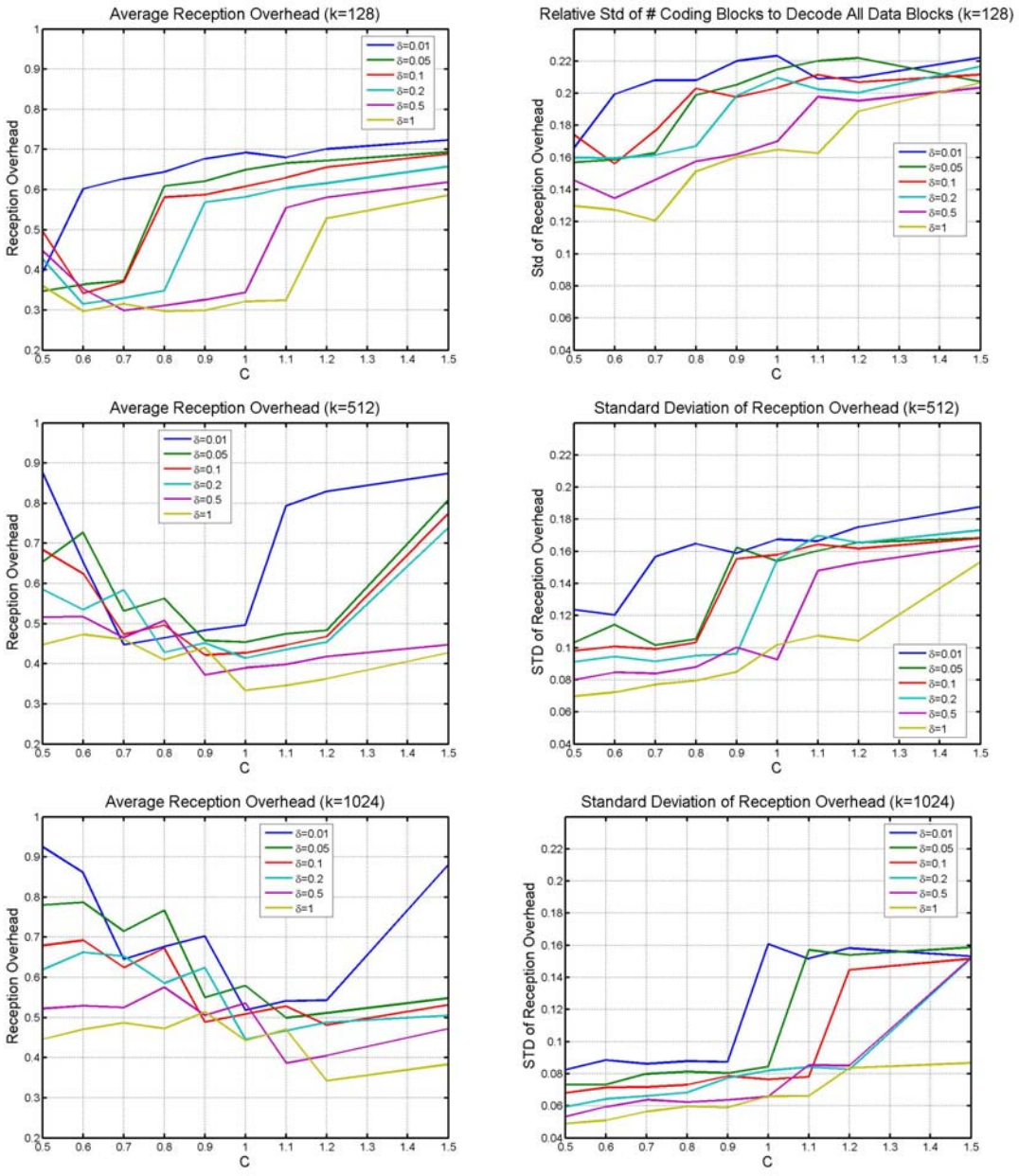


Figure 5-1. Reception Overhead of LT Codes. Left plots show average reception overhead, i.e., the number of coded nodes necessary to decode all data nodes; right plots show the relative standard deviation. The plots from top to bottom corresponds to K=128, 512, 1024.

As for CPU overhead, we first examined the amount of memory in XOR operations. In LT codes, each memory XOR operation involves one edge in the bipartite graph. Figure 5-2 shows the average number of edges to be used on decoding and its standard deviation. These actually correspond to K multiplied by the average node degree of coded blocks. Given C and δ, the node degree follows a fixed distribution, so the average node degree is independent of K; we only show the plots for K=1024 here.

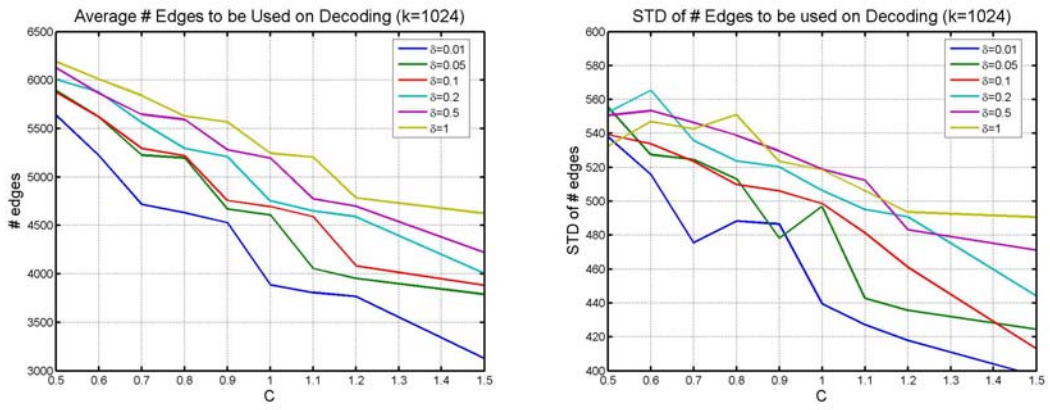


Figure 5-2. Number of Edges Used on LT Codes Decoding. Left plot shows average number of edges used; right plot shows its relative standard deviation. K=1024.

Comparing the plots in Figure 5-1 and Figure 5-2, we notice that C and δ have opposite effects on CPU overhead and communication overhead. Basically, small δ and large C cause less CPU overhead, but more communication overhead. For a specific system, we can choose the proper C and δ based on CPU speed and the

network speed of the client machines. In the following sections, we assume δ=0.1 and C=1 so that the reception overhead is 0.5.

The following figure shows the actual decoding bandwidth on one 2.8GHz AMD Opteron Processor. The results show a decoding speed fast enough to saturate network speed on most client systems. For example, for C=1.0 and δ=0.1, the decoding bandwidth is 394 MBps, while the reception overhead is about 50%, which means that it can keep up with a network interface with 394*(1+0.5)*8 Mbps=4.7 Gbps data receiving bandwidth. If the client is equipped with faster network interface card, we can choose larger C and smaller δ. For example, for C=2.0 and δ=0.01, the decoding bandwidth is 550 MBps and the reception overhead is about 136%, so that the decoder can saturate a 10 Gbps network interface card.

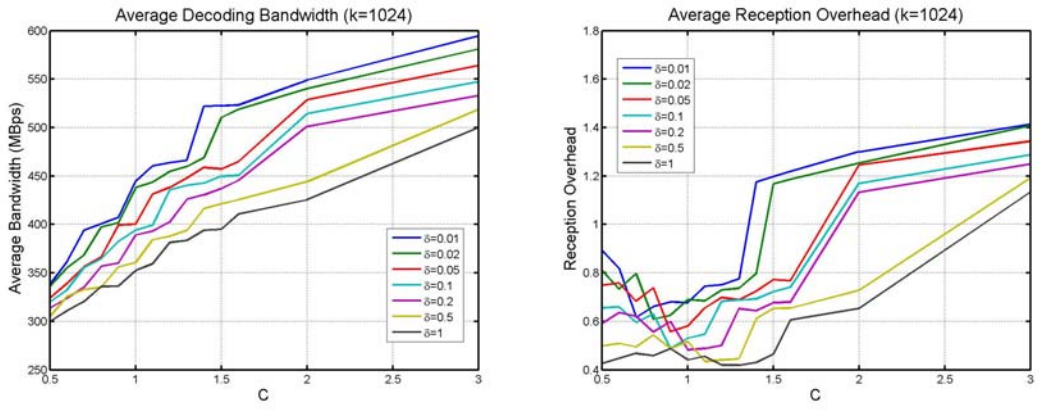


Figure 5-3. Decoding Bandwidth and Reception Overhead of LT Codes. Tested with a LT codes implementation on one 2.8 GHz AMD Opteron processor.

5.3. Choices of Speculative Access

During the speculative access, RobuSTore trades the abundant disk and network resources for better performance. However, it is important not to use the resources abusively. We discuss three issues related to proper resource utilization: disk selection, data redundancy, and request cancellation.

5.3.1 Disk Selection

On writing each new data, clients need to choose a certain number of disks to use. The number of disks to access decides the level of parallelism. When we stripe the data to multiple disks, we can write and read the disks in parallel to aggregate the performance of multiple disks. Therefore, the number of disks should be no less than the expected total access bandwidth divided by the average disk bandwidth. For example, if the average remote disk bandwidth is 20 MBps, which is typical in contemporary storage systems, we need to access about 64 disks to saturate a client network with 10 Gbps (1.2 GBps).

Besides the number of disks, clients should also choose which disks to use. Different disks have different workload patterns, different disk bandwidths, different capacities, different network bandwidths, different geographical locations, and different failure models. Proper disk selection is important for improving both access performance and data reliability.

Access performance can be improved by selecting distributed lightly-loaded disks. First, the disk performance is sensitive to the number of concurrent accesses due to its rotation and seeking feature. By selecting the lightly-loaded disks to access, the

client can reduce the access contention with other clients, and therefore get better performance and help the system to deliver a high overall system throughput. Another consideration is storage space. The clients should try to write data to those disks with larger free spaces; otherwise, when many disks are full the later-written clients would have fewer disks to choose and eventually get poorer performance due to intensive access contention. Finally, disk location is another factor that affects access performance. In a storage system with many distributed users, it is important to have each file striped across multiple distributed sites because this allows the accesses to flow through many different network paths, reducing network congestion.

Data reliability can benefit from selecting distributed disks and balancing different disk failure models. Using distributed disks provides better support for disaster recovery. A disaster usually destroys servers only in a small region. If data are spread across multiple sites with erasure-coded redundancy, they can be easily reconstructed from data blocks on the available disks.

Another consideration is a disk failure model. Different disks have different availability. If we choose all low-availability disks, we might be in danger of being unable to find enough disks to read later on. On the other hand, if we always choose high-availability disks, we may use up these disks and leave only low-availability disks for later accesses. Hence a mixed selection with both high-availability disks and low-availability disks is recommended. The disk failure models also have different patterns. For example, servers located in different time zones may have different maintenance times. It is better to select disks with mixed failure patterns.

5.3.2 Data Redundancy

The required storage space is decided by original data size and the degree of data redundancy. In RobuSTore, data redundancy is the ratio between the number of redundant coded blocks and the number of original data blocks. We denote data redundancy as D,

$$D=(N - K)/K = 1/R - 1.$$

Data redundancy affects both the writing performance and the reading flexibility, so it is important to choose a proper data redundancy. First, if more coded blocks are generated, this takes more network and disk bandwidth and also a longer time to write them into the storage system. Hence, to improve writing performance, we should use as few coded blocks as possible. On the other hand, more coded blocks provide higher flexibility in choosing which blocks to read, which allows RobuSTore to adapt to higher performance heterogeneity of the disks. Therefore, high data redundancy is good for read performance.

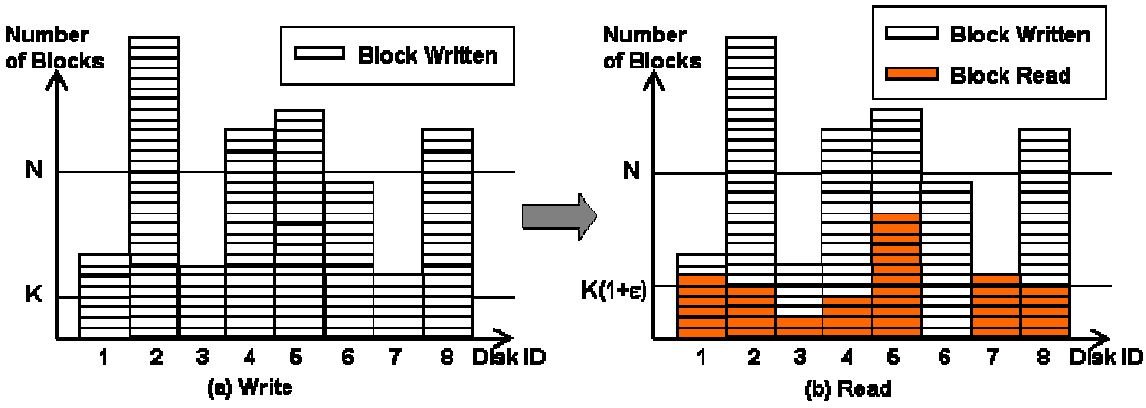


Figure 5-4. Tolerate Dynamic Disk Performances.

A good choice of data redundancy should represent a proper tradeoff between storage usage and flexibility for access adaptation. The number of coded blocks should be barely enough to allow each disk to have enough blocks to send during a read access. For example, as depicted in Figure5-4(a), when we write data into many disks, a different number of blocks are written to different disks due to disk performance heterogeneity. Assume there are H disks (here H=8), and disk *i* has bandwidth B_{iw}. The average writing bandwidth, then, is:

$$\overline{B_w} = \frac{\sum_{i=1}^{H} B_{iw}}{H}.$$

If the data redundancy is D, the total number of coded blocks is N=(D+1)K. The amount of data written to disk *i* is then:

$$F_{iw} = B_{iw} \cdot T = B_{iw} \cdot \frac{N}{H \cdot \overline{B_w}} = \frac{N \cdot B_{iw}}{H \cdot \overline{B_w}}.$$

When clients speculatively read the data blocks, the dynamic disk performance might be quite different from what it had been when writing the data, as depicted in Figure5-4(b). Assume disk *i* has bandwidth B_{ir}, then the average reading bandwidth is:

$$\overline{B_r} = \frac{\sum_{i=1}^{H} B_{ir}}{H}.$$

Non-optimal erasure codes have positive reception overhead ε, meaning that they require (1+ε)K coded blocks to construct the K original blocks. If there are enough blocks on every disk to read, the amount of data read from disk *i* are:

$$F_{ir} = B_{ir} \cdot T = B_{ir} \cdot \frac{K(1+\varepsilon)}{H \cdot \overline{B_r}} = \frac{K(1+\varepsilon) \cdot B_{ir}}{H \cdot \overline{B_r}}$$

To guarantee every disk has enough blocks, the following should be satisfied:

$$F_{ir} \le F_{iw}, \forall i \in [1, H]$$
$$\Leftrightarrow \frac{K(1+\varepsilon) \cdot B_{ir}}{H \cdot \overline{B_r}} \le \frac{N \cdot B_{iw}}{H \cdot \overline{B_w}}, \forall i \in [1, H]$$
$$\Leftrightarrow D = \frac{N}{K} - 1 \ge (1+\varepsilon) \cdot \frac{\overline{B_w} \cdot B_{ir}}{\overline{B_r} \cdot B_{iw}} - 1, \forall i \in [1, H]$$

When the number of disks is large, statistical theory tells that $\overline{B_r} \cong \overline{B_w}$. Therefore, the required data redundancy is:

$$D = (1+\varepsilon) \cdot \max_{1 \le i \le H} \frac{B_{ir}}{B_{iw}} - 1 .$$

B_{ir}/B_{iw} is the performance variation of each individual disk.

The performance variation of hard drives may be up to factors in the tens of or even hundreds with the existence of different access contentions. This is a strict boundary of D; however, a very large D is not required in practice. First, it is rare for the disks to be that heavily loaded since we always select the most lightly loaded disk upon which to write new data, and can usually achieve reasonably high B_{iw}. Second, if only a few disks have insufficient number of blocks to read, they will not have a significant impact on overall performance.

In a specific case, if we write same amount of data to each disk, i.e., $F_w = N/H$, then we can get the following using a similar analysis process:

$$D = (1+\varepsilon)\cdot \max_{1\le i\le H}(B_{ir} / \overline{B_{ir}}) - 1 .$$

In Chapter 6, our experiments show that data redundancy of two to three is enough to provide the best performance.

5.3.3 Request Cancellation

Request cancellation is important in RobuSTore for efficient resource sharing. RobuSTore uses speculative access, which request more coded blocks than what are needed to reconstruct data. However, as the network and disk resources are shared among a large number of users and applications, resource abuse hurts the accesses of other users and applications and eventually hurts our own accesses. Therefore, a mechanism is required to cancel the uncompleted request once enough blocks have been written or read.

Request cancellations can be implemented at the hard disk drive controller. Each modern hard drive has its own processor and buffer. The processor, usually called a controller, allows the CPU to talk to the hard disk. It coordinates the communication between bus and hard disk, maintains the disk request queue, and manages the disk cache. When a request comes from the bus, the controller first puts it into the request queue and waits until the disk is available for the next data request. All the unprocessed requests are in the disk request queue. We can therefore implement the request cancellation by removing the corresponding requests from the queue. This

is feasible for SCSI disks since they provide syntax-rich interfaces. For example, the Advanced SCSI Programming Interface (ASPI) [58] provides a function *SendASPI32Command* that can send a *SC_ABORT_SRB* command to cancel a previously submitted request. In ATA disks, however, no similar interface is yet available.

Another method is to implement request cancellations in the file system software. The file system can buffer all the requests and send them to the disk drive one by one so that all the unprocessed requests are in the file system software and can be cancelled when needed. This method does not require any support from the hard drive interface, and can thus be implemented in any file system.

5.4. Choices of Admission Control

In RobuSTore, the access to storage devices is granted by the admission controller. The purpose to use admission controller is in two folds: First, it can guarantee the quality of service to existing accesses. Further, it protects the storage systems from abusive sharing. Sharing same disk by multiple concurrent large accesses usually damages the disk throughput dramatically due to the rotating character of hard disks, so a proper admission controller is necessary to avoid exorbitant sharing and improve the total throughput of the server. Because the ultimate goal of RobuSTure is to provide robust performance for multi-user workloads in federated shared storage environments, the role of admission controllers is critical.

Admission controllers are associated with storage servers, reflecting the need to control resource access in a structure that is both scalable and is compatible with federated access in a distributed setting. Clients must negotiate on behalf of applications for access to storage servers, and the admission controllers act as resource managers for the storage servers.

Admission controllers make decision based on estimated storage throughput, ongoing accesses, and size/latency of the new request. When workload of a storage device exceeds its capability, new access requests to the device will be refused.

There are two classes of admission controlling mechanisms: capacity-based and priority-based. Capacity-based admission control (CAC) is to service requests based on arrival time. With CAC, new flows are indiscriminately admitted until capacity is exhausted (First Come First Admitted). New flows are not admitted until capacity is available. Priority-based admission control allows some requests to preempt others based on priority settings. However, adding traffic to a higher priority queue can affect the performance of lower-priority classes, so priority queued systems must use sophisticated admission control algorithms.

In network communication area, there are several mature works on admission control on network bandwidth. For example, RFC 2751 [59] and RFC 2815 [60] describe admission control in a priority-queued system; RFC 2816 [61] discusses methods for supporting RSVP in LAN environments.

However, admission control in storage systems is different from that in networks. In shared systems, disk bandwidth is less predictable than network

bandwidth. Multiple simultaneous accesses to one disk usually get much less total performance than exclusive access to the disk due to the costly disk seeking and rotation in switching between the accesses. Furthermore, the mapping of the high level QoS goals to low level storage device actions is usually complex. Admission control in storage systems is still a hot research topic.

In RobuSTore, we reserve the choice on admission control for future work. This is based on the following considerations. First, the experiments of admission control need good workload models for multiple competitive accesses. However, we do not have good enough workload model or traces, hence experiments on admission control is not very meaningful. Furthermore, different storage sites are likely to have different access control policies, which make it complex to study.

5.5. Summary

In this chapter, we have discussed several critical design issues in RobuSTore, including choices for erasure codes, speculative access, and access control. We show that LT codes are good candidates for coding; improved LT codes with parameters C=1 and δ=0.1 can achieve decoding bandwidths up to 500 MBps on a single Opteron 2.8 GHz CPU, with about 0.5 reception overhead. In a typical implementation, a client should also choose proper speculative access schemes. The number of disks to access in parallel is based on the expected average remote disk bandwidth and the client's process capability. The data redundancy should reflex the difference between the average disk bandwidth and the fastest disk bandwidth. Further, we show that request

cancellation can be implemented in either hard drive or filesystem software. Finally, we also discuss the choices for admission control. These discussions show the feasibility of a practical RobuSTore implementation in distributed environments.

Another important issue for distributed storage system is access control. However, this is not the focus of our dissertation, so we only briefly discuss the issue in Appendix C.

5.6. Acknowledgement

Part of Section 5.2 appears in UCSD Technical Report CS2004-0798, Evaluation of a High Performance Erasure Code Implementation, by Frank Uyeda, Huaxia Xia and Andrew Chien. The dissertation author was the primary researcher and co-author of this report.

Chapter 6. Performance Evaluation

In this chapter, we study the advantages of RobuSTore over traditional parallel storage schemes in terms of tolerating performance variation from data layout, competitive workloads, and filesystem caching. We evaluate these storage systems using detailed software simulation, and simulate the systems across a wide range of configurations, including different numbers of storage devices, network properties and degrees of data redundancy.

The chapter is organized as follows. Section 6.1 introduces the simulation-based performance evaluation method and its advantages. Section 6.2 gives the experimental design, including storage schemes, simulator design, workloads, metrics and experiment configurations. The experiment results are presented and analyzed in Section 6.3.

6.1. Introduction

We use detailed discrete-event simulation to evaluate the RobuSTore scheme quantitatively. Our theoretical analysis in previous chapters provides a qualitative view of the advantages of using erasure codes and speculative access in storage systems. A quantitative evaluation, however, requires detailed simulation, which captures behaviors of the components in a complex storage system and enables more accurate studies of the RobuSTore architecture.

With detailed simulation, we are able to explore a wide range of configuration spaces of a storage system. Storage systems are complex and include many

components; there are many choices of mechanisms, algorithms and other parameters for each component. The overall system performance may vary significantly under different configurations. With software simulation, we can evaluate the simulated system with various configurations and explore the configuration space systematically. The scenarios we are going to study include environments with performance variations from data layout, competitive workloads, and filesystem caching, and different configurations of disks, networks, and coding algorithms. Such study reveals the RobuSTore performance in different situations, thereby helping us to decide upon the optimal configurations for the RobuSTore scheme.

As a basis for comparison, we model and simulate three traditional parallel storage schemes in addition to RobuSTore: RAID-0, RRAID-S, and RRAID-A, defined in Section 6.2.1. We evaluate each scheme's performance for read, write, and read-after-write across a wide range of configurations. The results show that RobuSTore delivers a significant improvement on access bandwidth and robustness in most cases.

6.2. Experimental Design

In this section, we describe the storage schemes for comparison, simulation design, metrics, workloads, and experiment configurations in detail.

6.2.1 Storage Schemes for Comparison

We evaluate the RobuSTore scheme by comparing it against conventional parallel storage schemes. The conventional schemes are RAID-0, RRAID-S, and

RRAID-A, which are different from RobuSTore in terms of the data layout mechanism or access mechanism.

RAID-0 uses data striping with zero redundancy and accesses all the striped data in parallel. In RAID-0, each data segment is split into many plain-text blocks (Figure 6-1a), which are then interleaved across multiple disks in the system (Figure 6-1c). On data reading, since the data is not replicated, the client has to read all the blocks to get the complete data segment. To do this, the client sends one request to each disk for all the corresponding blocks on that disk and reads the blocks in parallel.

RRAID-S uses replicated data striping and speculative access. As in RAID-0, the original data segment is first split into many plain-text blocks. However, in this case, multiple copies of the blocks are distributed over the disks, each replica starting one disk rotated over. Specifically, i-th block of replica r is stored on disk $(i+r \bmod H)$. Figure 6-1d shows an example for an 8-block segment with two replicas. The data layout in RRAID-S is just slightly different from that in RAID-0+1, where data blocks are striped across a "disk set" and *mirrored* to multiple other disk sets. The data layout in RRAID-S is more flexible and allows arbitrary redundancy, so that we can study the performance with a wide range of data redundancy. On a read access, RRAID-S speculatively requests for all data blocks on each disk in parallel, depicted in Figure 6-2a. The read access is complete when at least one copy of each block in the data segment arrives.

In RRAID-A, the same data layout mechanism as RRAID-S is used, but the read access mechanism is different. Instead of sending a single request to each disk for

all the segment blocks, the client adaptively requests for required blocks according to the runtime disk performance, as shown in Figure 6-2b. Initially, the reader sends requests to all disks for the blocks in the first replica; in the example of Figure 6-2b, it requests block 0 and 4 from disk 0, block 1 and 5 from disk 1, etc. Once the reader has received all requested blocks from a disk, say disk *A*, the reader identifies the disk with the largest unreceived data blocks that *A* has, say disk *B*, then it divides *B*'s unreceived data into two halves and requests the second half from disk *A*. This scheme avoids reading redundant data, but it also risks large overheads in long network latency environments to send multiple-round adaptive requests.

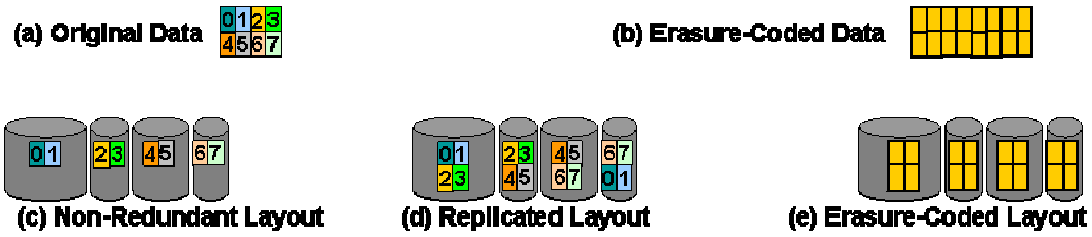


Figure 6-1. Data Layouts. 8 original blocks; 1x data redundancy in replicated and coded layouts.

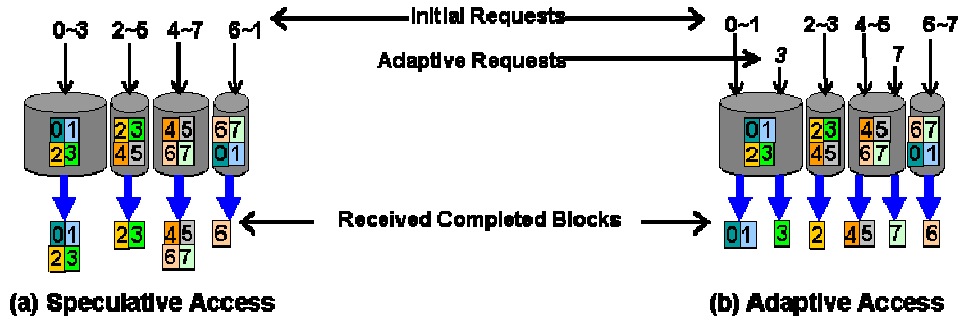


Figure 6-2. Access Mechanisms. Disk performance is varied.

RobuSTore, as we have discussed in previous chapters, uses erasure-coded redundancy (Figure 6-1e) and speculative access (Figure 6-2a).

In summary, we studied four schemes, including RAID-0 with zero data redundancy and speculative access, RRAID-S with replicated redundancy and speculative access, RRAID-A with replicated redundancy and adaptive access, and RobuSTore with erasure-coded redundancy and speculative access.

6.2.2 Simulator Design

Our simulator consists of two major parts: virtual client and virtual server. The virtual server further consists of a virtual filer and one or more virtual disks, as shown in Figure 6-3. When a client needs some data for its application, it figures out the data location and sends requests to the corresponding file servers. The requests are first processed by the virtual filer. It checks if the data is in cache or not. If not, it passes the requests to corresponding virtual disks and gets responses after proper access delay. Finally, the client receives the data from the filers and completes the whole access by decoding the data. Each virtual client, virtual filer, or virtual disk is implemented as one individual process in our simulator.

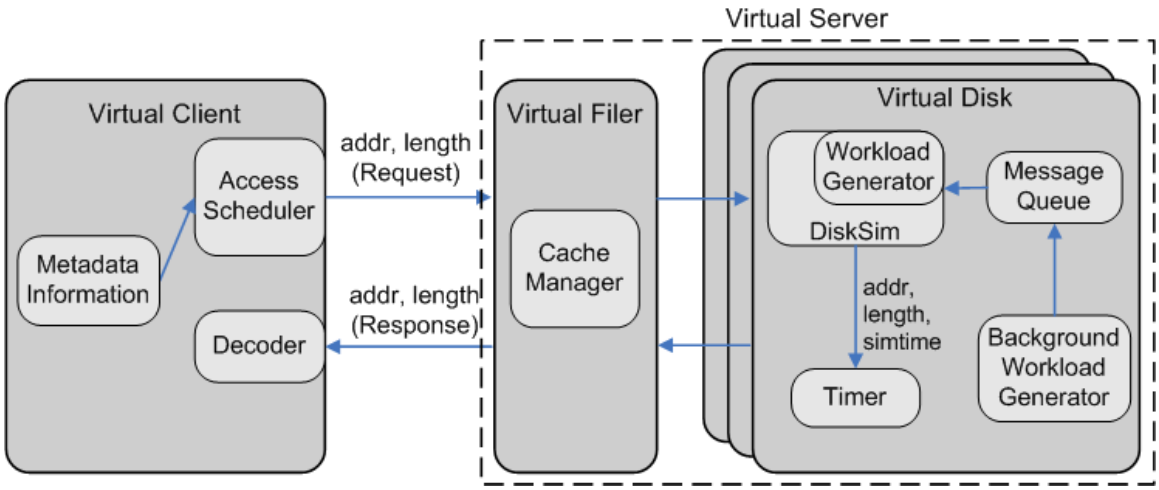


Figure 6-3. Simulator Architecture.

Virtual Client

The virtual client takes charge of metadata maintaining, access scheduling, decoding/encoding processing, the overhead estimation of metadata querying and storage server connection, and request cancellation.

The access scheduler selects disks to access and sends requests to corresponding virtual disks. For each access, the scheduler randomly selects a certain number of disks and randomly permutes the disks into a random order. The order of the disks is not a concern in RobuSTore, since the blocks stored in disks are erasure-coded blocks which are used symmetrically during decoding. For storage schemes that use plain-text blocks, however, the order of the disks affects access performance. Once the access scheduler decides which disks to use, it sends data requests to the corresponding virtual disks. In schemes using speculative access, such as RobuSTore, the scheduler sends single requests for all the available data blocks,

while in schemes using adaptive accesses, such as RRAID-A, multi-round requests are sent in order to adapt to the disk performance variation.

The virtual client is also responsible for maintaining the metadata. For simplicity, the simulator does not include a dedicated metadata server. Therefore, the virtual client should maintain the data location information, including which disks the data are stored in and which sectors on each disks are used. The information is used to accurately simulate the cache behavior and to simulate the read operations after writing the same data.

Furthermore, the virtual client should model the overheads for accessing the metadata server and setting up connections with the disks. Each access to these services by the client is modeled as a constant latency of five milliseconds.

Virtual Filer

The virtual filer is one part of the virtual server. It models the network latency between client and server, and maintains the filesystem cache. Since network bandwidth is presumed to be plentiful [6] in RobuSTore, the network is modeled as a link with fixed round-trip latency. Every time the filer receives one data request from the client, it delays for a certain amount of time to simulate the network latency. This simple mechanism works well since network bandwidth is abundant and the variation in network performance is usually much smaller than that of disk performance. Note that the latency is applied per data request instead of per data access. For schemes like RRAID-A that use adaptive accesses, a data access may include multiple rounds of data requests, which thus involves the delay of multiple round-trip-times (RTTs).

After network delay estimation, the filer checks if the requested data are in-cache. If the data are in-cache, the filer directly sends the data to the client at a rate decided by the maximum network speed; if the data is not in cache or is only partly in cache, the filer requests the missing data blocks from the corresponding virtual disks.

Virtual Disk

The virtual disk is the other part of the virtual server. Each virtual disk is a block-level simulator that simulates the complex behavior of hard disk drives. The hard disk drives have complex behavior due to the disk caching and disk rotating and a block-level simulator is required to get detailed information on caching and disk rotation. There are several block-level disk simulators, including those from Ruemmler and Wilkes [62], Kotz [63], Nieuwejaar [64], and Ganger [65]. All these tools model disk behavior on head-switch time, track-switch time, SCSI-bus overhead, controller overhead, rotational latency, and disk cache.

We built our disk simulator based on DiskSim. The DiskSim [65] from CMU is the most popular block-level disk simulator and is claimed as the most accurate one. It includes an internal synthetic workloads generator and hooks for inclusion in a larger scale system-level simulator. Parameters of DiskSim define bus, controller, cache, and disk (rotation rate, sector-level disk structure), which can be customized to model a wide range of commercial disks. It has been used extensively in the study of new filesystem and storage techniques [66-68]. Therefore, our disk simulator is implemented based on DiskSim and utilizes the DiskSim simulation engine to drive the event simulation in bus, disk controller, and disk rotation and seeking. Also, we

use the DiskSim synthetic workload generator to generate random access sequences inside disks.

Besides the original functions provided by DiskSim, we also implement new functions to support multiple competitive accesses to a single hard disk, to cancel the pending requests, and to synchronize with other virtual disks and the virtual client. First, we implement a background workload generator, which generates requests in a specified rate and pattern. The background requests are injected into the same queue together with the foreground requests and processed by the DiskSim simulation engine. Secondly, since all the unprocessed requests are kept in queue, we can easily implement the request cancellation mechanism by removing the corresponding requests from the message queue. Finally, each virtual disk has a timer to help keep synchronization with other simulation processes. The timer maintains both real clock time and simulated clock time of the previous event. When a new event arrives, the timer checks how much time has passed in the real clock and the simulated clock. If the real clock is slower, the timer stops the simulation for a certain time before dismissing the new event and resuming the simulation.

6.2.3 Metrics

In our experiments, we measure RobuSTore and other conventional storage systems in three metrics.

Variation of Access Latency: A critical RobuSTore goal is robust performance, i.e., minimum performance variation. We formalize this for access latency by

computing the standard deviation over a set of one hundred accesses. Naturally, smaller standard deviations correspond to higher degrees of robustness.

Access Bandwidth: While robust performance is the major goal of RobuSTore, we must also maintain high access bandwidth for the requirement of accessing large datasets. The delivered bandwidth for a single read or write is the original data size divided by the access latency, including connection, disk, data transfer, and coding time. We interpret access bandwidth to be a measure of delivered performance corresponding to our goal of "high performance".

I/O Overhead: The benefits of aggressive access to redundant copies can yield performance benefits, but it also increases network and disk I/O costs. We measure this increased I/O cost using the ratio between the additional bytes sent over networks and the original data size:

$$\text{I/O Overhead} = \frac{\text{Bytes sent over networks - Original data size}}{\text{Original data size}}.$$

Note that the bytes sent over networks may be more than the bytes read from disks if some bytes are read from the filesystem cache.

We measure both read performance and write performance in these three metrics.

6.2.4 Workloads

Since our focus is on supporting the needs of applications with large workloads [1-3], we use synthetic workloads with sequences of large-size accesses. In these applications, each data object is from 100s of MB to 10s of GB, and with the potential

to increase to 100s of GB or larger in the future. We study access performance for single 128 MB, 256 MB, 512 MB, and 1 GB accesses. Data objects larger than 1 GB are presumed to be accessed by multiple 1 GB accesses. There are both read sequences and write sequences accesses and sequences with mixed read and write operations.

Moreover, to simulate disks shared by multiple applications, we generate competitive background workloads for each disk. The background workload is a sequence of random accesses arriving in a certain interval. By varying the interval of the background workload, we can simulate different degrees of disk sharing.

6.2.5 Simulation Configurations

6.2.6 Simulation Parameters

Simulation Hardware Environments

All of our experiments involve a client and a subset of a wide-area storage system with 128 disks. The disks are attached to 16 filers. Each filer maintains a 2 GB filesystem cache shared by the eight disks attached to it. We model the cache as LRU (Least-Recently Used) based and four-way associative with a 4 KB cache line. We only simulate data caching in read accesses, and presume a write-through mechanism for write accesses.

The networks between the client and the servers are presumed to have plentiful bandwidth [6] and are modeled as fixed round-trip latencies varying from 1 millisecond to 100 milliseconds. This latency range covers the scenarios from LAN (less than 1 ms), campus network (less than 1 ms), and metropolitan area network (about 1 ms) to wide area network (up to 100s of milliseconds). Each storage server

access in RAID-0, RRAID-S, or RobuSTore involves one round-trip-time (RTT), while RRAID-A involves multiple RTTs.

We model sector-level disk behaviors using the DiskSim-based simulator. DiskSim parameters define bus, controller, cache, and disk (rotation rate, sector-level disk structure), which can be customized to model a wide range of commercial disks. We configure the DiskSim parameters based on the calibration of a hard drive on a local dual-Xeon server. It is a 120 GB IBM Deskstar 7K400 hard drive, with ATA-100 interface and 7200 rpm rotation speed. We use one DiskSim process to simulate each hard disk drive.

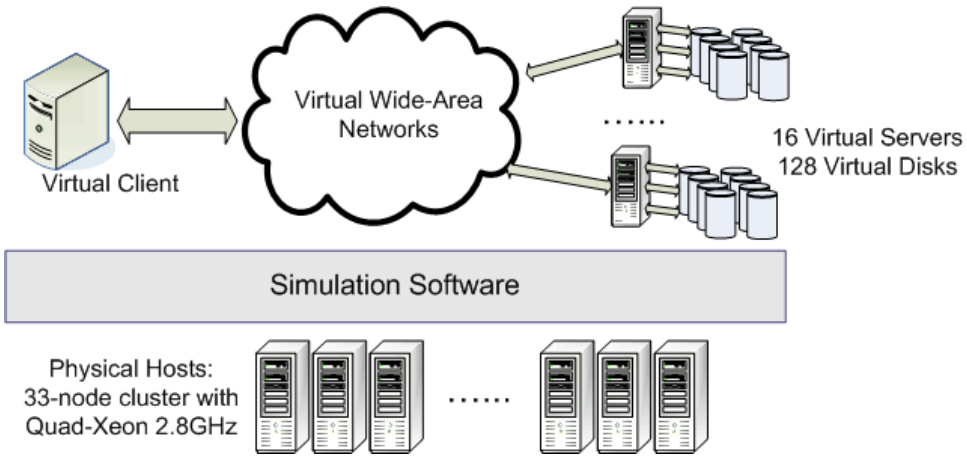


Figure 6-4. Experiment System Configuration.

The data in each access has a random intra-disk layout. In DiskSim, the layout is modeled using two parameters: *blocking factor* and *probability of sequential accesses*. Blocking factor defines the average data size per disk access. Blocking

factor is the average number of sectors per request. Larger blocking factor implies larger data to be accessed with each time overhead of command processing and disk head positioning, thereby delivering high access bandwidth. The probability of sequential accesses specifies the probability that a generated request is sequential to the immediately previous request. A sequential request starts at the address immediately following the last address accessed by the previously generated request, so it avoids the disk head positioning overhead. For each disk, we randomly choose a blocking factor from 8, 16, …, 512, and 1024, and randomly choose 0 or 1 as the probability of sequential accesses. Such different disk configurations lead to different disk performance. The configuration and measured average bandwidth of each disk are shown in Table 6-1. The average of disk bandwidth is 14.9 MBps. The resulting 100-fold performance (0.52 MBps to 53 MBps bandwidth) approximates a shared distributed storage environment with many sources of variation.

Table 6-1. Average Disk Bandwidths with Various In-Disk Layout Configurations.

(MB per second)

Blocking Factor / **Prob. Of Seq Access**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
0	0.52	0.76	1.3	2.5	4.7	8.3	14.3	21.4
1	3.6	6.9	9.3	12.7	16.8	29.8	53.0	53.0

The background workloads are generated by a separate generator. They are sequences of midsize requests, with about 50 sectors on average per request. We used sequences with different intervals to model different levels of the competitive loads. Our simulation shows that when the interval is 6 ms, the background workloads can utilize about 93% of the disk time. In the experiment, we changed the interval from 6 up to 200 milliseconds. Figure 6-5 shows the average disk utilization by the background workload with different intervals and the corresponding foreground access performance. When the background requests arrive about every 6 ms, the foreground access bandwidth is only 2.2 MBps. When the background requests arrive less frequently, the foreground bandwidth gets higher. The bandwidth corresponding to 200 ms of background request interval is about 43 MBps. The average bandwidth, if the background request interval is uniformly distributed between 6 ms ~ 200 ms, is 35 MBps.

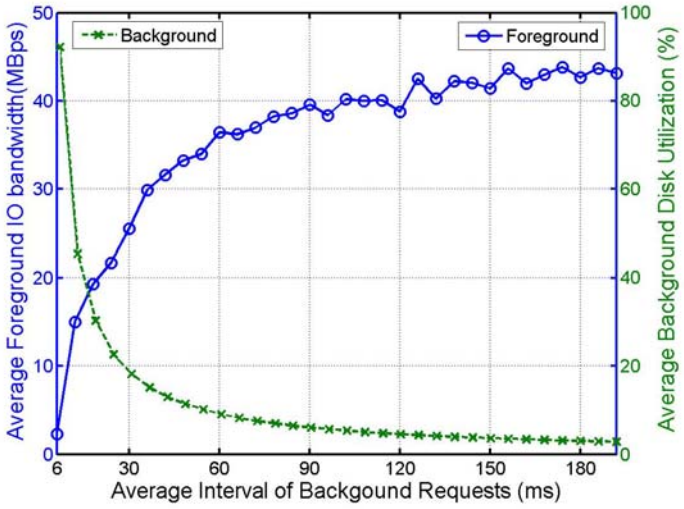


Figure 6-5. Performance Impacts from Background Workloads.

Erasure Coding: Coding and decoding performance is critical for RobuSTore. In each simulation, we first run the LT Codes algorithms to generate an LT coding graph (a bipartite graph connecting original blocks and coded blocks), then randomly select coded blocks and feed them to the LT decoding algorithm to find out the required number of blocks for complete decoding. The simulation of reading process succeeds once enough blocks are received. We use the following LT Codes parameters to generate the coding graph: $C = 1.0$ and $\delta = 0.5$ (C and δ are the parameters for the distribution of node degrees in the LT coding graph: when C increases, there will be more low-degree nodes, and δ has the opposite impact. More details are available in [48]). Previous discussion on LT codes in Section 5.2 shows that the reception overhead is typically 0.5, i.e., 1.5 times the number of blocks need to be received for a successful decoding. Since the decoding process can be overlapped with data I/O, it only incurs extra latency for decoding the last block. Our experiments show that the decoding speed of 500 MBps is possible with typical processors, so we use that rate to compute decode times. For example, for blocks of 1 MB, we add a constant latency of 2 ms.

6.2.7 Exploring Combinations of Simulation Parameters

The combination of all the parameters discussed above is a huge space. There are four storage schemes (RAID-0, RRAID-S, RRAID-A, and RobuSTore) to study. Each scheme may experience the performance variation from three different sources including in-disk data layout, competitive disk loads and filesystem caching. Each access can be configured differently, by using different degrees of data redundancy,

accessing different numbers of disks, using different sizes of coding blocks, or choosing disks with different network latencies. Furthermore, we need to study the behaviors of both read accesses and write accesses. The complete set of all the possible combinations of the parameters is a huge set, thereby hard to explore exhaustively.

We use a fixed configuration as a baseline, varying only one parameter in each experiment. By varying one parameter each time, we can explore all the parameters with a reasonably small set of configurations. Moreover, this method allows us to separate the performance impact of each parameter from other parameters and to study them one by one. The baseline is a typical SAN configuration: 64 disks, 1 ms network round-trip time (RTT), 1 MB block size, and 3x data redundancy, except for RAID-0 which always has 1x data redundancy.

We present the results for each of the above configurations against the performance variation due to the in-disk data layout; for the variation due to competitive workloads and filesystem caching, only the results for different data redundancy are presented since they are the most interesting ones. The configurations of the presented results are shown in Table 6-2.

For each configuration we simulate 100 times of accesses on each of the four storage systems and present the average and the standard deviation; in each access, disks are randomly selected if the experiment uses less than 128 disks.

Table 6-2. The Configurations of the Presented Results. (R: read; W: write; RaW: read after write)

Source of Variation / Configuration	**Data Layout**	**Competitive Workloads**	**Filesystem Caching**
Number of Disks	R		
Block Size	R		
Network Latency	R		
Data Redundancy	R, W, RaW	R, W, RaW	R

6.3. Experiment Results

We present the experiment results in this section, organizing the results into three subsections according to different sources of performance variation. In 6.3.1, we study the system behaviors against random in-disk data layouts. Section 6.3.2 shows the results against random competitive workloads on each disk. Section 6.3.3 shows the results against filesystem caching.

6.3.1 Impact of Data Layout Variation

In this section, we simulate the storage systems with performance variation from in-disk data layout. Since read is the dominant data operation in many data-intensive applications, we first study read performance for all different configurations. After that, we study the write performance to understand the benefit of

speculative writing in RobuSTore. Further, since speculative writing leads to unbalanced data striping among the disks, we also study read performance with unbalanced striping.

Varying Number of Disks

We simulate the four storage schemes discussed above with the baseline configuration and then vary the number of disks from 2 to 128. Figure 6-6 depicts the average bandwidths. First, it shows that RAID-0 exhibits the worst bandwidth, RRAID-S is the second worst, and the best performance is given by RobuSTore. This bandwidth gap grows as the number of disks is increased. RAID-0 suffers because it exploits no redundancy, and thus is subject to the slowest disk. RRAID-S does better because speculative access allows fast disk responses to mask the slow disk responses. However, the speculative access in RRAID-S wastes a lot of bandwidth on accessing replicated blocks; this is avoided in both RRAID-A and RobuSTore. RRAID-A accesses blocks selectively; and RobuSTore uses erasure-coded blocks. Both the RobuSTore and RRAID-A schemes better tolerate the slow disks, with performance increasing with the number of disks. RobuSTore is slightly worse than RRAID-A for small numbers of disks (<8) due to the 40% reception overhead in LT decoding, but outperforms RRAID-A for large numbers of disks due to the greater flexibility on utilizing all the stored blocks. RobuSTore achieves 15 times the bandwidth of RAID-0 for 16–128 disks. The access bandwidth to 1 GB data from 64 disks is as follows: 31 MBps in RAID-0, 117 MBps in RRAID-S, 228 MBps in RRAID-A, and 459 MBps in RobuSTore.

Second, although all four storage systems achieve higher bandwidth by accessing more disks, only RobuSTore achieves linear improvement. RAID-0, RRAID-S, and RRAID-A only achieve sub-linear improvement. The linear improvement in RobuSTore comes from the symmetric redundancy of the erasure-coded blocks, which allows the client to reconstruct the original data from any set of received blocks.

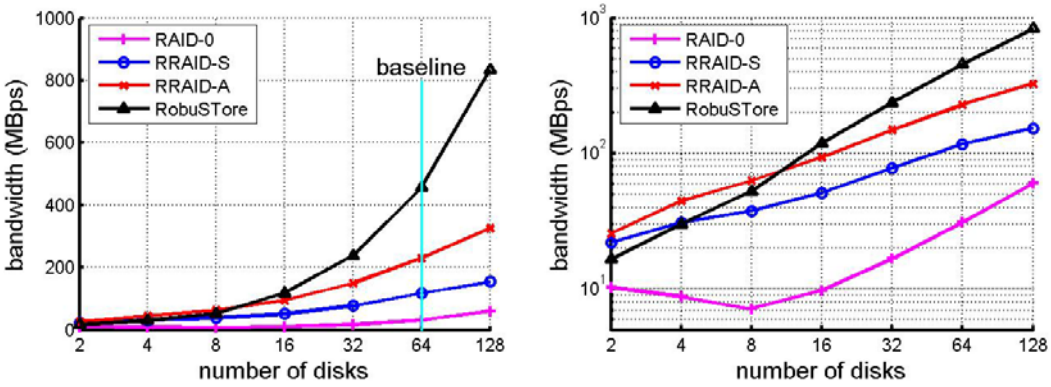
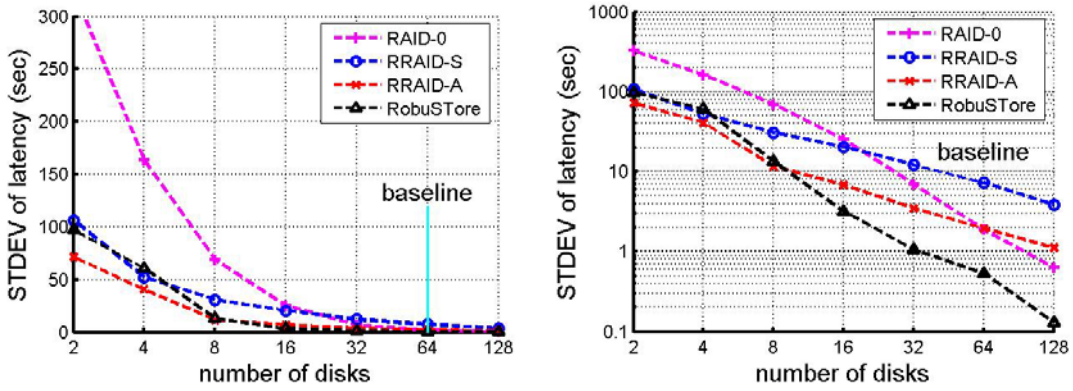


Figure 6-6. Read Bandwidth vs. Number of Disks with Heterogeneous Layout

When we vary the number of disks from 2 to 128, the standard deviation of latency changes, as depicted in Figure 6-7. It shows that RobuSTore has the lowest variation for systems with more than a few disks (>8). To distinguish the details, we show the results as a log-log plot in Figure 6-7(b).

The traditional storage schemes have higher variation due to the lack of access flexibility. RAID-0 suffers because it exploits no redundancy, and the performance is thus subject to the slowest disk. RRAID-S explores the replicated data to hide the slow

disks; however, it reads the blocks on the same disk in a fixed order, so its performance depends on 1) *intra-disk block ordering*: for example, if a replica of a block is stored as the last block on a fast disk, it will be read last from the disk and contributes less to hiding slow disks; and 2) *inter-disk block mapping*, i.e., which disk a block is stored on; if the replicas of the same block are all on slow disks, they cannot hide the slow disks. RRAID-S has the highest variation due to the combination of these factors. RRAID-A mitigates the dependency on intra-disk block ordering by accessing blocks selectively, but it is still dependent on inter-disk block mapping. For a small number of disks (<8), the replicated schemes RRAID-S and RRAID-A have comparable robustness to RobuSTore. As the data redundancy rate is fixed to 3x and few disks are used, the system is essentially performing whole-file replication and suffers a low level of inter-disk dependence.



(a) (b)

Figure 6-7. Variation of Read Latency vs. Number of Disks with Heterogeneous Layout.

Accesses in RobuSTore have significantly less variation. RobuSTore uses erasure-coded blocks and has greater flexibility on using all the stored blocks. Its performance variation comes from the varying total bandwidth of all the disks and the varying reception overhead of LT Codes. It is not related to intra-disk ordering or inter-disk mapping at all. RobuSTore has the lowest performance variation for systems with more than a few disks (>8). The standard deviation of access latency on 64 disks for RAID-0, RRAID-S, RRAID-A and RobuSTore is 1.9, 7.3, 1.9, and 0.5 seconds respectively; and 0.63, 3.8, 1.1, and 0.13 seconds on 128 disks. RobuSTore improves robustness for up to 5x compared to RAID-0, and more than 15x compared to RRAID-S. This lower variation of RobuSTore demonstrates the benefits of erasure-coding and the resulting order-freedom.

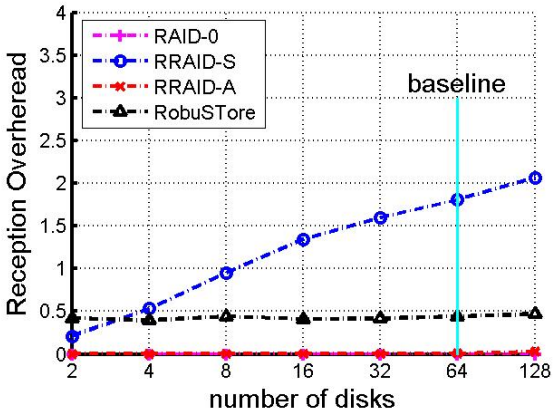


Figure 6-8. Reception Overhead vs. Number of Disks with Heterogeneous Layout.

The benefits of aggressive access to redundant copies can yield performance benefits, but it also increases network and disk I/O costs, as shown in Figure 6-8. Because RAID-0 has no speculative access, it incurs no additional costs, and has zero I/O overhead. RRAID-A costs just a little bit more than zero overhead, as it only generates additional accesses when they are clearly needed. RRAID-S uses a large number of speculative requests, thereby costs I/O overhead as high as 200%. RobuSTore has about 40% I/O overhead due to the requirement of extra blocks for decoding. Although RobuSTore also uses speculative access, its use of erasure codes avoids fetching duplicated blocks, showing perfect parallelism.

Varying Block Granularity

Block granularity affects the performance of RobuSTore and has no impact on other schemes. In RAID-0, RRAID-S and RRAID-A, data blocks are replicated in plain-text, so an access can use fractions of data blocks and ignore block boundaries. However, in RobuSTore, the coded blocks are not replicated and only whole blocks can be applied to block-XOR operations for decoding, so block granularity affects the RobuSTore performance. We vary block size from 0.5MB to 64MB in our experiments.

Figure 6-9 depicts the read bandwidth of the storage schemes. RobuSTore bandwidth decreases as block size grows. This is due to two facts: (1) larger block size increases the “wasted” bytes from those uncompleted blocks; (2) larger block size reduces the effectiveness of pipelining by causing a larger decoding overhead to contribute to access latency. The exception case is that found with a block size of 0.5

MB. In this case, the number of blocks is 2048, much larger than other cases, leading to higher reception overhead in LT Codes.

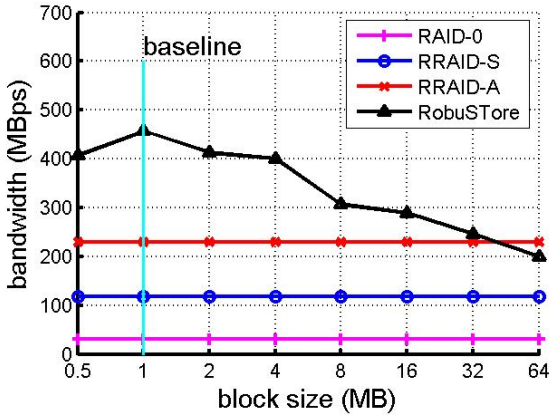


Figure 6-9. Read Bandwidth vs. Block Size, with Heterogeneous Layout.

The STDEV of access latency in RobuSTore increases slightly as block size grows, as shown in Figure 6-10. This is because the larger block size decreases the rate of reading the blocks, so if any block is delayed, the client will potentially wait longer for the next arriving block, i.e., the access latency is more sensitive to block delay. Furthermore, there are fewer blocks with large block size, which causes higher variation in the reception overhead of LT Codes.

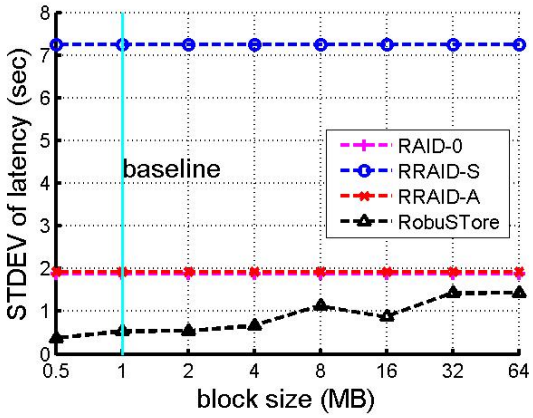


Figure 6-10. Variation of Read Latency vs. Block Size, with Random Layout

The impact of block size on I/O overhead is depicted in Figure 6-11. In this case, the overhead for RobuSTore increases with block size, but does not reach the levels of RRAID-S. As block size increases, larger "uncompleted" blocks are in-the-fly at the moment when the request is completed, increasing the I/O overhead.

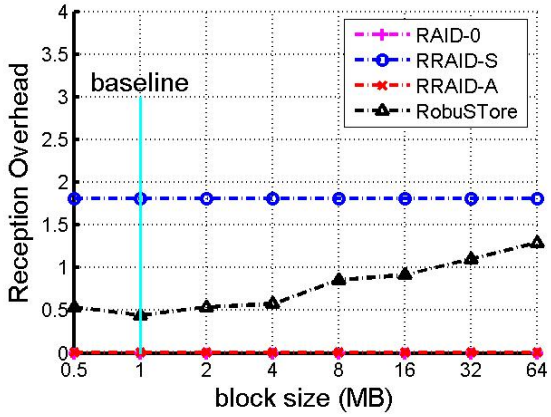


Figure 6-11. Reception Overhead vs. Block Size, with Heterogeneous Layout.

Varying Network Latency

Distributed storage systems usually have networks with a variety of latencies, thus a client can choose to stripe the data to disks with different latencies. To understand its performance impact, we vary network latency between client and storage servers from 1ms (machine room or Metro) to 100 ms (intercontinental). The resulting bandwidths to access 1 GB and 128 MB data segments are depicted in Figure 6-12(a) and Figure 6-12(b) respectively. For storage schemes using speculative access, including RAID-0, RRAID-S, and RobuSTore, network latency has little impact on the access bandwidth. This is because they only involve single round-trip-time (RTT) during the access that is much less than the total access latency of 2~30 seconds. In contrast, RRAID-A is more sensitive to network latency since the adaptive access mechanism involves multiple RTTs. When accessing a 1 GB data segment, RRAID-A suffers 30% bandwidth decrease as network latency increases, changing from 228 MBps to 161 MBps; for smaller requests of a 128 MB data segment, the bandwidth decrease is more significant, dropping 52% from 131 MBps to 63 MBps.

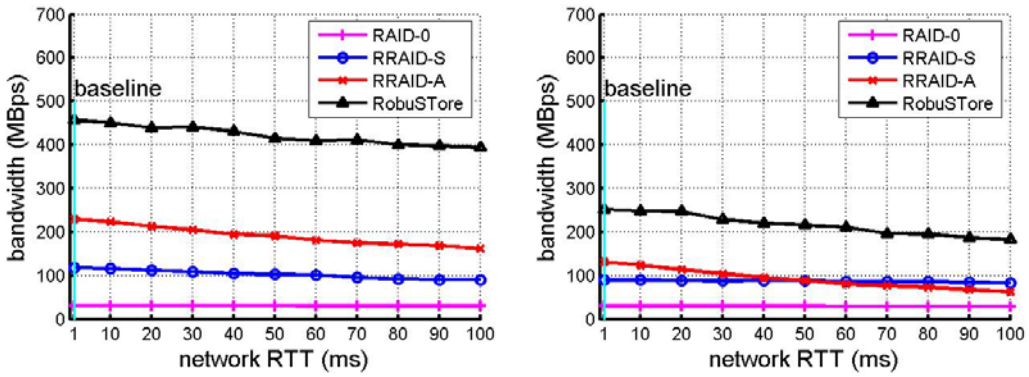


Figure 6-12. Read Bandwidth vs. Network Latency, with Heterogeneous Layout. (a) 1024 MB data access; (b) 128 MB data access.

Our results in Figure 6-13 show that network latency has a negligible impact on performance robustness. In RAID-0, RRAID-S, and RobuSTore, their accesses only involve one-time read requests, so the impact of varying network latency is at most one RTT on total access latency. RRAID-A uses adaptive access strategy and involves multi-RTT on total access latency. Considering that the total access latency is 2~30 seconds, the variation from the network is much less than that from disks in our model.

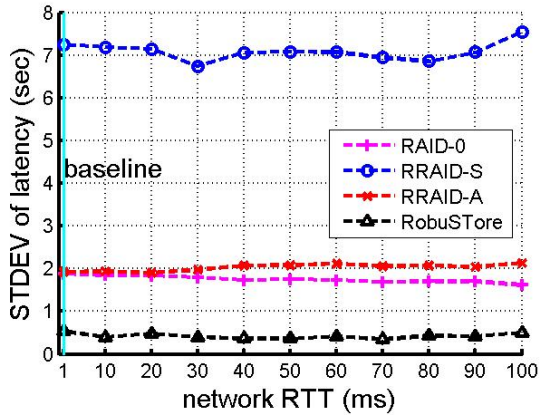


Figure 6-13. Variation of Read Latency vs. Network Latency, with Heterogeneous Layout.

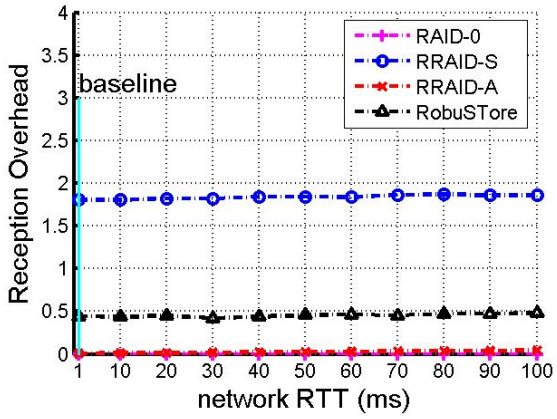


Figure 6-14. I/O Overhead vs. Network Latency, with Heterogeneous Layout.

Similarly, increasing network latency has minimal impact on I/O overhead, as shown in Figure 6-14. Higher network latency only increases the amount of in-the-fly bytes at the moment when the access is completed, which is proportional to the

network latency. Since the variety of network latency is much less than the total access latency, the in-the-fly bytes are only a small fraction of the total I/O cost.

Varying Degree of Data Redundancy

Data redundancy is the direct overhead in storage spaces; however, redundancy can also be used to increase read performance. We vary data redundancy from 0 to 900% (10 times the storage spaces used) to simulate its performance impact. Because the RAID-0 scheme always has zero redundancy, its performance is represented by the point of zero redundancy in RRAID-S or RRAID-A.

The access bandwidth is depicted in Figure 6-15. Increasing data redundancy increases the bandwidths in all the storage schemes, but to different extents. For RobuSTore, the bandwidth increases rapidly and approaches the best performance when the redundancy is higher than 200%. The peak performance is achieved when the redundancy is higher than 500%, i.e., when six times storage space is used. Considering that the fastest disk delivers a bandwidth of four times the average, and the reception overhead of LT Codes is about 40%, this proves our previous analysis that RobuSTore achieved best performance when there are enough blocks to read on every disk during the entire access period. RRAID-S and RRAID-A benefit less in their access bandwidths from high redundancy. This is because their structured data replication cannot adapt to read more blocks from the faster disks as flexibly as in RobuSTore.

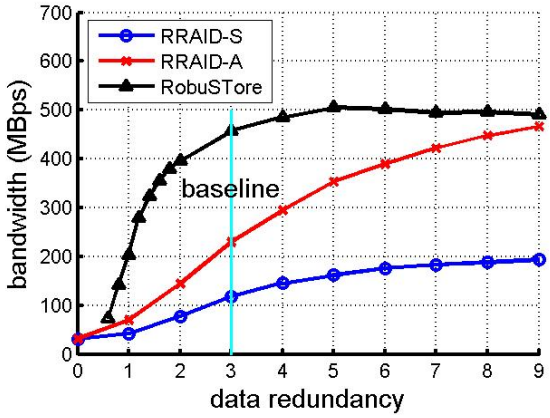


Figure 6-15. Read Bandwidth vs. Data Redundancy, with Heterogeneous Layout.

The impact of data redundancy on access robustness is depicted in Figure 6-16, which shows that RobuSTore achieves the lowest standard deviation of latency. In RRAID-S and RRAID-A, the variation comes from disk speed, intra-disk block ordering (in RRAID-S), and inter-disk block mapping. When they use higher data redundancy, their robustness will potentially suffer less from disk speed variation and inter-disk block mapping, while suffering more from intra-disk block ordering. RAID-0 only suffers variation from the slowest disk. Due to the combination of these factors, RRAID-S and RRAID-A with small redundancy have worse robustness than RAID-0, and gradually get better as redundancy increases. In RobuSTore, as long as the fast disks have enough data blocks, they can hide the slow disks effectively. It needs only 1x~2x data redundancy to obtain most of this robustness benefit. When data redundancy is more than 2x, the standard deviation of latency is only about 0.5 seconds, or 25% of the average access latency.

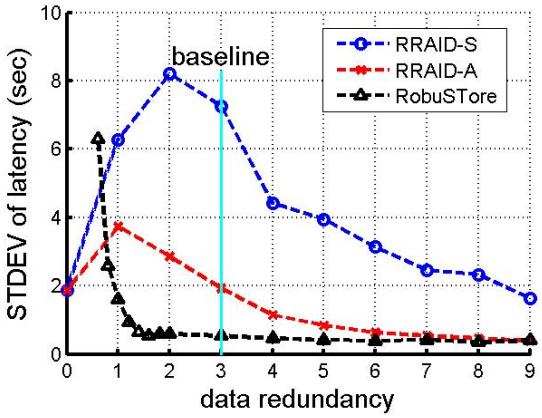


Figure 6-16. Variation of Read Latency vs. Data Redundancy, with Heterogeneous Layout.

As for I/O overhead, the data redundancy only affects RRAID-S. When data redundancy is increased, both RRAID-S and RobuSTore increase the requested data size in proportion. For RobuSTore, the access is completed as long as a certain number of coded blocks are received, so the final I/O overhead is mainly decided by the reception overhead of LT Codes. However, in RRAID-S, high data redundancy lets the client receive more duplicated data blocks, leading to high I/O overhead. The results are shown in Figure 6-17.

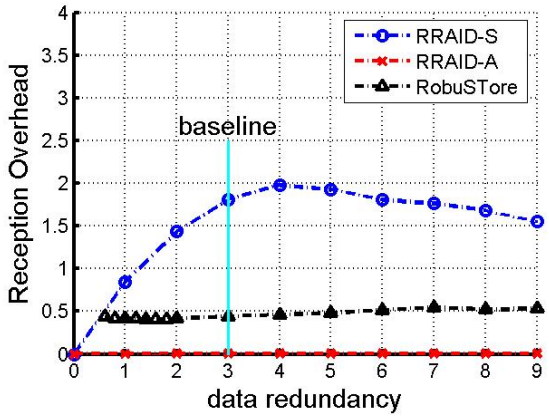


Figure 6-17. I/O Overhead vs. Data Redundancy, with Heterogeneous Layout.

Next, we study the write performance with various data redundancy. In the schemes of RAID-0, RRAID-S, and RRAID-A, a write operation uniformly writes the same number of blocks to each disk, and behaves similar to a read in RAID-0, except that the involved data size is multiple times larger in RRAID-S and RRAID-A. RobuSTore uses speculative access so different numbers of blocks are written to each disk according to the disk performance. A write in RobuSTore is similar to a read with unlimited data redundancy so that there are enough blocks to write to every disk until the operation is complete.

The experiment results of write accesses are depicted in Figure 6-18, Figure 6-19 and Figure 6-20. Figure 6-18 shows the writing bandwidth. High data redundancy requires writing more bytes, thereby resulting in low writing bandwidth. In RAID-0, RRAID-S, and RRAID-A, the write bandwidth is very low because it is limited by the slowest disk. RobuSTore achieves much higher bandwidth since its speculative writing can efficiently utilize the capability of all the disks. When the data

redundancy is 300%, the write bandwidth in RobuSTore is about 186 MBps, while RRAID-S and RRAID-A only deliver bandwidth of 7.5 MBps. It is 30 MBps for RAID-0 (with zero redundancy).

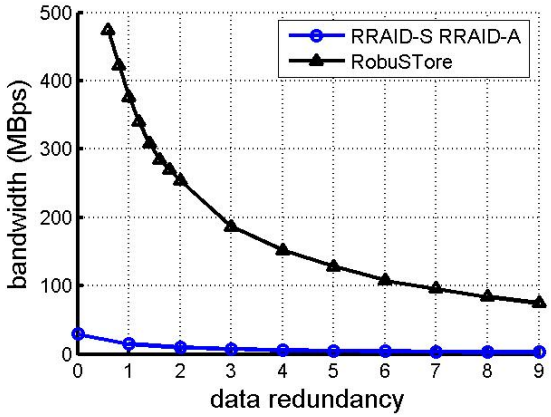


Figure 6-18. Write Bandwidth vs. Data Redundancy with Heterogeneous Layout.

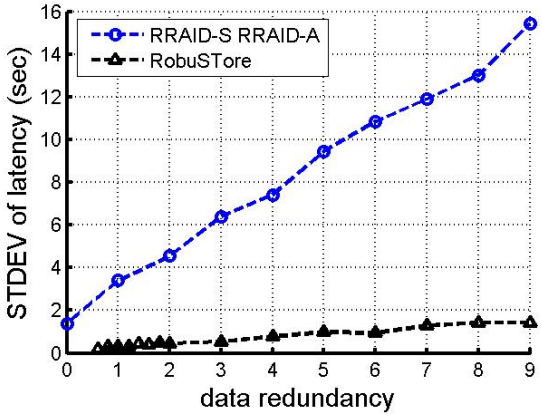


Figure 6-19. Variation of Write Latency vs. Data Redundancy with Heterogeneous Layout.

The standard deviation of write latency is more than 10 times better in RobuSTore than in RRAID-S and RRAID-A, as shown in Figure 6-19. For example, when the data redundancy is 300%, the standard deviation is 0.5 seconds for RobuSTore and 6.4 seconds for RRAID-S and RRAID-A.

For write accesses, the I/O overhead is proportional to data redundancy because a write operation needs to write every byte of the redundant data. RobuSTore may incur slightly more overhead due to the usage of a speculative writing mechanism. The simulated results are shown in Figure 6-20.

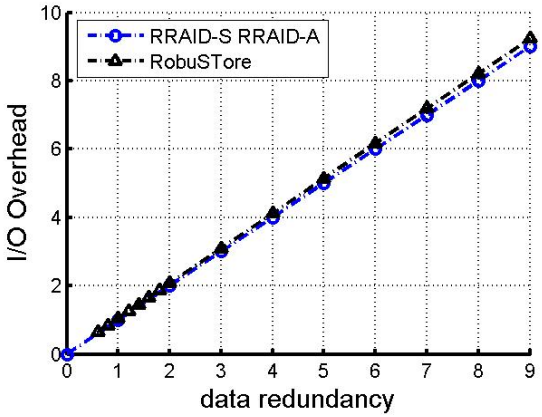


Figure 6-20. I/O Overhead vs. Data Redundancy with Heterogeneous Layout.

Next, we study read accesses in RobuSTore with unbalanced data striping. In the beginning of the section we had already simulated read accesses with balanced data striping. However, in RobuSTore, speculative writing may cause unbalanced data striping across multiple disks. Fewer data blocks are written into a disk if the disk is

slow during the write accesses. Since the individual disk performance is dynamically changing in shared distributed environments, it is possible that the disk delivers higher performance in later read accesses. In RobuSTore, such disks are more quickly exhausted during the read accesses so that their bandwidth cannot be fully utilized. RAID-0, RRAID-S, and RRAID-A have no such issue since they always use balanced data striping.

The simulated results are depicted in Figure 6-21, Figure 6-22 and Figure 6-23. For comparison, the figures also show the read results for RRAID-S and RRAID-A with balanced data striping. Figure 6-21 depicts the average read bandwidth. By comparing it to Figure 6-15, we see that the RobuSTore bandwidth with unbalanced striping is slightly worse than that with balanced striping. However, it is still higher than RAID-0, RRAID-S, and RRAID-A. The variation of RobuSTore access latency, as shown in Figure 6-22, is the least among the four storage schemes. The unbalanced striping has little impact on RobuSTore I/O overhead that is mainly decided by the LT Codes reception overhead, so Figure 6-23 is almost the same as Figure 6-17.

In real systems, RobuSTore with unbalanced striping may be a little different, with better performance than what we get here. Our simulation experiments use the assumption that the performances of all the disks follow the same statistical pattern, and that each random change is independent of each other. Besides the dynamic performance, the disks in a real storage system usually also have heterogeneous performance. Due to the performance heterogeneity, a disk that is fast during a write operation tends to be fast during later read operations. In such cases, read operations

can benefit from unbalanced data striping and achieve better performance than with balanced data striping.

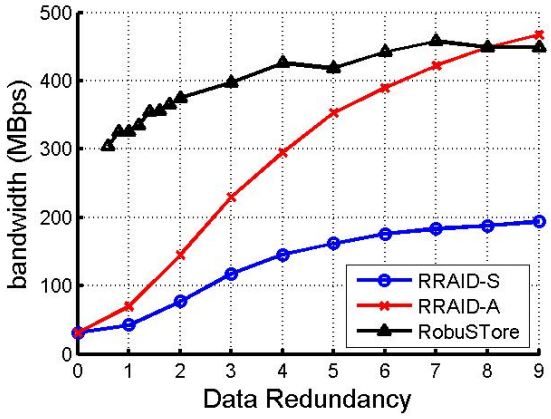


Figure 6-21. Read Bandwidth vs. Data Redundancy with Heterogeneous Layout and Unbalanced Data Striping.

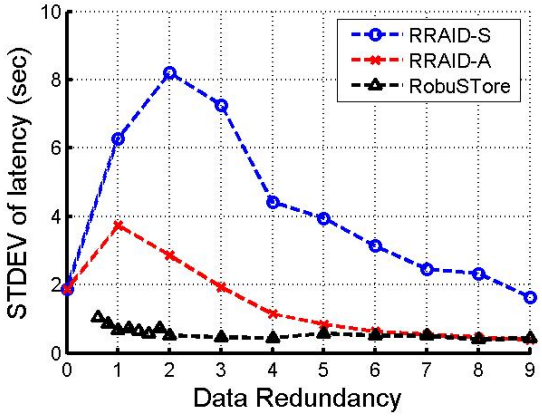


Figure 6-22. Variation of Read Latency vs. Data Redundancy with Random Layout and Unbalanced Data Striping.

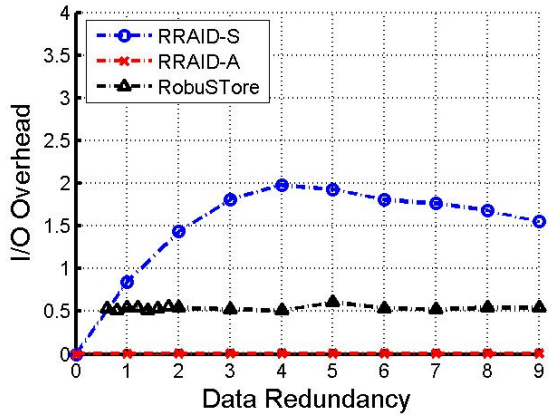


Figure 6-23. I/O Overhead vs. Data Redundancy with Heterogeneous Layout and Unbalanced Data Striping.

6.3.2 Impact of Competitive Workloads Variation

In this section, we study how well the systems can tolerate the performance variation from competitive workloads. In shared distributed environments, multiple applications may access the storage systems simultaneously. If the applications access the same disks, they will compete for the disk bandwidth and affect each other's access latency. In our experiments, we model the competitive workloads as random arrival background requests.

We first study a simple homogeneous scenario in which all the disks follow the same statistical pattern for the competitive workloads. Figure 6-24 and Figure 6-25 depict the results of the read experiments. The performance of all storage systems increases when the background workloads arrive less frequently. In such environment, all the disks have same pattern. The major source of disk performance variation is that accesses to different disk zone achieve different performance. The performance

variation is as much as two. RobuSTore does not achieve as good a performance as the other three systems because of the 50% reception overhead required for LT decoding. However, the performance difference is much less than 50%. For example, RRAID-S has a peak bandwidth of about 1650 MBps, while RobuSTore has about 1360 MBps peak bandwidth, with 18% difference.

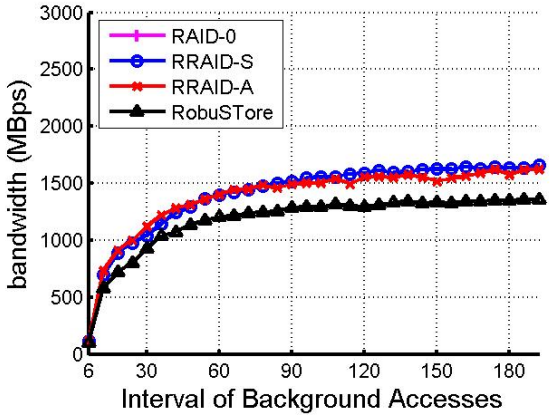


Figure 6-24. Read Bandwidth vs. Competitive Workloads with Homogeneous Layout and Homogeneous Competitive Workloads.

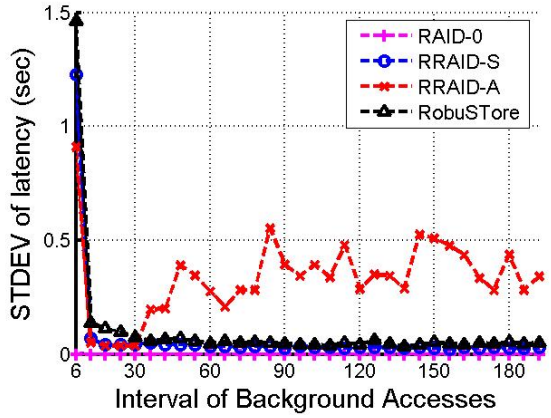


Figure 6-25. Variation of Read Latency vs. Competitive Workloads in Homogeneous Layout and Homogeneous Competitive Workloads.

Next we simulate the read accesses in heterogeneous environments with random competitive workloads. More specifically, every time we simulate a new access, we reset the competitive workload generator randomly for each disk. Figure 6-26 depicts the results with various degrees of data redundancy. In RobuSTore, the read performance quickly increases with the data redundancy. The best performance is achieved with data redundancy greater than 140%. In Section 6.2.4, the performance results show that with different competitive workloads the fastest disk can deliver a peak performance of about 44 MBps while the average performance is about 33 MBps. Further, considering the 50% reception overhead, we again prove the claim that RobuSTore achieves the best performance when the data redundancy is at least the ratio between the fast disk's bandwidth and the average disk bandwidth.

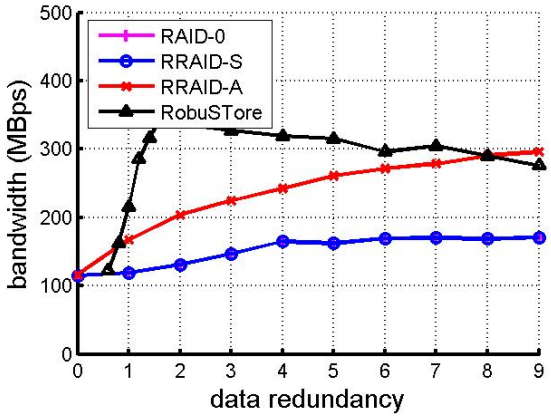


Figure 6-26. Read Bandwidth vs. Data Redundancy with Heterogeneous Competitive Workloads.

Figure 6-27 depicts the standard deviation of latency in systems with random competitive workloads. When the data redundancy is greater than 140%, RobuSTore has much lower variation than RRAID-S and RRAID-A. Figure 6-28 shows the I/O overheads, which is similar to the previous experiments. RobuSTore has about 50% overhead; while RRAID-A has almost zero overhead and RRAID-S has up to 230% overhead.

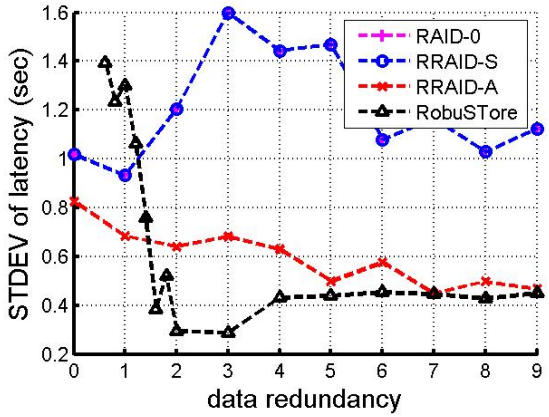


Figure 6-27. Variation of Read Latency vs. Data Redundancy with Heterogeneous Competitive Workloads.

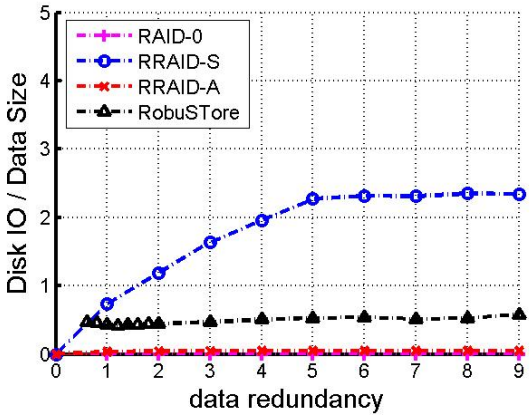


Figure 6-28. Reception Overhead vs. Data Redundancy with Heterogeneous Competitive Workloads.

The results for write accesses are depicted in Figure 6-29, Figure 6-30, and Figure 6-31. The write bandwidth decreases as the data redundancy increases. RobuSTore delivers much higher bandwidth than RAID-0, RRAID-S, and RRAID-A, and significantly less standard deviation of write latency.

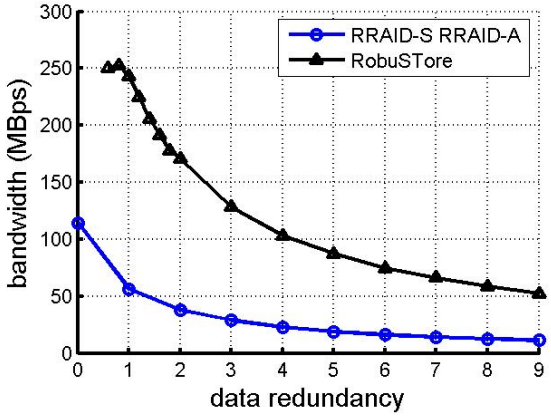


Figure 6-29. Write Bandwidth vs. Data Redundancy with Heterogeneous Competitive Workloads.

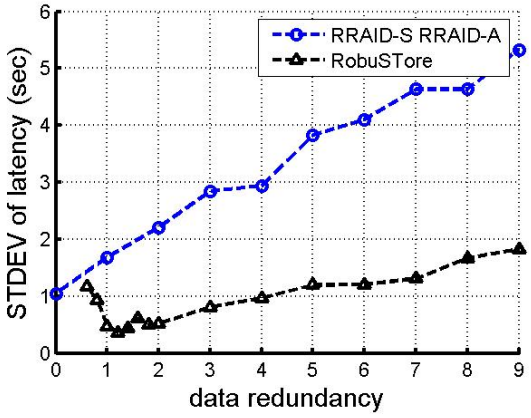


Figure 6-30. Variation of Write Latency vs. Data Redundancy with Heterogeneous Competitive Workloads.

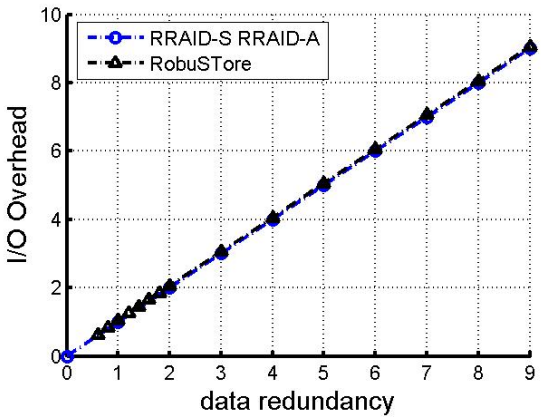


Figure 6-31. I/O Overhead vs. Data Redundancy with Heterogeneous Competitive Workloads.

Next, we study read accesses in RobuSTore with unbalanced data striping. The results are shown in Figure 6-32, Figure 6-33, and Figure 6-34. Figure 6-32 depicts the average read bandwidth, which shows that RobuSTore with unbalanced striping delivers higher bandwidth than RAID-0, RRAID-S, and RRAID-A. The variation of RobuSTore access latency, as shown in Figure 6-33, is the least among the four storage schemes. The I/O overhead of RobuSTore, as shown in Figure 6-34, is about 40~50%, the same as in RobuSTore with balanced striping. This is because the RobuSTore I/O overhead is mainly decided by the LT Codes reception overhead and is related to data striping.

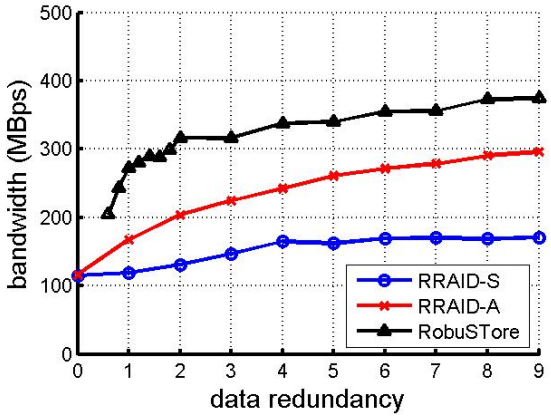


Figure 6-32. Read Bandwidth vs. Data Redundancy with Heterogeneous Competitive Workloads and Unbalanced Data Striping.

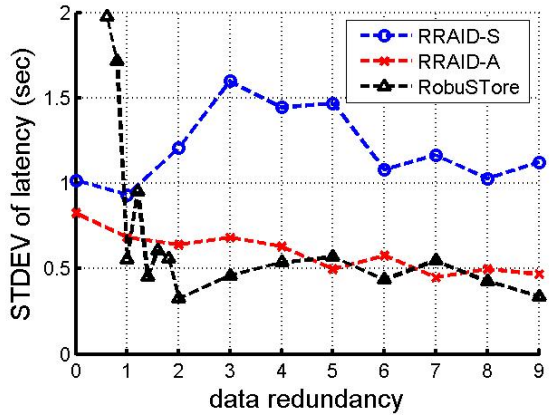


Figure 6-33. Variation of Read Latency vs. Data Redundancy with Heterogeneous Competitive Workloads and Unbalanced Data Striping.

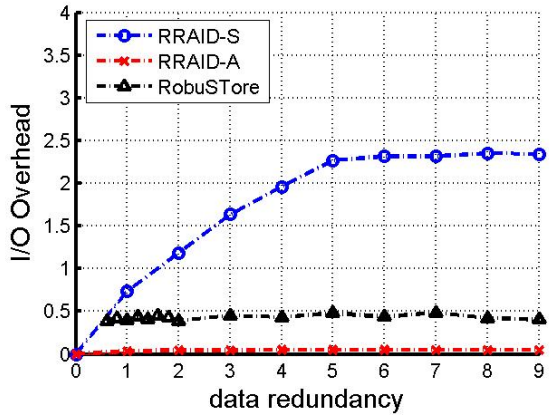


Figure 6-34. I/O Overhead vs. Data Redundancy with Heterogeneous Competitive Workloads and Unbalanced Data Striping.

6.3.3 Impact of Filesystem Caching Variation

Another source of the performance variation is the filesystem caching. A read access gets higher bandwidth if the accesses data are entirely or partly in cache. In our simulation, we presume a write-through mechanism for write accesses, so write accesses are not related to filesystem cache. Therefore, we only studied the cache impact on read accesses.

In the experiments, we run the baseline configuration with random competitive workloads, and simulate 2 GB filesystem cache on each storage server which is shared by all accesses to the eight disks in this server. We compare the results against those without filesystem caching. Figure 6-35 depicts the access bandwidth, which shows that using cache can improve the performance for all the four storage systems. Figure 6-36 depicts the variation of access latency, which shows that using cache causes higher variation of access latency. Among all the four storage schemes, RobuSTore

still performs the best, delivering the highest access bandwidth and the lowest variation of access latency.

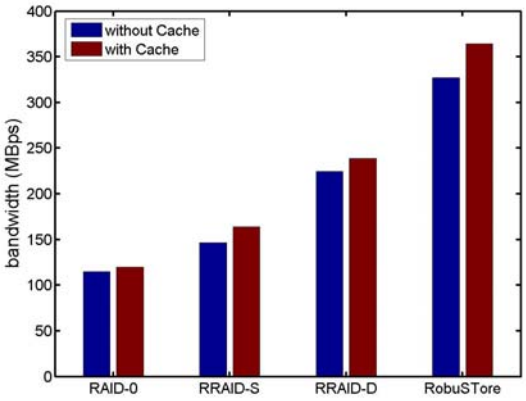


Figure 6-35. Cache Impact on Access Bandwidth.

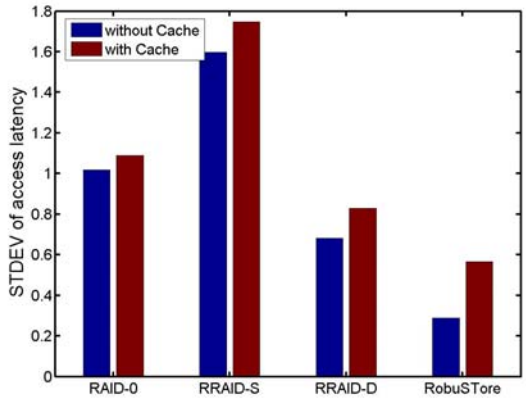


Figure 6-36. Cache Impact on Variation of Access Latency.

6.4. Summary

We compared RobuSTore to RRAID-S, RRAID-A, and RAID-0 across a wide range of system configurations by using detailed software simulation. The simulation

results show that RobuSTore provides best performance while incurring only moderate I/O overheads of about 50% and storage space overheads of 1–2x. For example, to read 1 GB data from 64 disks with random data layout, RobuSTore achieved an average bandwidth of over 400 MBps, nearly 15x that achieved by a baseline RAID-0 scheme. At the same time, RobuSTore achieves standard deviation of access latency of only 0.5 seconds, less than 25% of the total access latency.

6.5. Acknowledgement

Part of Chapter 6 is published as “RobuSTore: Robust Performance for Distributed Storage Systems” by Huaxia Xia and Andrew Chien in the proceedings of 14th NASA Goddard - 23rd IEEE Conference on Mass Storage Systems and Technologies, 2006. The dissertation author was the primary researcher and author of this paper.

Chapter 7. Conclusions

In this chapter, we summarize the research described in this dissertation. Section 7.1 summarizes the RobuSTore idea and highlights the key contributions of our research. Section 7.2 discusses the implications and impacts of our research. Section 7.3 discusses avenues for future work.

7.1. Dissertation Summary

Achieving high and robust performance in distributed storage systems is an important open research challenge. Existing and emerging large-scale data-intensive applications have high level requirements on storage service, including large capacity, distributed applications, high access bandwidth, and robust access latency. Traditional network filesystems or local parallel filesystems cannot satisfy these requirements. The performance variation of the individual disks is the major obstacle facing current systems.

We propose a distributed storage architecture called RobuSTore to meet the storage requirements of these applications. RobuSTore combines erasure coding and speculative access mechanisms for high and robust storage performance. In RobuSTore, the erasure coding mechanism encodes the original data into fragment blocks with symmetric redundancy, allowing flexible data striping during write accesses and flexible data reconstruction during read accesses. The speculative access mechanism fully utilizes the available disk bandwidths to read/write redundant fragment blocks from/to heterogeneous distributed disks. RobuSTore exploits both

erasure codes and speculative access to improve performance robustness and absolute performance. This idea is a general one, and can enable fundamentally better behaved storage systems – with low standard deviations in access times.

To realize the RobuSTore idea, we present a system framework. The system framework explains the major architecture and the functions required to make the RobuSTore idea workable. The framework design shows the feasibility of the RobuSTore idea.

We then study a wide range of critical design choices for RobuSTore implementation and configuration. The choices include those for erasure codes, speculative access, and admission control. First, we compare different erasure codes by analyzing their usage in storage systems. Our analysis shows that near-optimal erasure codes such as LT codes fit RobuSTore the best in delivering high coding throughput and providing long codeword. Second, we analyze the choices of the erasure codes parameters, using LT codes as the example. The best erasure codes parameters reflect the tradeoff between computation overhead, storage space overhead, and read flexibility. Our analysis shows that the LT codes fit RobuSTore the best when (1) the number of original blocks is around K=128~1024, (2) the number of coded blocks is around N=1024~4096, and (3) the coded blocks have an average degree of five in the coding graph. Third, we explore the number of disks that a speculative access should use. Our analysis shows that the number of disks should be no less than the expected total access bandwidth divided by the average disk bandwidth. This is further studied using software simulation, which shows that the

average RobuSTore access bandwidth is proportional to the number of disks, thereby proves our analysis. Fourth, we study the choice of data redundancy in RobuSTore. We first analyze the write and read processes and conclude that RobuSTore should use data redundancy of the ratio between the peak disk performance and the average disk performance. This is proved in our quantitative study using detailed simulation, which shows that RobuSTore gains most of the performance benefits with 2-3x data redundancy. Fifth, we simulate the RobuSTore performance with a range of block sizes in coding. The results show that both too small blocks and too large blocks hurt the access performance. Specifically, 1 MB is the optimal block size in our simulated configurations. Sixth, our simulation experiments show that network latency between the client and the storage servers does not impact RobuSTore performance significantly. Our study on the critical design choices gives a guideline of RobuSTore implementation and configuration.

Furthermore, our simulation experiments prove that RobuSTore can better tolerate disk performance variation than traditional parallel storage schemes and delivers much higher bandwidth and more robust latency. We compare the performance of RobuSTore with three traditional parallel storage schemes and see superior performance from RobuSTore. For example, for a 1GB read using 64 disks with random in-disk data layout, RobuSTore achieves average bandwidth of 400MBps, nearly 15x that achieved by a RAID-0 system. The standard deviation of access latency is only 0.5 second, less than 25% of the access latency, and a 5-fold improvement from RAID-0. The improvements are achieved at moderate cost: about

40% increase in I/O operations and two to three times increase in storage capacity utilization.

7.2. Implications and Impact

Our research implies that RobuSTore is a sound scheme of distributed storage systems. First, the presented RobuSTore framework and the critical choices demonstrate the feasibility of a RobuSTore implementation. Furthermore, our experiments show that RobuSTore can effectively tolerate the performance variation which exists in distributed storage systems. Specifically, the experiment results prove that RobuSTore can potentially deliver access bandwidth that are proportional to the total bandwidth of all the accessed disks, and achieve robust access latency when using a reasonable large number of disks.

A second implication is that RobuSTore is not the best choice in homogeneous storage environments. When there is no significant performance variation among the disks, our experiments show that the read performance of RobuSTore is slightly worse than that of the traditional parallel storage systems like RAID-0. This is because the homogeneous storage systems do not need the capability of erasure codes to tolerate performance variation, while they suffer from the extra reception overhead and decoding overhead from erasure codes with RobuSTore.

Another implication is that abundant hardware resources are required for a RobuSTore system to be effective. Since RobuSTore exploits erasure codes and speculative access, it consumes extra CPU cycles for coding, extra network and disk bandwidth for transferring redundant data, and extra disk space to hold the redundant

data. In our experiments, it takes near 100% of the client CPU time to achieve around 500 MBps decoding bandwidth, 50% I/O overhead for decoding, and 200%-300% redundant data to tolerate the performance variation.

Our research has two major impacts. First, our research provides a foundation for the combined use of erasure codes and speculative access to tolerate performance variation. Although erasure codes have been widely used in communication and storage media for fault tolerance, little study was focusing on exploiting erasure codes for high and robustness performance. On another aspect, while there are a number of local parallel filesystem schemes, none of them fits for large-scale distributed environment. Our study solves the problem by combining erasure codes and speculative access.

Second, our research provides a general RobuSTore framework and configuration guidelines. We present a flexible system framework, and explore the critical design choices using both theoretical analysis and software simulation. The study provides a guideline for the implementation and configuration of RobuSTore, paving the way for the large-scale use of the scheme in many applications.

7.3. Future Work

The research in this dissertation mainly focused on presenting the RobuSTore idea and demonstrating the advantage and feasibility of the idea. While we believe that we have made significant contributions in meeting the goals, more advances can be made to improve the RobuSTore performance, reduce the RobuSTore overhead, and

deploy the RobuSTore system in practice. We briefly discuss these future directions as follows.

Erasure Codes with Higher Performance

With rapidly increasing network bandwidth, we will need erasure codes algorithms that can deliver higher coding bandwidth to match the network bandwidth increase. The LT codes that we implemented achieve around 400 MBps decoding bandwidth on 2.8 GHz AMD Opteron Processor, and have about 50% reception overhead. This is equivalent to around 5 Gbps network utilization and is not enough to keep up with network with higher bandwidth.

Several methods are possible to achieve higher coding bandwidth. First, we can use more efficient erasure codes. Although LT codes deliver fairly good performance, they are only optimized for communication. Storage systems have inherent features different from communication systems. Therefore, it is possible to design codes optimized for storage systems that outperform LT codes. A second method is to design parallel coding algorithms. We can use a cluster of workstations as a coding agent to do parallel coding with high bandwidth. Finally, dedicated coding hardware is another feasible method. Considering that the LT codes algorithms are relative simple, a hardware coder is not hard to implement.

Evaluation for Multi-User Workloads

Efficient multi-user sharing is one of the important goals for distributed storage systems. In our experiments, we model the competitive access using random background requests. This is a simplified model to approximate the shared accesses.

While such a simplified model is helpful in understanding the access performance of one client, a more accurate model of multi-user workloads can definitely help the study to optimize the multi-user performance and the throughput of the entire storage system.

Admission Control

Admission control is important for QoS guarantee and efficient resource sharing. In RobuSTore, we reserve the choice on admission control for future work. This is based on the following considerations. First, the experiments of admission control need good workload models for multiple competitive accesses. However, we do not have good enough workload model or traces, hence experiments on admission control is not very meaningful. Furthermore, different storage sites are likely to have different access control policies, which make it complex to study the admission control. In future study, we can first build good models for multiple competitive accesses and different admission control policies, and then we can study admission control in more details.

Real System Implementation and Configuration

A real system allows experiments with real applications and real testbeds, thereby improving the fidelity of the study. At the same time, real implementation and configuration also examine the system design in practical considerations, for example, the application interface, metadata management, security management across multiple administration domains, etc. Further research can explore these questions on a real RobuSTore system.

Appendix A. Replication vs. Erasure-Coding

We give a complete analysis of the replicated redundancy and erasure-coded redundancy. We assume both methods use 300% data redundancy (four times storage space). The object is to know how many blocks are required to reconstruct the original data using the two methods.

General problem description: Assume we have K original blocks, and we translate them into 4K output blocks using either replication or erasure coding. Now randomly permute the 4K blocks. What is the probability that we can reassemble the original blocks using the first M output blocks?

A1. Plain-text Replication

The problem is equivalent to the following:

Given: 4K balls with K different colors (four balls per color); randomly pick M balls from them

Want: probability of at least one ball per color.

Assume the number of M-ball sets to have at least one ball per color is $F_M(K)$. Then we have:

$F_M(K)$ = (All sets) – (sets with less than N colors)

$$= \binom{4K}{M} - \sum_{i=1}^{K-1} \binom{K}{i} F_M(i), \quad \text{(Let } \binom{a}{b} = 0 \text{ if a<b)}$$

We will prove the following using induction:

$$(A.1) \qquad F_M(K) = \sum_{i=1}^{K} (-1)^{K-i} \binom{K}{i} \binom{4i}{M}$$

First, since there are only 4 balls per color, we have

$F_M(K) = 0$, if $K<M/4$,

which satisfies (A.1).

Now we assume (A.1) is satisfied for any number less than K, then:

$$F_M(K) = \binom{4K}{M} - \sum_{i=1}^{K-1} \binom{K}{i} F_M(i)$$
$$= \binom{4K}{M} - \sum_{i=1}^{K-1} \binom{K}{i} \sum_{j=1}^{i} (-1)^{i-j} \binom{i}{j} \binom{4j}{M}$$
$$= \binom{4K}{M} - \sum_{i=1}^{K-1} \sum_{j=1}^{i} (-1)^{i-j} \binom{K}{i} \binom{i}{j} \binom{4j}{M}$$
$$= \binom{4K}{M} - \sum_{j=1}^{K-1} \sum_{i=j}^{K-1} (-1)^{i-j} \frac{K!}{i!(K-i)!} \cdot \frac{i!}{j!(i-j)!} \binom{4j}{M}$$
$$= \binom{4K}{M} - \sum_{j=1}^{K-1} \frac{K!}{j!(K-j)!} \sum_{i=j}^{K-1} (-1)^{i-j} \frac{(K-j)!}{(K-i)!(i-j)!} \binom{4j}{M}$$
$$= \binom{4K}{M} - \sum_{j=1}^{K-1} \frac{K!}{j!(K-j)!} \sum_{k=0}^{K-j-1} (-1)^{k} \frac{(K-j)!}{(K-j-k)!k!} \binom{4j}{M} \quad \text{(let k=i-j)}$$
$$= \binom{4K}{M} - \sum_{j=1}^{K-1} \frac{K!}{j!(K-j)!} (-(-1)^{K-j}) \binom{4j}{M}$$
$$= \sum_{j=1}^{K} (-1)^{K-j} \binom{K}{j} \binom{4j}{M}$$

So (A.1) also fits for K.

Therefore, the probability of picking M balls to include at least one ball per color is:

$$P(M) = \frac{F_M(K)}{\binom{4K}{M}} = \sum_{i=1}^{K} (-1)^{K-i} \frac{\binom{K}{i}\binom{4i}{M}}{\binom{4K}{M}}$$

A2. Erasure-Coded Case

With parameter of C=1.1 and δ=0.5, the average output-node degree in the LT coding graph is about 5. To simplify the analysis, we assume that all output nodes have degree 5 and their neighbors are independently randomly selected from the N original blocks. The number of blocks to reconstruct the original K blocks is about the number of blocks whose neighbors include all the K blocks. So the probability that M coded blocks are sufficient is the probability that 5M neighbors can cover all the K original blocks. Using similar induction as in the above section, we can prove that:

$$P_c(M) = \sum_{i=1}^{K} (-1)^{K-i} \binom{K}{i} (\frac{i}{K})^{5M}$$

Appendix B. Open Operation of RobuSTore

In RobuSTore, a data access is initiated by an application request to open a file. The application interface to open the file is:

- `FDescriptor open(Filename, AccessType, QoS_option);`

"Filename" is a string specifying the name of the accessed data. "AccessType" indicates if the data access is a read or a write. "QoS_option" specifies the required quality of service, which we will discuss below. The function returns a file descriptor which includes information like data location, coding algorithm, coding parameters, and data offset, etc.

The QoS specification may consist of a traffic profile and performance requirements. The traffic profile declares the amount of storage capacity to be reserved, the time and duration of the reservation, and the distribution of data accesses, if known. The performance requirements can be expressed along one or more of the following (sometimes overlapping) dimensions:

- data access latency (average, variation, or maximum),
- data access jitter,
- data availability/redundancy,
- coverage area,
- number of simultaneous accesses
- bandwidth savings, and

- cost.

There are a series of corresponding actions happening at the client, the metadata server and the storage servers. First, the client accesses the metadata server to get necessary information including data locations, data encoding algorithm and parameters, storage capacities of the servers, static and dynamic performance of the servers, etc. Necessary file locking is applied by the metadata server. Once the client gets the information, it plans an access schedule based on these information and the application QoS requirements. The access schedule includes selection of disks to access, coding algorithms (decided for write accesses), and data layout. The client then connects to the selected disks and requests for access admissions from the admission controllers of the disks.

Appendix C. Access Control in RobuSTore

Since RobuSTore aggregates distributed resources which are shared with many distributed users, it is important to have distributed security mechanisms to allow secure and safe remote accesses. Although security is not the focus of the dissertation, we briefly discuss possible access control mechanisms to use in RobuSTore.

In a distributed multi-domain environment, centralized access control is not feasible. First, centralized mechanisms have limited scalability and can easily become performance bottlenecks when the storage system scales. Further, it is prohibitively hard to manage a centralized namespace when there are multiple administration domains. The administration domains can add new users and new files separately and assign access rights dynamically, so the central access controller has to authenticate each administrator. Finally, the trust model of centralized mechanism is not practical. Administrators usually do not want to build their own system based on the trust of a third party.

Existing methods include ACL-based methods and capability-based methods.

PKI-based methods use ACL mechanisms with PKI-based user authentication. Each user has both a public key and a private key, and also a certificate signed by a third party. The third party is often called the certificate authority, or CA. The certificate's purpose is to prove a user's identity, which includes the information of its identity name, its public key, and the CA's name, all encrypted using the CA's private key. If two users have certificates signed by same CA, they can authenticate each other; after communication is connected, user A can give user B its certificate, then

user B can use the CA's public key to decrypt the certificate to get user A's identity and public key. Once B has checked out A's certificate, B must make sure that the certificate is really from A by generating a random message and sending it to A. A then encrypts the message using its private key and sends it back to B. B decrypts the message using A's public key. If the decrypted message is the original random message, then B knows that A is who it says it is. Using the same method, A can authenticate B. After the mutual authentication, they can decide on another party's access rights based on their ACLs.

The credential chain method uses a capability mechanism. In this method, each user also has both a public key and a private key. When the owner of data wants to give access capability of the file to another user, it generates a credential for that user by encrypting the description of the access capability, the owner's public key, and the user's public key. When the client wants to access the data, it sends the credential to the owner. The credential already includes everything needed for the data owner to authenticate the client and to check the client's access capability.

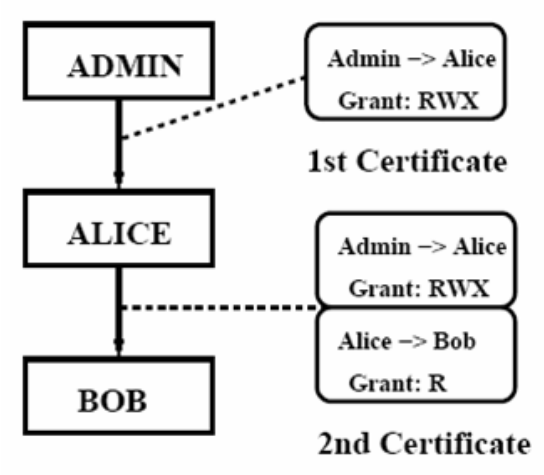


Figure C-1. Two-level credential chain

A level-one credential for Alice could be like this:

Authorizer: "<Administrator's Public Key>"

Licensees: "<Alice's Public Key>"

Conditions: (app_domain == "RobuSTore") && (HANDLE == "666240") -> "RWX";

Comment: "robustore_dir"

signature: "<Signature by Administrator>"

Alice can use her credential to generate a level-two credential for Bob:

Authorizer: "<Administrator's Public Key>" Licensees: "<Alice's Public Key>" Conditions: (app_domain == "RobuSTore") && (HANDLE == "666240") -> 'RWX"; Comment: "robustore_dir" signature: "<Signature by Administrator>"
Authorizer: "<Alice's Public Key>" Licensees: "<Bob' Public Key>" Conditions: (app_domain == "RobuSTore") && (HANDLE == "666240") && (localtime >= "20021106000001") && (localtime <= "20021106235959") -> "RWX"; Comment: "robustore_dir" signature: "<Signature by Alice>"

The credential chain method is more flexible than GSI. It requires no third-party trust, and uses fewer communication rounds during authentication.

References

[1] Biomedical Informatics Research Network (BIRN), http://www.nbirn.net.

[2] GriPhyN: Grid Physics Network, http://www.griphyn.org.

[3] The EarthScope Project, http://www.earthscope.org.

[4] AstroGrid, http://www.astrogrid.org.

[5] iGrid2005, http://www.igrid2005.org, San Diego, CA, Sep 26-30, 2005.

[6] L. L. Smarr, A. A. Chien, T. DeFanti, J. Leigh, and P. M. Papadopoulos, "The OptIPuter," *Communications of the ACM*, vol. 46, pp. 58-67, 2003.

[7] National LambdaRail, http://www.nlr.net.

[8] NetherLight, http://www.netherlight.net.

[9] The Global Lambda Interchange Facility (GLIF), http://www.glif.is/.

[10] C. Walter, "Kryder's Law," *Scientific American*, 2005.

[11] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System," USENIX Summer Technical Conference, Portland, Oregon, 1985.

[12] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, vol. 23, 1990.

[13] P. Leach and D. Perry, "CIFS: A Common Internet File System," *Microsoft Interactive Developer*, 1996.

[14] J. K. Ousterhout, A. R. Cerenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 1988.

[15] G. J. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Theil, "LOCUS: A Network Transparent, High Reliability Distributed System," the Eighth ACM Symposium on Operating Systems Principles, published in Operating Systems Review 15, Pacific Grove, CA, USA, 1981.

[16] B. Lyon and R. Sandberg, "Breaking through the NFS Performance Barrier," Legato Systems, Inc., Technical report.

[17] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," USENIX Winter 1994 Technical Conference, San Fransisco, CA, 1994.

[18] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," USENIX 1996 Technical Conference, San Diego, CA, 1996.

[19] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, "Taming Aggressive Replication in the Pangaea Wide-Area File System," OSDI '02, 2002.

[20] J. H. Hartman and J. K. Ousterhout, "The Zebra striped network file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 13, pp. 274 - 310, 1995.

[21] D. L. Black and S. Fridella, "pNFS Block/Volume Layout," draft-ietf-nfsv4-pnfs-block-01.txt, Aug 30, 2006.

[22] B. Halevy, B. Welch, J. Zelenka, and T. Pisek, "Object-based pNFS Operations," draft-ietf-nfsv4-pnfs-obj-02.txt, Aug 29, 2006.

[23] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS Version 4 Protocol," 2nd International System Administration and Networking Conference (SANE2000), 2000.

[24] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," International Conference on Management of Data (SIGMOD), 1988.

[25] P. F. Corbett and D. G. Feitelson, "The Vesta parallel file system," *ACM Transactions on Computer Systems*, vol. 14, pp. 225--264, 1996.

[26] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek, "Parallel file systems for the IBM SP computers," *IBM Systems Journal*, vol. 34, pp. 222-248, 1995.

[27] I. S. S. Division, "Paragon System User's Guide," Order Number 312489-004, May, 1995.

[28] N. Nieuwejaar and D. Kotz, "The Galley Parallel File System," *Parallel Computing*, vol. 23, pp. 447-476, 1997.

[29] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters," 4th Annual Linux Showcase and Conference, Atlanta, GA, 2000.

[30] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," USENIX Conference on File and Storage Technologies (FAST), Monterey, CA, 2002.

[31] C. F. System, "Lustre: A Scalable, High-Performance File System," Lustre File System v1.0 Architecture White Paper from clusterfs.org, 2002.

[32] The Panasas File System, http://www.panasas.com.

[33] Kazaa, http://www.kazaa.com.

[34] BitTorrent, http://www.bittorrent.com.

[35] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal*, vol. 29, pp. 147-160, 1950.

[36] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300--304, 1960.

[37] J. S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems," *Software -- Practice & Experience*, vol. 27, pp. 995-1012., 1997.

[38] A. J. Viterbi, "Error bounds for convolutional codes and an asymtotically optimum decoding algorithm," *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 260--269, 1967.

[39] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," ACM Computer Communication Review, 1997.

[40] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical Loss-Resilient Codes," 29th ACM Symposium on Theory of Computing (STOC '97), New York, 1997.

[41] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, "Efficient erasure correcting codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 569-584, 2001.

[42] P. Maymounkov, "Online Codes," Tech Report of NYU, TR2002-833, November 5, 2002.

[43] J. Kubiatowicz, D. Bindel, and e. al., "OceanStore: An Architecture for Global-Scale Persistent Storage," the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, 2000.

[44] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, and B. Z. Kubiatowicz, "Pond: The OceanStore Prototype," FAST 2003, 2003.

[45] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker, "Total recall: System support for automated availability management," the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, 2004.

[46] R. G. Gallager, *Low Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[47] D. MacKey and R. Neal, "Good codes based on very sparse matrices," 5th IMA Conference on Cryptography and Coding, Cirencester, UK, 1995.

[48] M. Luby, "LT Codes," Proc. IEEE Symp. On Foundations of Computer Science 2002, 2002.

[49] A. Shokrollahi, "Raptor codes," Digital Fountain and EPFL, 2003.

[50] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," the First International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, 2002.

[51] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System," 16th ACM Symposium on Operating Systems Principles, Saint Malo, France, 1997.

[52] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," Symposium on Operating Systems Principles, Brighton, UK, 2005.

[53] M. Isard, M. Manasse, and C. Thekkath, Koh-I-Noor Project, http://research.microsoft.com/research/sv/kohinoor/.

[54] H. Weatherspoon, M. Delco, and S. Zhuang, "Typhoon: An Archival System for Tolerating High Degrees of File Server Failure," University of California, Berkeley.

[55] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using Erasure Codes Efficiently for Storage in a Distributed System," DSN 2005: The International Conference on Dependable Systems and Networks, Yokohama, Japan, 2005.

[56] R. L. Collins and J. S. Plank, "Assessing the performance of Erasure Codes in the Wide Area," DSN-2005: The International Conference on Dependable Systems and Networks, Yokohama, Japan, 2005.

[57] M. Luby, "Practical Loss-Resilient Codes," ITW 1998, San Diego, CA, 1998.

[58] Adaptec, "Advanced SCSI Programming Interface," November 6, 2001.

[59] S. Herzog, "Signaled Preemption Priority Policy Element," IETF/RFC2751, January 2000.

[60] M. Seaman, A. Smith, E. Crawley, and J. Wroclawski, "Integrated Service Mappings on IEEE 802 Networks," IETF/RFC2815, May 2000.

[61] A. Ghanwani, W. Pace, V. Srinivasan, A. Smith, and M. Seaman, "A Framework for Integrated Services," IETF/RFC2816, May 2000.

[62] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *IEEE Computer*, vol. 27, pp. 17-28, 1994.

[63] D. Kotz, S. B. Toh, and S. Radhakrishnan, "A Detailed Simulation Model of the HP 97560 Disk Drive," TR94-220.

[64] N. Nieuwejaar and D. Kotz, "Performance of the Galley Parallel File System," Fourth Annual Workshop on I/O in Parallel and Distributed Systems, 1996.

[65] J. S. Bucy and G. R. Ganger, "The DiskSim Simulation Environment Version 3.0 Reference Manual," Carnegie Mellon University CMU-CS-03-102, January 2003.

[66] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang, "Modeling hard-disk power consumption," 2nd USENIX Conference on File and Storage Technologies (FAST '03), 2003.

[67] J. L. Griffin, J. Schindler, S. W. Schlosser, J. C. Bucy, and G. R. Ganger, "Timing-accurate storage emulation," Conference on File and Storage Technologies (FAST'02), Monterey, CA, 2002.

[68] A. Riska, E. Riedel, and S. Iren, "Managing Overload Via Adaptive Scheduling," 1st Workshop on Algorithms and Architecture for Self-Managing Systems, San Diego, CA, 2003.