Symphony: An Integrated Multimedia File System

by

Prashant Jagdish Shenoy, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1998

Symphony: An Integrated Multimedia File System

Approved by
Dissertation Committee:

To Mom and Dad

Acknowledgments

The long-winding road towards a doctoral degree can be fraught with frustration and pain. Instead, the company and support that I received during my five year stint in Austin have made this journey rewarding and downright fun. I will try and acknowledge the people who have shared these memorable years. My apologies to those I may forget.

I would first like to thank my dissertation supervisor, Prof. Harrick Vin, for his constant encouragement, mentoring and guidance. His perseverance and strong emphasis on excellence—qualities that I pledge to emulate—have been inspiring.

I am grateful to Prof. Lorenzo Alvisi, Prof. Mike Dahlin, Prof. Don Fussell, Prof. Kevin Jeffay, and Dr. Dan Swinehart for agreeing to serve on my dissertation committee. Their constructive comments and suggestions have contributed greatly to my dissertation research. I would especially like to thank Prof. Don Fussell for his help and advice during the final stages of my dissertation and my interview trail. My sincere thanks to Patti Spencer, our Facilities and Equipment Coordinator, for providing hardware and software support at a moments notice; and to Gloria Ramirez, our Graduate Coordinator, for helping me through the bureaucratic maze at UT.

Several exceptional collaborators have helped shape the ideas in this dissertation. The kernel support for Symphony was jointly developed with Balaji Balasubramanayan. Portions of the work on failure recovery techniques and the initial implementation of the Symphony prototype was joint with Sriram Rao. The design and implementation of the buffer subsystem of Symphony was jointly done with Renu Tewari. Pawan Goyal has contributed, both directly and indirectly, to many of the ideas contained in this dissertation. I thank him for his critical comments and insightful suggestions.

Life in Austin has been enriched by several people I have gotten to know over the past few years. I have immensely benefited from many stimulating discussions, technical and otherwise, with my colleagues at the Distributed Multimedia Computing Laboratory (DMCL): Balaji Balasubramanyan, Xingang Guo, Amir Husain, Jon Kay, Scott Page, Ed Posnak, Sriram Rao, Jasleen Sahni and T R. Vishwanath. I will fondly remember the endless discussions and arguments during the lunch hour and the 4 p.m. coffee break. The company of Sundeep Abraham, Rajesh Govindan, Srinivas Guggilla, Raja Krishnaswamy, Anand Kudari, Dev Putchala, Rajmohan Rajaraman, Divas Sanwal and Premal Shah provided many memorable moments. I owe special thanks to Bhagat Nainani for his help and support during my early years in Austin.

I am especially indebted to two fabulous colleagues and close friends, Pawan Goyal and Renu Tewari. I thank them for patiently listening to my numerous grouses, their help on innumerable occasions, their advice on technical matters and their perspectives on life in general—I have learnt a lot from them.

Lastly, but most importantly, I would like to thank my family for sharing the ups and downs of graduate student life and for putting up with long years of separation. My sister, Preeti, has been a endless source of enthusiasm through these years. My parents, Shakuntala and Jagdish Shenoy, have provided inspiration and guidance throughout the

course of my graduate studies and my entire life. I am what they taught me to be. I dedicate this dissertation to them.

PRASHANT JAGDISH SHENOY

The University of Texas at Austin
August 1998

Symphony: An Integrated Multimedia File System

Publication No. ______________________

Prashant Jagdish Shenoy, Ph.D.
The University of Texas at Austin, 1998

Supervisor: Harrick M. Vin

Advances in computing technologies coupled with the growing popularity of the World Wide Web have led to a proliferation of applications that have diverse performance requirements and access heterogeneous data. Existing general purpose file systems have been developed for a single class of applications. Since these file systems treat all applications alike regardless of their requirements, they are ineffective at simultaneously supporting multiple application classes. A simple approach that addresses this limitation is to design a file system that employs multiple servers, each optimized for a particular class of applications. However, we demonstrate that static partitioning of resources among component servers inherent in this approach results in a manifold degradation in performance over an approach that employs an integrated server for all application classes. The design of such an integrated multimedia file system poses three challenges: (i) the file system must support multiple application classes and align the service provided within a class with application needs, (ii) it must enable the coexistence of multiple data type specific policies, and (iii) it must employ an extensible architecture to facilitate easy integration of new application classes and data types. The design, implementation and evaluation of Symphony, an integrated multimedia file system that meets these requirements is the focus of this dissertation.

In this dissertation, we first design mechanisms for disk scheduling, placement, caching and failure recovery that implement core file system functionality. Whereas the mechanism for disk scheduling supports multiple application classes and aligns the service provided within a class with application needs, those for placement, caching and failure recovery enable the coexistence of multiple data type specific policies. Moreover, these mechanisms prevent interference between policies with conflicting requirements and dynamically allocate resources to applications on demand. Next we develop a number of data type specific policies for placement, failure recovery and meta data management that exploit the characteristics of the data to optimize file server performance. We then develop a novel two layer architecture for Symphony, consisting of a data type independent layer and a data type specific layer. The two layers of Symphony separate data type independent mechanisms from data type specific policies, and thereby facilitate easy extensions to the file system.

We implement the policies and mechanisms that we develop in the two layer architecture of Symphony and then experimentally evaluate the Symphony prototype to demonstrate its effectiveness in supporting diverse applications

and managing heterogeneous data. Our results show that, even at moderate utilization levels, a four disk Symphony prototype can yield a factor of 1.9 improvement in response times of interactive text requests over existing disk scheduling techniques, while meeting the deadlines of all real-time continuous media requests. Moreover, tailoring policies to needs of data types improves server throughput and reduces the recovery overhead for continuous media from a factor of two to zero.

Contents

Chapter 1

Introduction

multi: many, multiple, much.
medium: pl media: an individual held to be a channel of communication between the earthly world and a world of spirits.

—*Mariam Webster's Collegiate Dictionary*

Operating systems form an essential software substrate of computing systems. An operating system manages physical resources such as processors, memory, and storage devices and presents a virtualized view of these resources to applications. The virtualized view provided for storage devices is that of a *file system*. A file system provides persistent storage of data by implementing *files*—named objects containing data that are immune to temporary failures of the operating system. From the user's perspective, the file system is one of the most ubiquitous components of an operating system. Consequently, the services provided by a file system have a critical impact on the usability of the system and ease of application design. Since the file system manages non-volatile storage devices such as disks that are orders of magnitude slower than volatile storage devices such as memory and registers, the policies and mechanisms employed by the file system significantly impact the overall system performance. Hence, file systems must be designed to maximize utility to applications as well as optimize system performance.

Traditionally, general purpose file systems have focused on supporting a single class of best-effort applications and techniques that optimize performance for such applications. The explosive growth of the World Wide Web coupled with synergistic advances in computing and communication technologies have led to a proliferation of a new generation of applications with significantly differing requirements. Hence, file systems will increasingly need to support applications with diverse performance requirements. Consider the following examples of emerging applications that must be supported by general purpose file systems:

- *I/O intensive applications:* Applications in this class arise in a variety of disciplines such as Biology (e.g., simulation of viral structures), Chemistry (e.g., molecular dynamics), Earth Sciences (e.g., seismic data processing), Engineering (e.g., study of aircraft turbulence), Graphics (e.g., parallel rendering systems), and Space Sciences (e.g., visualization of satellite images). All of these applications generate, access and process massive amounts of data; the volume of data managed is orders of magnitude larger than that managed by conventional applications. Moreover, unlike conventional applications that access only textual data, these applications access different types of data such as audio, video, images, animations sequences, and text (collectively referred

Table 1.1: Characteristics of data types

Data type	Storage requirement	Real-time requirements
Text	4-8KB	No
Gif image	64 KB	No
CD audio	175 MB (384 Kb/s)	Yes
MPEG video	2 GB (4 Mb/s)	Yes

to as *multimedia*). These data types inherently differ in their characteristics such as size, format, data rate, and real-time requirements. To illustrate, typical textual and image files are a few kilobytes in size, whereas audio and video files are orders of magnitude larger. Moreover, textual data accesses are aperiodic in nature, whereas audio and video (continuous media) accesses are periodic and sequential. Finally, textual and image accesses are best-effort in nature, while continuous media accesses impose real-time constraints and require performance guarantees. Table 1.1 summarizes these characteristics.

- *Interactive applications:* Examples of interactive applications include flight training simulators, virtual worlds, and multi-player games. In addition to the requirements imposed by I/O intensive applications, these applications demand real-time interactivity from the system.

The need for supporting diverse applications and managing heterogeneous data necessitates fundamental changes to the services provided by a file system. The design, implementation, and evaluation of a file system that achieves these objectives, referred to as an *integrated multimedia file system*, is the focus of this dissertation.

The rest of this chapter is organized as follows. A brief survey of existing file systems is presented in Section 1.1. In Section 1.2, we argue that these file systems are inadequate for meeting the requirements of emerging applications. Section 1.3 summarizes the research contributions of this dissertation, and Section 1.4 presents an outline of the dissertation.

1.1 File Systems: A Perspective

File systems have evolved significantly since their inception. Historically the first file systems were developed for mainframe operating systems, such as CTSS, OS/360, MVS, VMS and Multics, and were optimized for supporting data processing applications such as payroll and inventory control [79]. The advent of the UNIX[1] operating system in the 1970s led to a proliferation of affordable minicomputers. Early versions of UNIX supported a file system—now known as the System V file system—that employed simple storage and retrieval policies, such as small block sizes and limited prefetching [73]. These policies limited the file server throughput to less than 4% of the peak disk bandwidth [56]. The Berkeley Fast File System (FFS), introduced in 4.2BSD UNIX, addressed these limitations by increasing the block size used to store files and by using sophisticated storage allocation and read-ahead techniques [56]. It also introduced new features such as file locking, long file names and symbolic links.

[1] UNIX is a registered trademark of the X/OPEN consortium.

The growing popularity of local area networks (LANs) in the eighties saw the emergence of distributed file systems, such as Sun Microsystem's Network File System (NFS) and AT&T's Remote File Sharing (RFS) [87]. These file systems allowed users to access files stored on remote servers from any computer on a network. Both NFS and RFS were designed for LAN environments with a small number of clients; neither scales well to enterprise-wide and campus-wide networks with thousands of clients. The Andrew file system (AFS) from CMU and IBM was developed to address this limitation [87]. AFS partitioned the network into clusters and used a dedicated server per cluster to store files frequently accessed by clients in the cluster. AFS also used aggressive caching techniques to minimize network traffic. Together, these techniques enabled AFS to scale to a large number of servers servicing thousands of clients.

Several technological trends have driven file system research in the nineties. Falling memory prices have made large file caches on file servers commonplace. These caches service a majority of the read requests using cached data, causing disk traffic to be dominated by write requests. The log-structured file system from Berkeley exploits this trend by writing all information to disk as a sequential append-only log, thereby substantially improving disk throughput [74]. xFS extends this idea to a distributed file system. In addition, xFS exploits the availability of fast local area networks to harness resources available on client machines to improve file I/O performance. All aspects of file system services—data storage, data caching and control processing—are distributed across potentially all machines in the system, leading to a serverless architecture [3].

Concurrently, advances in compression technologies, coupled with several standardization efforts such as MPEG, have led to a proliferation of digitized audio and video. Since audio and video inherently differ in their characteristics as compared to textual data, conventional file systems are inadequate for managing these data types. To address these limitations, several special-purpose file systems, referred to as continuous media servers, have been developed [2, 29, 44, 71, 84, 88]. These servers exploit the sequential and periodic nature of continuous media accesses to improve server throughput, and employ resource reservation algorithms to provide real-time performance guarantees.

Recently, the growing popularity of the World Wide Web has led to an increased interest in mechanisms for wide area information access. File systems such as WebNFS [12] and Common Internet File System (CIFS) [49] that enable global information access over wide area networks (WANs) have been developed to fill this void. These file systems are optimized for file accesses over networks with large latencies.

In summary, rapid technological advances have fueled file system research in the past few decades. File systems have evolved from simple data storage systems to sophisticated networked information access systems. However, as we shall argue in the next section, even these modern file systems have certain fundamental shortcomings that make them unsuitable for supporting diverse applications accessing heterogeneous data.

1.2 Limitations of Existing File Systems

The need for managing heterogeneity in application requirements and data characteristics imposes three key requirements on a file system:

- *Heterogeneity in application requirements:* Applications that coexist in general purpose computing environments have diverse performance requirements. For instance, interactive best-effort applications, such as word-processors and compilers, desire low average response times but no absolute performance guarantees. Throughput-intensive best-effort applications, such as ftp and http servers, desire high aggregate throughput and are less concerned about the response times of individual requests. Soft real-time applications, such as video players, require performance guarantees from the file system (e.g., bounds on response times) but can tolerate occasional violations of these guarantees. To efficiently support diverse applications, a file system will be required to simultaneously optimize different performance criteria. To do so, the file system should support multiple classes of service (e.g., real-time, interactive) and align the service provided within a class with application needs.

- *Heterogeneity in data characteristics:* Emerging applications access multiple types of data. Since data types inherently differ in their characteristics, use of a single policy for managing all data types can yield sub-optimal results. For instance, the least recently used (LRU) cache replacement policy is suitable for textual file accesses that exhibit locality of reference, but is completely ineffective for sequential continuous media accesses. Similarly, the Interval Caching policy developed for sequential continuous media accesses is unsuitable for textual accesses [26]. Since no single cache replacement policy is suitable for all data types, the file system must either sacrifice cache hit ratio or support multiple policies to reconcile conflicting requirements. Similar arguments are applicable to policies employed for storage, retrieval, and failure recovery. Hence, to efficiently support heterogeneous data, the file system should enable the coexistence of multiple data type specific specific policies.

- *Extensibility:* Since it is difficult, if not impossible, to foresee requirements imposed by future applications and data types, a file system should employ techniques that facilitate easy integration of new application classes and data types. This requires file systems to employ an extensible architecture so that file system services can be easily tailored to meet new requirements.

Unfortunately, existing file systems fail to meet one or more of these requirements. Most existing file systems are optimized for a single application class and a single data type. For instance, conventional file systems are optimized for best-effort applications that store and retrieve textual data [56]. These file systems provide a best-effort service to all applications regardless of their performance requirements. Consequently, they are ineffective at meeting the performance requirements of real-time applications. In contrast, continuous media file servers employ predictable resource allocation techniques to meet the real-time requirements of audio and video. Most of these servers assume a predominantly read-only environment with substantially less stringent response time requirements (e.g., video-on-demand services in which latencies of a few seconds for initiating continuous media playback are considered acceptable). Hence, they are not suitable for conventional textual applications. Finally, since all of these file systems have been developed for a single class of applications, none of them employ an extensible architecture. Due to these limitations of existing file systems, realizing emerging applications requires the development of integrated multimedia file systems (henceforth referred to as integrated file systems).

1.3 Summary of Contributions

This dissertation focuses on the design, implementation and evaluation of Symphony—an integrated file system that can manage heterogeneity in application requirements and data characteristics. A key thesis of our work is that a single technique is inadequate for meeting the diverse requirements of applications and data types, and hence, an integrated file system should enable the *coexistence of multiple application-specific and data-type specific techniques*. This is a fundamental departure from existing file systems, which employ a single technique for servicing all applications and data types regardless of their requirements.

This dissertation makes five contributions. First, we propose two different methodologies for designing integrated file systems and evaluate their tradeoffs. Second, we design mechanisms that enable the coexistence of diverse data type specific policies in the file system. Third, we design data type specific policies that exploit the characteristics of the data to optimize file server performance. Fourth, we develop a novel two layer architecture for Symphony that separates data type independent mechanisms from data type specific policies, and thereby facilitates easy extensions to the file system. Fifth, we instantiate our policies and mechanisms in a prototype implementation of Symphony and experimentally demonstrate the efficacy of our techniques for managing diverse applications and heterogeneous data. In what follows, we describe our contributions in detail.

We first propose two different methodologies for designing integrated file systems—logically integrated file systems and physically integrated file systems—and evaluate their tradeoffs. We argue that use of a single physically integrated server for all applications is desirable in many environments over logically integrated file systems that employ separate servers for each application class. For such environments, we demonstrate that dynamic sharing of resources inherent in the former approach yields a manifold performance improvement over employing separate servers. Based on these results, we choose the physically integrated file system architecture for designing Symphony. We then examine the requirements imposed on physically integrated file systems and derive key principles underlying the design of such file systems. Specifically, we argue that, a physically integrated file system must: (i) export multiple classes of service to applications and align the service provided with application needs, (ii) support multiple data type specific policies for placement, caching, failure recovery, etc., and employ mechanisms that enable their coexistence, and (iii) employ an extensible architecture.

To meet these requirements, we first develop mechanisms for dynamically allocating file system resources. The key challenges in designing such mechanisms are: (i) they must be powerful enough to enable the coexistence of diverse policies, and (ii) they must prevent interference between policies with conflicting requirements, while providing all the benefits of dynamic resource allocation. We develop mechanisms for disk scheduling, placement, caching, failure recovery, and meta data management that achieve these objectives. The most noteworthy of these mechanisms is the Cello disk scheduling framework that we develop for servicing disk requests with diverse performance requirements. Cello achieves this objective by employing a novel two level disk scheduling architecture that allocates disk bandwidth at two time scales. We demonstrate that such a two level disk scheduling framework is suitable for integrated file systems since: (i) it aligns the service provided with the application requirements, (ii) it protects application classes from one another, (iii) it is work-conserving and can adapt to changes in work-load, (iv) it minimizes the seek time and rotational latency overhead incurred during access, and (v) it is computationally efficient. Our experimental evaluation of Cello shows that, at a disk utilization of 60%, Cello yields a factor of 2.5 improvement in response time over a conventional disk scheduling algorithm such as SCAN when scheduling a mixture of text and video clients.

Next, we develop several data type specific policies for placement, failure recovery and meta data management. For placement, we consider an integrated file system that employs disk arrays for storing continuous media files and develop techniques for determining the optimal stripe unit size and degree of striping for such arrays. A key insight behind our techniques is that a stripe unit size and degree of striping that minimizes the tail of the response time distribution yield optimal throughput for real-time continuous media workloads (unlike best effort text workloads for which minimizing the average response time yields optimal performance). We develop analytical models that use the server configuration and workload characteristics to predict the optimal stripe unit size and degree of striping. These models are the first in the literature to accurately characterize the performance of the disk arrays storing variable bit rate continuous media. Our results show that the conventional wisdom of using a large stripe unit size and striping each file across all disks in the array does not necessarily yield good file server performance. Instead, such an approach increases load imbalance across disks and adversely affects real-time performance guarantees provided to continuous media clients, thereby reducing server throughput.

For disk failure recovery, we develop two novel techniques that utilize the characteristics of continuous media files for efficient recovery. Whereas the first technique exploits the sequentiality of continuous media playback to reduce the recovery overhead in conventional disk arrays, the second technique exploits the inherent redundancy in video files (rather than error correcting codes) to *approximately* reconstruct data stored on failed disks. We show that the former technique yields a factor of $G - 1$ reduction in recovery overhead over conventional techniques, where G is the parity group size. The latter technique decouples the task of *online reconstruction* of requested data from that of *rebuild* of failed disks—a fundamental departure from conventional recovery techniques, which employ a single mechanism, such as parity, for both tasks. Furthermore, the technique enhances the scalability of integrated file systems by: (1) integrating online reconstruction with the decompression of video files at client sites, and thereby reducing the recovery overhead at the server to zero; and (2) supporting graceful degradation in the quality of recovered images with increase in the number of disk failures.

We then develop a novel two layer architecture for Symphony, consisting of a data type independent layer and a data type specific layer. Our two layer architecture cleanly separates data type independent mechanisms from data type specific policies, and thereby facilitates easy extensions to the file system. We instantiate the policies and mechanisms that we develop in a prototype implementation of such a two layer architecture. The data type independent layer of our prototype consists of a number of novel features including: (i) the Cello disk scheduling framework that supports multiple classes of service, (ii) a storage manager that supports multiple placement policies, (iii) a fault-tolerance layer that enables data type specific failure recovery, (iv) a meta data manager that enables data type specific structure to be assigned to files while supporting the traditional byte stream interface, (v) a buffer manager that supports multiple caching policies, and (vi) a resource manager that reserves resources to provide performance guarantees to real-time applications. The data type specific layer contains modules for text, video and audio that use these mechanisms to implement data type specific policies. The video module, for instance, implements policies for placement, retrieval, failure recovery, meta data management, and caching that are tailored for multi-resolution video files and supports both server-push and client-pull modes for accessing files.

Our experimental evaluation of the Symphony prototype demonstrates its effectiveness in supporting diverse application classes and managing heterogeneous data. Our results show that: (i) even at moderate utilization levels, a four disk Symphony prototype yields a factor of 1.9 improvement in response time of text requests over conventional disk scheduling techniques, while meeting the deadlines of all real-time video requests, (ii) tailoring the placement policy to application needs enables Symphony to optimize server throughput and (iii) supporting data type specific failure recovery policies enables Symphony to reduce the recovery overhead from a factor of two to zero for continuous media applications.

1.4 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 examines the merits and limitations of two different methodologies for designing integrated file systems. We argue qualitatively and quantitatively that dynamic resource allocation employed by physically integrated file systems yields better resource utilization than static resource partitioning inherent in the logically integrated file systems.

Chapter 3 articulates design considerations for physically integrated file systems. We first examine the requirements imposed by applications and data types on a physically integrated file system and then propose a two layer architecture for Symphony that meets these requirements. The two layers of the architecture implement data type independent mechanisms and data type specific policies, respectively. We provide an overview of the mechanisms and policies for placement, retrieval, caching, failure recovery and meta data management that we have developed for such a two layer architecture. The next three chapters describe some of these mechanisms and policies in detail.

Chapter 4 presents the Cello disk scheduling framework that we have developed for servicing disk requests with different performance requirements. We describe the two-level disk scheduling architecture employed by Cello, consisting of a class-independent scheduler and a set of class specific schedulers. We describe class-specific schedulers for a variety of application classes and present experimental results to demonstrate that Cello is a suitable disk scheduling framework for integrated file systems.

Chapter 5 discusses optimal placement policies for striped continuous media files. We consider an integrated file system employing disk arrays and develop analytical models to determine the optimal stripe unit size and degree of striping for such arrays. We validate our models through simulations and then use them to (1) study the effect of various system parameters (such as the number of clients, number of disks, etc.) on the stripe unit size, and (2) derive procedures for determining optimal placement policies.

Chapter 6 presents failure recovery policies for continuous media files. We present two techniques that exploit the characteristics of continuous media data for efficient failure recovery. We present analytical and experimental results to demonstrate the efficacy of these techniques in reducing the failure recovery overhead.

Chapter 7 discusses the implementation and evaluation of Symphony. We first describe how the mechanisms and policies that we have developed are implemented in the two layer architecture of Symphony. We then present results of our experimental evaluation of Symphony and demonstrate that Symphony is suitable for managing diverse applications and heterogeneous data.

Finally, Chapter 8 summarizes our results and outlines directions for future research.

Chapter 2

Design Methodologies

United we stand, divided we fall.

—Aesop, The Four Oxen and the Lion

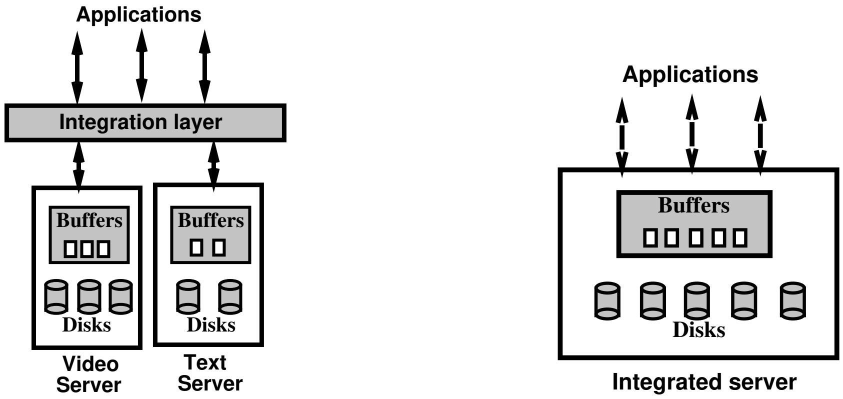
Integrated file systems must support diverse applications accessing heterogeneous data. There are two methodologies for designing such file systems. *Logically integrated* file systems consist of multiple component file servers, each optimized for a particular application class, glued together by an integration layer that provides an uniform way of accessing file stored on different file servers. *Physically integrated* file systems consist of a single server that supports all application classes. Figure 2.1 illustrates these architectures.[1] Both methodologies have their advantages and disadvantages.

- *Resource utilization issues:* Logically integrated file systems statically partition server resources among component file servers, which may result in under-utilization of resources. Physically integrated file systems do not suffer from these limitations since they employ a single server that shares all resources among all application classes. The resulting statistical multiplexing gains yield higher utilization of resources and better performance.

- *Development costs:* Since file servers optimized for a single application class are simple to develop, logically integrated file systems are easy to design. Dynamic resource sharing in physically integrated file systems requires the development of sophisticated multiplexing mechanisms, which can increase file system complexity. However, physically integrated file systems have a software engineering advantage over their counterpart. Adding support for a new application class in a logically integrated file system requires the development of a new component server. In contrast, physically integrated file systems employ mechanisms that enable the coexistence of multiple policies, which allows new application classes to be easily supported. This also amortizes development costs over the set of supported classes

- *Maintenance costs:* The system administration costs of maintaining separate servers in logically integrated file systems is often higher than that of maintaining a single physically integrated server.

[1] A third approach is to develop applications that access data from multiple file systems. Although such an approach complicates application design, it may be desirable in large enterprises that employ a number of disparate systems. We do not consider this approach in this dissertation.



(a) Logically integrated file system (b) Physically integrated file system

Figure 2.1: Logically and physically integrated file systems. The logically integrated file system employs a separate server for each application class and partitions resources among these servers; the physically integrated file server employs a single server that shares all resources among all application classes.

Due to these considerations, the two methodologies are suited for different environments. Logically integrated file systems are desirable in specialized environments, such as large video-on-demand systems, or in environments where the workload is known a priori and does not change over time (which allows static partitioning of resources to be effective). Physically integrated file systems are desirable in environments with heterogeneous or dynamically fluctuating workloads. They are also suitable for small office environments where the cost of maintaining separate servers can be prohibitive.

The objective of this chapter is to compare the performance of the two methodologies in environments with heterogeneous workloads. Specifically, we examine if dynamic sharing of resources in physically integrated file systems yields substantial performance gains over static resource partitioning of resources employed by logically integrated file systems. We also examine if over-provisioning logically integrated file systems so as to provide performance comparable to physically integrated file systems is a cost-effective approach to designing integrated file systems. We then use these results to determine a suitable methodology for designing Symphony.

The rest of this chapter is organized as follows. Section 2.1 compares the two architectures qualitatively, while Section 2.2 presents a quantitative comparison. Finally, Section 2.3 summarizes our results.

2.1 Qualitative Comparison

Logically integrated file systems statically partition server resources (disks, buffers, etc) among component file servers (see Figure 2.1(a)). Such partitioning enables a logically integrated file system to prevent interference between component servers. Static partitioning of disks, for instance, causes requests accessing different servers to access mutually exclusive sets of disks, thereby isolating requests with different performance requirements from each other. However, static partitioning of resources has the following limitations:

- Static partitioning of resources among component servers is governed by the expected workload on each server. If the observed workload varies dynamically or deviates significantly from the expected, then resources allocated to some servers may be under-utilized, while other servers are saturated. In such a scenario,

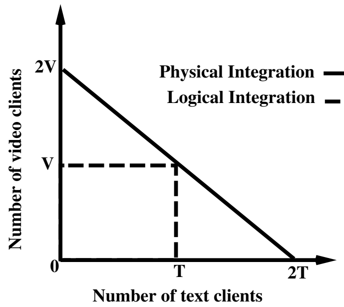


Figure 2.2: Boundaries of operation for logically integrated and physically integrated file systems. A logically integrated file system that equally partitions resources among a text and a video server can support up to T text and V video clients. A physically integrated server with an identical configuration can support up to twice as many clients from each class if the other class has no requests.

repartitioning of resources may be necessary; such repartitioning is tedious and may require the system to be taken off-line. An alternative to repartitioning is to add more resources to the server, which causes resources in underutilized partitions to be wasted.

- Resource partitioning for application classes that require different fractions of different resources can lead to significant underutilization of resources. To illustrate, an application class may need a certain fraction of the total storage space, but a different fraction of the aggregate disk bandwidth. In such a scenario, allocation of disks to the corresponding component server will be governed by the maximum of the two values. This can lead to under-utilization of either storage space or disk bandwidth allocated to the server.

In contrast, physically integrated file systems share all resources among all application classes, obviating the need for static partitioning of resources. In such servers, resources are allocated to applications on demand, thereby enabling such file systems to adapt to dynamically changing or skewed workloads (e.g., colocating very large cold files and small hot files improves resource utilization). Since all resources are shared among all applications, more resources are available to service each user request, which improves performance. Sharing resources also enables physically integrated file systems to handle a larger number of clients from an application class as compared to logically integrated file systems. Figure 2.2 depicts this behavior. Dynamic resource allocation can, however, cause applications with different performance requirements to interfere with each other. For example, best-effort textual applications can issue a large number of requests and cause deadlines of real-time continuous media requests to be violated. Hence, a key challenge in designing physically integrated file systems is to develop mechanisms that isolate applications with different performance requirements, while providing all the benefits of dynamic resource allocation.

In what follows, we examine the merits and limitations of static resource partitioning and dynamic resource allocation for each resource managed by a file system, namely disk bandwidth, storage space and buffer space.

2.1.1 Disk Bandwidth Management

Due to immense sizes and bandwidth requirements of data types such as audio and video, integrated file systems use disk arrays as their underlying storage medium. Logically integrated file systems statically partition the set of disks in the array among component file servers, which enables each server to employ a different disk scheduling algorithm for servicing requests. Depending upon the performance requirements of applications, a disk scheduling algorithm that optimizes an appropriate performance criterion can be chosen. Since each server optimizes a single performance criterion, all applications accessing the server are provided with a single class of service regardless of their needs. Consequently, the server is unable to optimize data retrieval for applications that access a common set of files but have different performance requirements (e.g., a video editor that requires interactive service and a video player that requires real-time service, both of which access video files). Another disadvantage of partitioning disk bandwidth is that skewed or dynamically changing workloads can cause disks assigned to certain servers to idle, while those assigned to other servers are saturated, leading to wastage of disk bandwidth.

Since physically integrated file systems share disk bandwidth among all application classes, they employ single disk scheduling algorithms that can simultaneously support disk requests with different performance requirements. The design of such disk scheduling algorithms poses certain challenges. First, these algorithms must support multiple service classes and align the service provided with application needs. For instance, they must provide low average response times to interactive best-effort requests, or meet the deadlines of real-time requests, and so on. Second, these algorithms must isolate application classes form each other so as to prevent interference and starvation. Third, they must adapt to dynamically fluctuating workloads. Finally, they must prevent bandwidth unused by a class from being wasted by reallocating it to other classes.

2.1.2 Storage Space Management

Static partitioning of storage space enables logically integrated file systems to employ different placement policies for each component server. This simplifies storage space management but can lead to under-utilization of storage space. Adding storage space can alleviate this problem. While the hardware costs of such an approach are small, the system administrative costs can be substantial [94]. Dynamic sharing of storage space enables physically integrated file systems to adapt to skewed and changing space usage. It also requires the file system to employ mechanisms that support multiple placement policies, so as to provide the same flexibility as logically integrated file systems. Such mechanisms must prevent interference between conflicting policies and minimize disk fragmentation.

2.1.3 Buffer Space Management

Static partitioning of the buffer cache in logically integrated file systems enables each server to employ a different cache replacement policy for its partition. This enables each policy to independently optimize the cache hit ratio for its partition. Static partitioning can lead to under-utilization of the buffer cache during periods of skewed accesses and degrade the overall cache hit ratio. Dynamic sharing of the buffer cache overcomes these drawbacks. Such sharing, however, requires mechanisms that enable the coexistence of multiple cache replacement policies and minimize interference between policies competing for the same set of buffers.

Having qualitatively articulated the differences between logically integrated and physically integrated file systems, in what follows, we quantitatively compare the two architectures. Since disk bandwidth management has the most significant impact on the performance seen by applications, we restrict our quantitative evaluation to this resource.

2.2 Experimental Evaluation

2.2.1 Experimental Setup

To quantitatively compare the two architectures using simulations, consider an integrated file system that stores two types of data, video and text, and services two classes of applications, real-time and interactive best-effort. Although we choose only two applications classes and data types for our study, our results should be applicable to more general scenarios consisting of a large number of data types and application classes.

Supporting the real-time and interactive best-effort classes requires the logically integrated file system to employ two component servers, namely a text server and a video server. Let us assume that the file system employs an array of D disks to store data. These disks are assumed to be statically partitioned among the text and video servers and each server is assumed to stripe all files across all disks allocated to it. The text server is assumed to use the SCAN disk scheduling algorithm [81] to service requests, while the video server is assumed to use SCAN-EDF [72]. Whereas SCAN services requests in the increasing order of cylinder numbers so as to reduce seek overheads, SCAN-EDF services requests in the increasing order of deadlines; requests with identical deadlines are serviced in SCAN order.

In contrast, the physically integrated file system employs a single server to service both application classes and stripes all files across all D disks in the array. Since conventional disk scheduling algorithms such as SCAN are unsuitable for servicing disks requests with different performance requirements (see Figure 2.3), we assume that the Cello disk scheduling framework is used to service requests. The Cello disk scheduling framework: (i) supports multiple application classes and aligns the service provided within a class with application needs, (ii) protects application classes from one another, (iii) dynamically reassigns unused bandwidth to classes with pending requests and adapts to changes in workload, (iv) minimizes the seek time and rotational latency overhead incurred during access, and (v) is computationally efficient. Cello exploits characteristics of requests to align the service provided within each class with application needs. For instance, Cello delays the scheduling of real-time requests until their deadlines and uses the available slack to service interactive requests, and thereby provides low average response times to the latter without violating the deadlines of the former. Cello also allows weights to be assigned to each class and allocates bandwidth to classes in proportion to their weight; bandwidth unused by a class is reassigned to other classes with pending requests. Chapter 4 describes the Cello framework in detail.

Each video client accessing the file system is assumed to retrieve a variable bit rate (VBR) MPEG-1 file. The characteristics of MPEG-1 traces used in our experiments are described in Table 2.1. Video retrievals are assumed to be periodic and real-time—each video client requests f frames in each period and the deadline of each request is set to the end of the period. We use $f = 30$ and a period of 1 second for our experiments. Each text client is assumed to access a randomly selected file and interarrival times between successive requests from a client are assumed to be exponentially distributed. For our experiments, we assume 32KB requests with mean interarrival times of 1s. We investigate the behavior of the two architectures for different number of text and video clients, ranging from low levels of utilizations to near-capacity utilizations. These workloads are characteristics of many environments which are likely to use integrated file systems, such as web servers. Such servers typically operate at low to moderate utilization levels, but can often face transient overloads [4].

Given the above setup, we compare the two architectures using two different metrics— response time seen by interactive requests and percentage of deadlines violated for real-time requests. In what follows, we attempt to answer four questions:

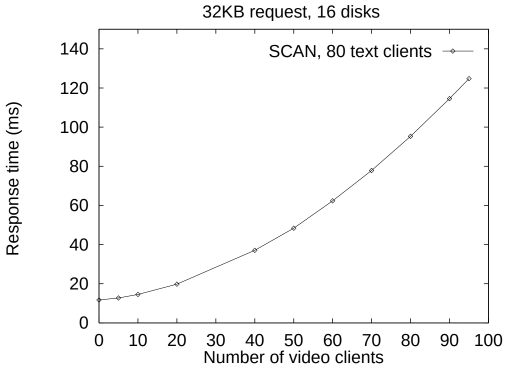


Figure 2.3: Response time of text requests with varying number of video requests. Since SCAN does not distinguish between different types of requests, response time of text requests degrades with increasing video load. Hence, SCAN is ineffective at isolating text and video requests from each other.

Table 2.1: Characteristics of MPEG-1 traces

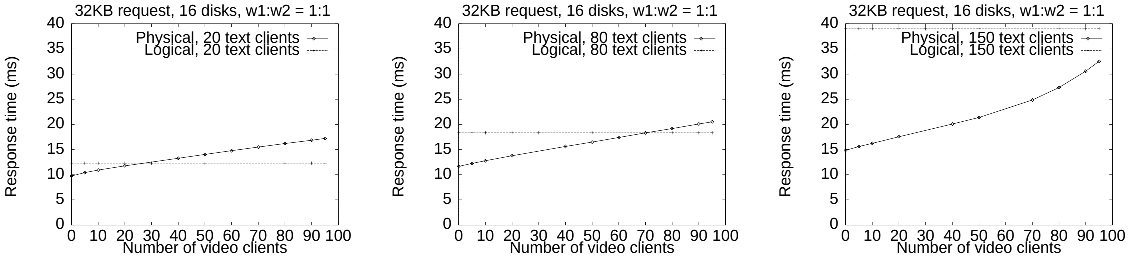
MPEG File	Encoding Pattern	Length (frames)	Bit rate Mb/s
Frasier	$I(BBP)^3BB$	5960	1.49
Newscast	$I(BBP)^3BB$	9000	2.33
Flintstones	$I(BBP)^3BB$	9000	1.67
Olympics	$I(BBP)^3BB$	9000	1.49

- How do the two architectures behave under different text and video loads?
- Which architecture yields better performance and in what operating region?
- How many extra disks are required for a particular server to match the performance of the other regardless of the load?
- How sensitive are our results to the parameters chosen for our experiments?

2.2.2 Behavior Under Different Loads

To determine the response time seen by text requests, consider an integrated file system consisting of 16 Seagate Elite3 disks. Let eight disks each be assigned to the text and video servers in the logically integrated file system, while all disks are shared by all data types in the physically integrated server. Let equal weights be assigned to the two classes employed by the Cello scheduler in the physically integrated file system (i.e., $w_1 : w_2 = 1 : 1$). Thus, in both architectures, 50% of the array bandwidth is allocated to each application class. Behavior under other partitioning ratios are examined in Section 2.2.5.

Figure 2.4 plots the response times of text requests for different video loads. Since text and video requests access mutually exclusive set of disks, the response time of text requests in the logically integrated file system is



(a) (b) (c)

Figure 2.4: Response time of interactive text requests in logically integrated and physically integrated servers. Figures (a), (b), and (c) plot the variation in response times for different video workloads and a background text load of 20, 80, and 150 clients.

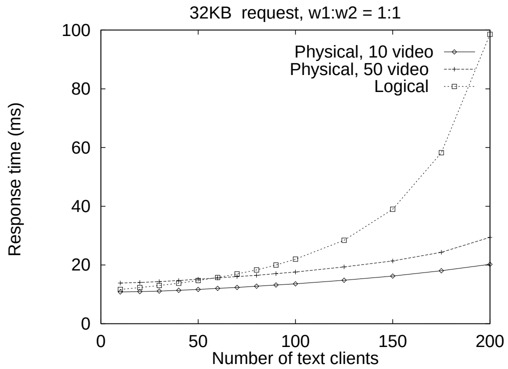


Figure 2.5: Response time of interactive text requests in logically integrated and physically integrated servers. The figure plots the variation in response times for different text workloads.

independent of the video load (see Figure 2.4(a)). The physically integrated file system uses Cello to service text and video requests. Cello uses slack stealing to isolate text requests from video requests; however this isolation is not total. Hence, the response time of text requests increases slowly with increase in video load (see Figure 2.4(a)). This increase is caused by two factors. First, increasing the video load increases the probability of a text request arriving when a video request is being serviced by the disk. Since requests in service cannot be preempted, the text request must wait until the request has been serviced. Second, increasing the video load also reduces the slack available to service text requests. In the absence of slack, video requests must be serviced before text requests; doing otherwise will cause deadlines of video requests to be violated. This results in increased queuing delays and degrades the response time of text requests.

Figure 2.5 plots the response times of text requests for different text loads and a fixed video load. As expected, increasing the text load causes the response time of text requests to increase in both architectures. The figure also shows that the magnitude of increase is larger in logically integrated file systems. To understand this behavior, consider the two factors determining the response time of a request, namely service time and queuing delay. The service time of a request is defined to be the summation of the seek time, rotational latency and transfer time. In the *average* case, the seek time and the rotational latency incurred by a request depends on the physical characteristics of the disk and is largely independent of the load. The transfer time depends on the request size and is also independent

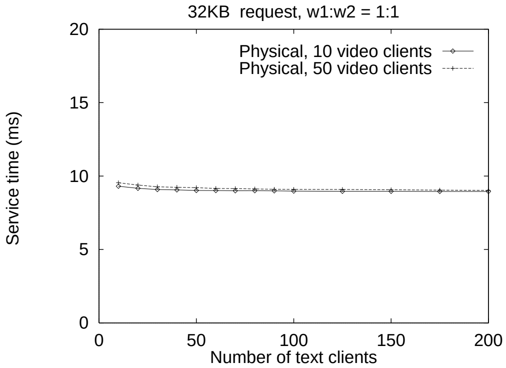


Figure 2.6: Service time incurred by a request for different workloads. The figure shows that the service time of a request is largely independent of the workload.

of the load. Hence, the service time of a request is largely independent of the load (see Figure 2.6). In contrast, the queuing delay incurred by a request completely depends on the system load. Since text files are interleaved across all disks in physically integrated file systems, the number of disks servicing text requests is larger than that in logically integrated file systems. This results in a smaller number of text requests per disks, and hence, shorter queues at each disk. Since requests incur a smaller queuing delay in physically integrated file systems, such file systems can yield better response times over a range of video loads. In fact, at heavy text loads, the queuing delay dominates the response time, causing the physically integrated server to outperform its counterpart *regardless of the video load* (see Figure 2.4(c)).

Figure 2.7 compares the percentage of deadlines violated for video requests in two architectures. The logically integrated server uses the SCAN-EDF disk scheduling algorithm to service video requests, whereas the physically integrated server uses Cello. As shown in the figure, at light and moderate video loads, both architectures meet deadlines of all real-time requests regardless of the text load. This is because text and video requests access mutually exclusive set of disks in the logically integrated server, and hence, text requests do not interfere with video requests. Since Cello services text requests only if sufficient slack is available, it ensures that text requests do not violate the deadlines of real-time video requests in the physically integrated server. The figure also shows that, at very heavy video loads, the logically integrated server saturates, resulting in request deadline violations. The physically integrated server, on the other hand, uses the bandwidth unused by text requests to service video requests. Hence, no request deadlines get violated even at very heavy video loads, so long as the text load is light. Increasing the text load, however, reduces the amount of bandwidth unused by text clients and causes deadlines of video requests to be violated.

2.2.3 Performance Comparison

A comparison of response times and percentage of deadlines violated at different workloads in Figures 2.4, 2.5 and 2.7 reveals the following:

- At light video loads, physically integrated file systems yield better response times than logically integrated file systems (since bandwidth unused by video requests is used to service text requests). See Figure 2.4(a).

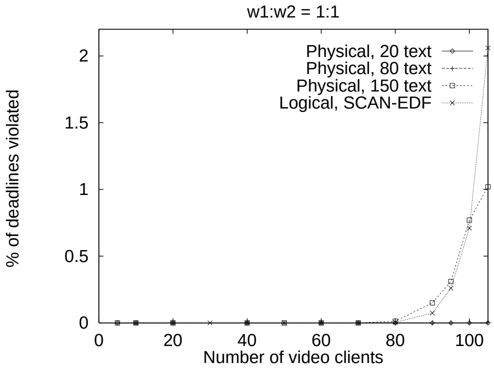


Figure 2.7: Percentage of deadlines violated for video requests in logically integrated and physically integrated file systems.

Table 2.2: Performance comparison

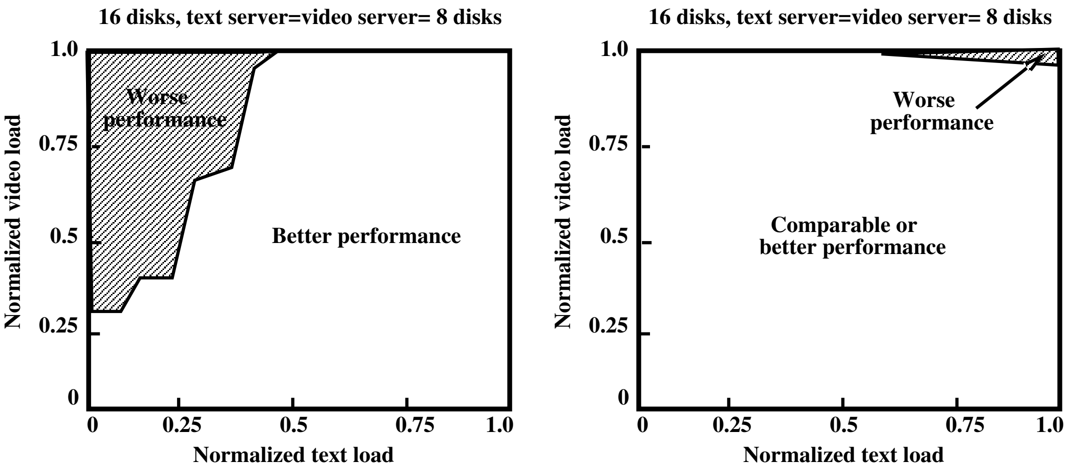
		Physically integrated server	
Video load	Text load	Response time	Percentage of deadlines violated
Light	Any	Better	Comparable
Heavy	Light	Worse	Better
Heavy	Heavy	Better	Comparable (marginally worse at very heavy loads)

- At heavy video loads and light text loads, physically integrated file systems yield worse response times than their counterpart (since the amount of slack available reduces, thereby increasing the queuing delay). See Figures 2.4(a) and (b).

- At heavy video and text loads, physically integrated file systems again outperform logically integrated file systems (since queuing delays dominate the response times and requests incur a smaller queuing delay due to the presence of a larger number of disks). See Figures 2.4(c) and 2.5.

- Both architecture meet deadlines of all video requests at light to moderate text and video loads. At heavy video and light text loads, the physically integrated server yields better performance than the logically integrated server (since unused text bandwidth can be used to service video requests). At heavy video and text loads, a small fraction of request deadlines get violated in both architectures. At such loads, the physically integrated server yields a marginally worse performance until the point where the video server saturates, after which the performance of the latter worsens. See Figure 2.7.

Table 2.2 tabulates these results, while Figures 2.8 depicts them pictorially. Figure 2.8 compares the performance of the two architectures over all possible workloads mixes. The X and Y axes plot normalized values of text and video workloads; a normalized load of 1 corresponds to one that saturates the logically integrated server. The non-shaded regions indicate workloads at which the physically integrated server either yields comparable performance or outperforms the logically integrated server. The figure demonstrates that sharing of disk bandwidth enables the



(a) Response time (b) Percentage of deadlines violated

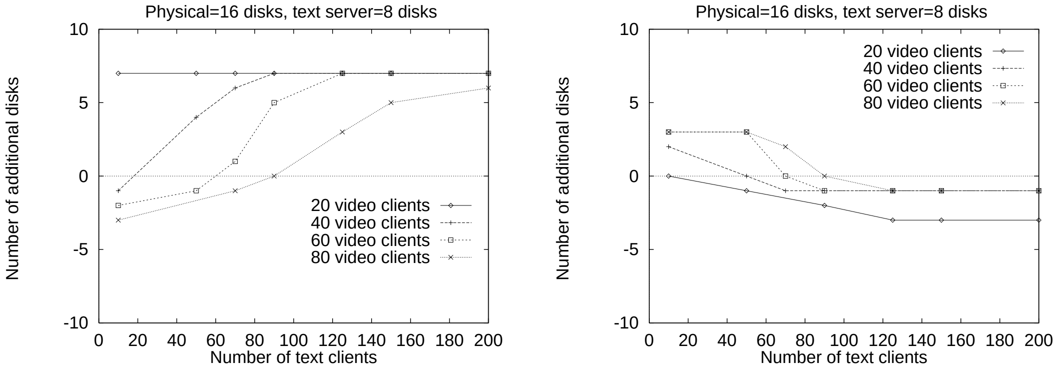
Figure 2.8: Performance under different workload mixes. The physically integrated file system yields worse performance in the shaded areas.

physically integrated server to provide comparable or better performance over a large operating region. Note that, the physically integrated file system can also operate in regions *beyond* those shown in Figure 2.8, whereas the logically integrated server saturates at such workloads (see Figure 2.2).

2.2.4 Overprovisioning Issues

Figure 2.8 compares the performance of the two architectures for different workload mixes; however it does not reveal the magnitude of the difference in performance. To quantify this difference, we computed the number of additional disks required by each architecture to match the performance of the other. Figure 2.9(a) plots the number of additional disks required for a 8 disk text server to match the performance of a 16 disk physically integrated server. As shown in the figure, the number of additional disks depends on the video load accessing the physically integrated server. The figure illustrates the following behavior:

- At light video loads, the bandwidth of all 16 disks can be used to service text requests in the physically integrated server. Hence, the text server requires as many disks as the physically integrated server (i.e., approximately twice as many disks) to provide comparable performance.
- Increasing the video load causes a corresponding decrease in the amount of slack available to service text requests in the physically integrated server. This causes the text response time to approach that in the logically integrated server; hence the latter needs fewer additional disks to match performance.
- Increasing the text load causes the queuing delay to dominate the response time. Since the physically integrated server uses all disks to service text requests, these requests incur a smaller queuing delay as compared to the logically integrated server. Hence, the number of disks required to match queuing delays (and response times) in the text server approaches that in the physically integrated server.



(a) Text server

(b) Physically integrated server

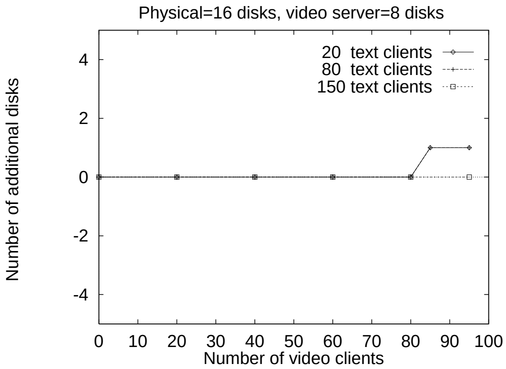
Figure 2.9: Amount of additional resources required to match performance for text requests. Figures (a) and (b) plot the number of additional disks required for an 8 disk text server to match the performance of a 16 disk physically integrated file server and vice versa, respectively.

The upper envelope of these curves yields a configuration that enables the text server to always match or outperform the physically integrated server. The envelope in Figure 2.9(a) indicates that the text server needs approximately twice as many disks to match the performance of the physically integrated server.

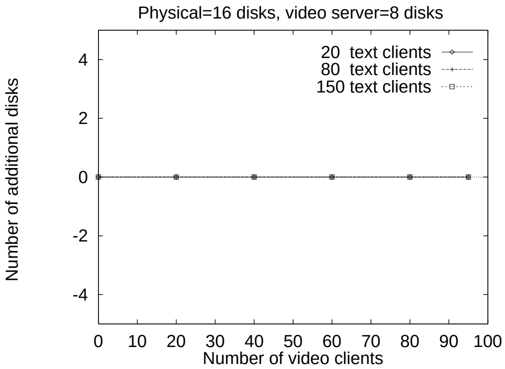
Figure 2.9(b) plots the number of additional disks required for a 16 disk physically integrated server to match the performance of an 8 disk text server. As explained in Section 2.2.3, the physically integrated file system yields worse response times at heavy video and light text loads. However, the difference in response times yielded by the two architectures is small, and hence, the physically integrated server requires only a small number of additional disks to match performance. Increasing the text load increases the queuing delay incurred by a request. Since the physically integrated server incurs a smaller increase in the queuing delay, the difference in response time decreases with increasing text load. As a result, the number of additional disks required to match performance also shows a corresponding decrease.

Figure 2.10 shows the number of additional disks required by an 8 disk video server to match the performance of a 16 disk physically integrated server and vice-versa. Since the disk scheduling policies employed by both file servers meet all request deadlines at light to moderate workloads, no additional disks are required for either server. At heavy video and light text loads, the physically integrated server outperforms the video server; hence the latter needs a few extra disks to match the performance of the former (see Figure 2.10(a)). At heavy text and video loads, the physically integrated server yields marginally worse performance. Since the difference in performance is very small (less than 5%), no additional disks are required (see Figure 2.10(b)).

We conclude from the above experiments that, the physically integrated server outperforms the logically integrated server by a substantial amount in certain operating regions and under-performs it by only a small amount in remaining regions. Consequently, a logically integrated file system needs a substantial amount of overprovisioning to match the performance of its counterpart. In contrast, a small amount overprovisioning is sufficient for the physically integrated file system to provide matching performance. Note that, we have restricted our focus to only two application classes in our experiments. The performance gains due to dynamic resource allocation are amplified with increase in number of application classes. This is because, in the worst case, each component server in the logically integrated file system may require as many disks as the physically integrated server to provide comparable performance (especially



(a) Video server



(b) Physically integrated server

Figure 2.10: Amount of additional resources required to match performance for video requests. Figure (a) and (b) plot the number of additional disks required for an 8 disk video server to match the performance of a 16 disk physically integrated file server and vice versa, respectively.

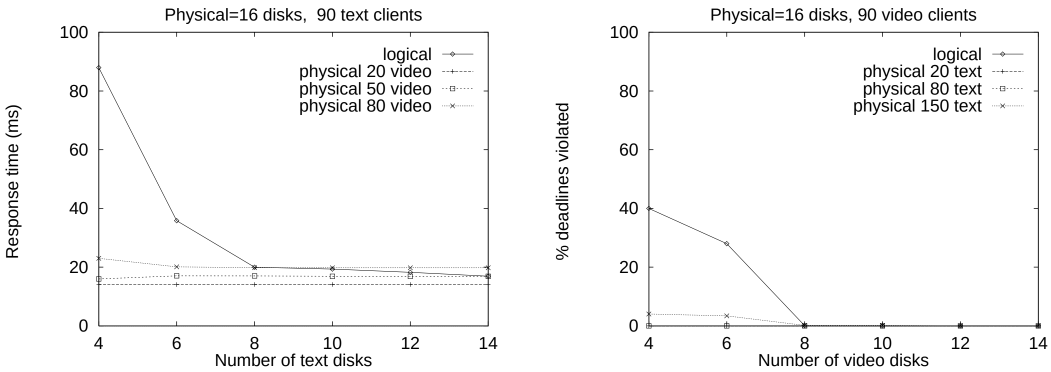
for workloads in which only that class is active).

2.2.5 Sensitivity to Experimental Parameters

Our experiments so far have assumed that the disks in the array are equally partitioned among the text and video servers. Figure 2.11 plots the response time of text requests and the percentage of deadlines violated for video requests when the text and video servers are assigned different number of disks (the weights assigned to the two application classes in the physically integrated server correspond to the fraction of the total number of disks assigned to the two servers in the logically integrated server). The figure shows that the relative performance of a component server in the logically integrated server worsens with decreasing number of disks. Consequently, a larger number of additional disks would be required for that component server to match the performance of the physically integrated server. Conversely, a component server with a larger fraction of the disks would require fewer additional disks to match performance. Thus, given an array with a fixed number of disks, depending on the relative partition sizes, either the text server or the video server or both would need to be overprovisioned to provide comparable performance.

2.3 Concluding Remarks

In this chapter, we examined the merits and limitations of two methodologies for designing integrated file systems. We argued that use of a single physically integrated server for all applications is desirable in many environments over logically integrated file systems that employ separate servers for each application class. For such environments, we argued that employing separate servers for different application classes simplifies the design of logically integrated file systems, but can lead to under-utilization of resources. By employing a single server for all application classes, physically integrated file systems obviate the need for partitioning resources among multiple servers. The resulting statistical multiplexing of resources yields higher utilization and better performance. We demonstrated that physically integrated file systems outperform their counterpart by a substantial amount in a large operating regions



(a) (b)

Figure 2.11: Effect of varying the number of disks on performance. Figures (a) and (b) show the response time and percentage of deadlines violated for different number of disks

and under-perform them by only a small amount in other regions. Hence, each server in the logically integrated file system may need to be overprovisioned by a substantial amount to provide comparable performance (e.g., an eight disk text server needs twice as many disks to match the response time of a sixteen disk physically integrated server). A limitation of dynamic resource allocation, though, is that it can cause interference between applications belonging to different classes. Consequently, a key challenge in designing physically integrated file systems is to design mechanisms that isolate application classes and prevent interference, while providing all the benefits of resource sharing. Due to the inherent benefits of dynamic resource allocation, we choose the physically integrated file system architecture for designing Symphony and develop several mechanisms in the following chapters to address this design challenge.

Chapter 3

Symphony Architecture: An Overview

Everything should be made as simple as possible, but no simpler.

—Albert Einstein

Physically integrated file systems employ a single server that can efficiently support diverse application classes and manage heterogeneous data. The resulting statistical multiplexing gains enable such file systems to achieve high utilization of resources and yield better performance. Managing heterogeneity, however, requires physically integrated file systems to often reconcile conflicting requirements as well as employ techniques that can simultaneously optimize different performance criteria. This chapter examines the implications of managing heterogeneous applications and data on the architecture of physically integrated file systems.

The rest of this chapter is organized as follows. In Section 3.1, we formulate the requirements imposed on physically integrated file systems. In Section 3.2, we propose a two layer file system architecture for Symphony that meets these requirements and present an overview of various components of such an architecture. Finally, Section 3.3 presents our conclusions.

3.1 Requirements for a Physically Integrated File System

A physically integrated file system should efficiently utilize file system resources while meeting the requirements of heterogeneous applications and data. In what follows, we discuss the effect of this objective on the service model, the retrieval mode, as well as techniques for placement, fault tolerance, meta data management and caching.

3.1.1 Service Model

An integrated file system must support applications with different performance requirements. For instance, it must support: (i) interactive best-effort applications, such as word processors and compilers, that desire low average response times, (ii) throughput-intensive best-effort applications, such as ftp and http servers, that desire high average throughput, and (iii) real-time applications, such as audio and video players, that require performance guarantees from the file system. Most existing file systems provide a single class of service to all applications, regardless of their requirements. The UNIX file system, for instance, provides a best-effort service to all applications. If one could cost-effectively overprovision such file systems such that the offered load is always significantly smaller than the capacity, then a single class of service is adequate for meeting heterogeneous application requirements. However, the increasing variability in performance requirements of applications means that the aggregate usage is more variable. In such a scenario, overprovisioning for the worst case usage would require substantial amount of resources. Regardless of the amount of overprovisioning, a file system can increase the overall utility to applications by supporting multiple classes of service. This enables applications to use the service class that is most suited to their needs and simplifies application development. Instantiating multiple service classes requires the file system to employ (i) a disk scheduling algorithm that supports these classes and aligns the service provided with application needs, and (ii) resource reservation algorithms that provide performance guarantees to real-time applications.

3.1.2 Retrieval Architecture

Most conventional file systems employ the *client-pull* mode of retrieval, in which the server retrieves information from disks only in response to an explicit read request from a client.[1] Whereas the client-pull mode is suitable for textual applications, adapting continuous media applications for client-pull accesses is difficult. This is because maintaining continuity in continuous media playback requires that retrieval requests be issued sufficiently in advance of the playback instant. To do so, applications must estimate the response time of the server and issue requests appropriately. Since the response time varies dynamically depending on the server and the network load, client-pull based continuous media applications are non-trivial to develop [80]. Hence, most continuous media file servers employ the *server-push* (or *streaming*) mode of retrieval, in which the server periodically retrieves and transmits data to clients without explicit read requests. The server-push mode of retrieval is inappropriate for aperiodic text requests. Hence, to efficiently support multiple application classes, an integrated file system should support both the client-pull and the server-push retrieval modes.

3.1.3 Placement Techniques

Due to the large storage space and bandwidth requirements of continuous media, integrated file systems will use disk arrays as their underlying storage medium. To effectively utilize the array bandwidth, the file server must *interleave* (i.e., *stripe*) each file across disks in the array. Placement of files on such striped arrays is governed by two parameters: (1) the *stripe unit size*[2], which is the maximum amount of logically contiguous data stored on a single disk, and (2) the *the degree of striping*, which denotes the number of disks across which a file is striped. The characteristics of data stored on the array has a significant impact on both parameters. To illustrate, the large bandwidth requirements of real-time continuous media yields an optimal stripe unit size that is an order of magnitude larger than that for text [77]. Use of a single stripe unit size for all data types either degrades performance for data types with large bandwidth requirements (e.g., continuous media), or causes internal fragmentation in small files (e.g., textual files). Similarly, a policy that stripes each file across all disks in the array is suitable for textual data, but yields suboptimal performance for variable bit rate continuous media [77]. Moreover, since continuous media files can have a multi-resolution nature (e.g, MPEG-2 encoded video), an integrated file system can optimize the placement of such files by storing blocks belonging to different resolutions adjacent to each other on disk [78]. Such contiguous placement of blocks substantially reduces seek and rotational latencies incurred during playback. No

[1] Observe that, although such servers generally employ some prefetching and caching techniques to improve performance, due to the aperiodic nature of accesses, requests for retrieving data from disks are triggered only in response to explicit access requests from the client.

[2] A stripe unit is also referred to as a media block. We use these terms interchangeably in this dissertation.

such optimizations are necessary for "single resolution" textual files. Since placement techniques for different data types differ significantly, to enhance system throughput, an integrated file system should support multiple data type specific placement policies and mechanisms to enable their coexistence.

3.1.4 Failure Recovery Techniques

Since disk arrays are highly susceptible to disk failures, a file server should employ failure recovery techniques to provide uninterrupted service to clients in the presence of failures. Disk failure recovery involves two tasks: *on-line reconstruction*, which involves recovering the data stored on the failed disk in response to an explicit request for that data; and *rebuild*, which involves creating an exact replica of the failed disk on a replacement disk. Conventional textual file systems use mirroring or parity-based techniques for *exact* on-line reconstruction of data blocks stored on the failed disk [19, 67]. Continuous media applications with stringent quality requirements also require such exact on-line reconstruction of lost data. However, for many continuous media applications, *approximate* on-line reconstruction of lost data may be sufficient. For these applications, on-line failure recovery techniques that exploit the spatial and temporal redundancies present within continuous media data to reconstruct a reasonable approximation of the data stored on the failed disk have been developed [91]. Unlike mirroring and parity based techniques, these techniques do not require any additional data to be accessed for failure recovery, and thereby significantly reduce the recovery overhead. Hence, to enhance system utilization, an integrated file system should support multiple data type specific failure recovery techniques and mechanisms that enable their coexistence.

3.1.5 Caching Techniques

File systems employ memory caches to improve application performance. To efficiently manage these caches, conventional textual file systems employ cache replacement policies such as LRU. It is well known that LRU performs poorly for sequential data accesses [13], and hence, is inappropriate for continuous media. Policies that are tailored for sequential data accesses, such as Interval Caching [26], are more desirable for continuous media, but are unsuitable for textual data. Since an integrated file system must support applications with different access characteristics, use of a single cache replacement policy for all applications can degrade cache hit ratio. Consequently, to enhance utilization of the cache, an integrated file system should support multiple data type specific caching policies, as well as mechanisms that enable these policies to efficiently share the cache.

3.1.6 Meta Data Management

Most conventional file systems allow files to be accessed as a sequence of bytes (i.e., they do not assign any structure to files). For continuous media files, however, the logical unit of access is a video frame or an audio sample. Accessing such files in conventional file systems requires the application to separately maintain logical unit indices. Developing such applications would be simplified if the file system were to allow files to be accessed as a sequence of logical units. Such support is also required to implement the server-push architecture since the file system needs logical unit sizes (e.g., frame sizes) to determine the amount of data that must be accessed on behalf of clients. Consequently, an integrated file system should maintain meta data structures that allow both logical unit and byte level access to files, thereby enabling any data type specific structure to be assigned to files.

3.1.7 Extensibility

The past few years have seen the emergence of a new generation of applications with diverse requirements and data type with heterogeneous characteristics. Rapid advances in computing technologies are likely to accelerate this trend, leading to a proliferation of new applications and data types. An integrated file system should efficiently support both present and future applications and data types. Since it is difficult, if not impossible, to foresee requirements that will be imposed by future applications and data types, an integrated file system will need to employ an extensible architecture. Such an architecture will facilitate easy extensions to the file systems, thereby simplifying the integration of new application classes and data types into the file system.

3.1.8 Requirements Summary

The preceding arguments demonstrate that a physically integrated file system differs from existing file systems in several fundamental ways. Managing heterogeneity in application requirements and data characteristics is key to an integrated file system. To achieve this objective, an integrated file system should: (i) support multiple classes of service as well as the client-pull and server-push modes of retrieval, (ii) employ data-type specific policies for placement, caching, failure recovery and meta data management and mechanisms that enable their coexistence, and (iii) employ an extensible architecture.

We have developed Symphony, a physically integrated file system that meets these requirements. Next, we provide an overview of the Symphony architecture.

3.2 Architecture of Symphony

To meet the requirements outlined in Section 3.1, Symphony employs a two layer architecture. The lower layer of Symphony (data type independent layer) employs a set of mechanisms that provide core file system functionality. These mechanisms support multiple classes of service and enable the coexistence of diverse policies. The upper layer of Symphony (data type specific layer) consists of a set of modules, one per data type, which uses these mechanisms to implement data type specific policies. The layer also exports a file server interface that supports both the server-push and client-pull modes of file access. Figure 3.1 depicts this architecture. Such a two layer architecture provides a clean separation of data type independent mechanisms from data type specific policies. The architecture is extensible since it allows the file system designer to easily add modules for new data types, or modify policies implemented in existing modules. In what follows, we provide an overview of the mechanisms and policies that we have developed for the two layers of Symphony.

3.2.1 Mechanisms for Enabling Coexistence of Diverse Policies

Implementing diverse policies in the data type specific layer requires the development of mechanisms that enable their coexistence. To achieve this objective, we have developed the following mechanisms for disk scheduling, placement, failure recovery, caching and meta data management.

Disk Scheduling A desirable disk scheduling framework for an integrated file system must support multiple classes of service and align the service provided within each class with application needs. It must minimize disk seek

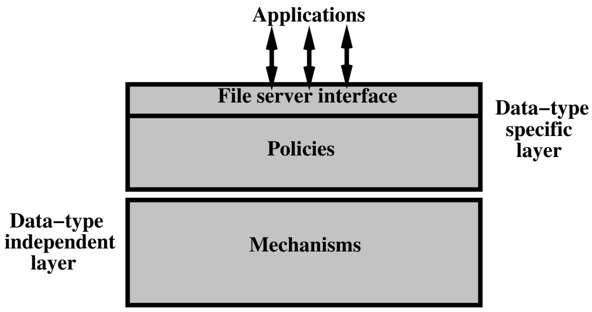


Figure 3.1: The Symphony two layer architecture

and rotational latency overheads to maximize disk throughput. The framework must protect applications in different classes from one other, reallocate unused bandwidth to classes with pending requests so as to adapt to changing workloads, and be computationally efficient.

We have developed the Cello disk scheduling framework that meets these objectives. Cello employs a two-level disk scheduling architecture, consisting of a *class-independent scheduler* and a set of *class-specific schedulers*. The two levels of the framework allocate resources at two time scales. The class independent scheduler governs the coarse-grain bandwidth allocation to application classes, while the class-specific schedulers control the fine-grain interleaving of requests from different application classes. The two levels of the architecture cleanly separate class-independent mechanisms from class-specific scheduling policies, thereby facilitating the coexistence of multiple class-specific schedulers.

To protect requests in an application class from starvation, Cello allows control over the amount of disk bandwidth allocated to each class. To do so, the class-independent scheduler requires weights to be assigned to classes and allocates disk bandwidth to classes in proportion to their weights. To ensure that bandwidth unused by a class is not wasted, the class independent scheduler utilizes this bandwidth to service pending requests from other classes. The class-specific schedulers exploit the characteristics of requests to derive a schedule that meets application needs and minimizes seek and rotational latency overhead. To demonstrate that Cello can support applications with diverse requirements, we have developed class-specific schedulers for a number of application classes such as soft real-time, interactive best-effort, and throughput-intensive best-effort.

Our experiments have demonstrated the efficacy of Cello in aligning the service provided with the application needs. For instance, with six text and six video clients (a disk utilization of 60%), the response time provided by Cello is 2.5 times smaller than SCAN, and no deadlines of video requests are violated. We have also shown that the overhead of scheduling requests in Cello is small (request rates of up to 50 requests/s result in a CPU load of less than 1%), thereby demonstrating that Cello is computationally efficient.

Chapter 4 describes the Cello framework in detail.

Placement Since the optimal stripe unit size and the degree of striping for continuous media differ significantly from those for textual and image data, use of a single placement policy for all data types reduces server throughput. We have developed a storage manager that enables multiple placement policies to coexist by: (i) supporting multiple block sizes, and (ii) allowing control over their placement on the disk array. The storage manager supports multiple block sizes by defining a base block size and constructing larger blocks by allocating a sequence of contiguous base blocks on disk. To allow control over their placement, the storage manager accepts location hints and allocates blocks at locations conforming to these hints. Together, these mechanisms allow diverse placement policies to be implemented. To illustrate, support for multiple block sizes allows different stripe unit sizes to be chosen for different data types. Moreover, by appropriately generating location hints, each file can be striped across all disks in the array, or only a subset of the disks. Location hints can also be used to cluster blocks of a file on disk, thereby reducing seek and rotational latency overheads incurred in accessing these blocks. By accepting locations hints, the storage manager exports the presence of multiple disks to the rest of the file system. This is a fundamental departure from existing file systems, which employ mechanisms such as logical volumes to hide the presence of multiple disks, albeit at the expense of providing no control over placement of blocks.

Failure Recovery We have developed a fault tolerance layer that enables multiple data type specific failure recovery policies to coexist within the file system. Whereas conventional fault tolerance techniques use a single mechanism for both on-line reconstruction and rebuild, the fault tolerance layer decouples the two tasks. Like conventional techniques, the fault tolerance layer uses parity information to rebuild failed disks onto spare disks. Unlike conventional techniques, however, the fault tolerance layer supports multiple online reconstruction policies. The fault tolerance layer achieves this objective by supporting two types of requests: (i) *reliable* requests, in which parity information is used to reconstruct blocks stored on the failed disk, and (ii) an *unreliable* requests, in which parity-based reconstruction is disabled, thereby shifting the responsibility of failure recovery to clients. Whereas reliable requests can be used for data types such as text that require exact reconstruction of data, unreliable requests can be used for approximate reconstruction of data (e.g., by using loss resilient compression algorithms). Thus, these mechanisms enable multiple failure recovery policies to be implemented in the file system.

Buffer Management Due to the differences in access characteristics of data types, no single cache replacement policy can provide optimal cache hit ratios. We have developed a buffer manager that enables multiple cache replacement policies (such as LRU, MRU, Interval Caching) to coexist within the file system. To achieve this objective, the buffer manager partitions the buffer cache and allows each cache replacement policy to independently manage its partition. The size of each cache partition is allowed to vary dynamically depending on the workload. The buffer manager services a request for a new buffer by evicting a cached buffer that is least likely to be accessed in the near future. To determine this buffer, the buffer manager queries each cache replacement policy for a candidate buffer for eviction. Each policy returns the buffer that is least likely to be accessed in its partition, and also computes a cost function, such as the time to reaccess, for this buffer. The buffer manager then compares the cost functions of all candidate buffers and evicts the buffer with the largest cost function. Such a mechanism improves the overall cache hit ratio by ensuring that buffers that are least likely to be accessed across all cache partitions are evicted first.

Meta data Management To enable data type specific structure to be assigned to files, we have developed a meta data structure that maintains a two level index for each file. Level one of the index maps logical access units (e.g., frames) to byte offsets, whereas level two maps byte offsets to disk block locations. Use of such a two level index enables a file to be accessed as a sequence of logical units. Moreover, by using only the second level of the index, byte level access can also be provided to files. Thus, by appropriately defining the logical unit of access, any data type specific structure can be assigned to a file.

3.2.2 Policies for Managing Heterogeneous Data types

The design of the data type specific layer requires the development of policies for managing storage space, disk bandwidth, and buffer space, and for handling disk failures. Whereas these policies for textual data are well known, the corresponding policies for continuous media have not been adequately investigated. Consequently, we focus on the development of policies for managing continuous media data. Recent research efforts have developed policies for managing disk bandwidth and buffer space for continuous media (e.g., the SCAN-EDF disk scheduling algorithm [72], the Interval Caching policy [26], etc.). Hence, we confine our focus to placement and failure recovery for continuous media.

Placement Policies The performance of striped disk arrays is governed by the stripe unit size and the degree of striping used to interleave files. We have developed techniques for determining the optimal values of these parameters for continuous media workloads. A key insight behind our techniques is that a stripe unit size and degree of striping that minimizes the tail of the response time distribution yield optimal throughput for real-time continuous media workloads (unlike best effort text workloads for which minimizing the average response time yields optimal performance). Since the tail of the response time distribution is governed by the load on the most heavily loaded disk in the array, the stripe unit size and the degree of striping that minimize the load on this disk are considered optimal.

We have developed analytical models that use the server configuration and workload characteristics to predict the load on the most heavily loaded disk in redundant and non-redundant disk arrays. We have validated our models through simulations and have used them to: (1) evaluate the effect of various system parameters (such as the number of clients, number of disks, etc.) on the stripe unit size, and (2) derive techniques for selecting an optimal stripe unit size for various design scenarios. Our models have shown that the optimal stripe unit size is governed by two parameters: the load imbalance across disks in the array, and disk seek and rotational latency overheads. Our results have shown that, contrary to conventional wisdom, a very large stripe unit size does not necessarily yield good file server performance. Instead, such a stripe unit size increases load imbalance across disks and adversely affects real-time performance guarantees provided to continuous media clients, thereby reducing server throughput.

As for the degree of striping, we have developed an analytical model to determine the number of disks across which a continuous media file should be striped. Our model has shown that, in relatively small disk arrays, striping continuous media files across all disks in the array (referred to as wide striping) yields a balanced load and maximizes throughput. However, the number of clients supported increases sub-linearly with increase in the number of disks, and hence, wide striping is an inadequate load balancing mechanism for large disk arrays. Consequently, to maximize throughput, the server must partition such arrays and stripe each continuous media file across a single partition (referred to as narrow striping). We have used our model to determine an optimal partition size that minimizes the load imbalance across partitions, and thereby, maximizes the number of clients supported.

Chapter 5 describes these analytical models in detail.

Failure Recovery Policies Conventional failure recovery policies use parity information for online reconstruction of data in the event of disk failures. Parity-based reconstruction can, however, substantially increase the load on surviving disks and adversely affecting the real-time performance guarantees provided to continuous media clients. To overcome these limitations, we have developed two failure recovery techniques that exploit the characteristics of the data to minimize the overhead of online reconstruction. Our first technique exploits spatial and temporal redundancies inherent in video files to *approximately* reconstruct lost video data (unlike techniques for textual data that exactly reconstruct lost data). Since human perception is tolerant to minor distortions in video playback [69], such a technique is adequate for a large class of continuous media applications. The technique partitions each image in a video file into multiple sub-images and stores them on separate disks. Each image must be partitioned such that, in the event of a disk failure, spatial and temporal redundancies can be used to reconstruct a reasonable approximation of the original image. A straightforward approach that assigns successive pixels of an image to consecutive sub-images ensures this property, but also reduces the correlation between pixels within a sub-image, thereby degrading compression efficiency. A key challenge is to develop image partitioning techniques that ensure reasonable reconstruction, without affecting compression efficiency. The recovery technique achieves this objective by partitioning each image in the *compressed* domain, thereby eliminating any adverse effects of partitioning on compression efficiency. We demonstrate the efficacy of our technique by developing loss resilient versions of JPEG and MPEG compression algorithms. Our technique decouples the process of approximate reconstruction from that of perfect rebuild of a failed disk—a fundamental departure from conventional failure recovery techniques. Furthermore, the technique enhances the scalability of integrated file systems by: (1) integrating online reconstruction with the decompression of video files at client sites, and thereby reducing the recovery overhead at the server to zero; and (2) supporting graceful degradation in the quality of recovered images with increase in the number of disk failures. Our experiments have demonstrated that the quality of reconstructed images using our technique is adequate for most continuous media applications. Finally, our recovery technique is also effective in masking packet losses resulting from network congestion, and hence, is an end-to-end solution for failure recovery.

For applications that require exact reconstruction of data, we have developed a recovery technique that exploits the sequential nature of continuous media accesses to reduce the reconstruction overhead in parity-based arrays. Our technique computes each parity block over a sequence of data blocks from the same continuous media file. This enables data blocks retrieved for playback to be used for online reconstruction and vice-versa, thereby reducing the load on the array. We have shown that our technique reduces the worst case reconstruction overhead by a factor of $(G - 1)$ as compared to parity-based recovery techniques, where G is the parity group size.

Chapter 6 describes both failure recovery techniques in detail.

3.3 Concluding Remarks

In this chapter, we first examined the requirements imposed on a physically integrated file system. We argued that managing heterogeneity in application classes and data characteristics is key to integrated file systems. We proposed a two layer file system architecture for Symphony that meets this requirements. The lower layer of such an architecture employs a set of data type independent mechanisms that provide core file system functionality, while the upper layer uses these mechanisms to implement data type specific policies. We then presented an overview of the policies and mechanisms for disk scheduling, placement, failure recovery, caching and meta data management that we developed for our two layer architecture. The next three chapters describe some these mechanisms and policies in more detail, while Chapter 7 describes their implementation in the Symphony prototype.

Chapter 4

Cello Disk Scheduling Framework

Why work on processors when I/O is where the action is?
—Dave Patterson, Keynote address, ACM SIGMETRICS'93 conference

Since the invention of movable head disks, several algorithms have been developed to improve I/O performance through intelligent scheduling of disk accesses. These algorithms can be broadly divided into two classes:

1. Disk scheduling algorithms optimized to service best-effort requests: The simplest of these algorithms is First Come First Served (FCFS), that schedules requests in the order of their arrival. Since the access schedule thus derived is independent of the relative positions of the requested data on disk, FCFS scheduling can incur significant seek time and rotational latency overhead. This limitation has been addressed by several disk scheduling algorithms, such as Shortest Seek Time First (SSTF), SCAN, LOOK, V(R), etc., that schedule requests to minimize seek time [21, 22, 28, 33, 37, 81, 96]; and Shortest Total/Access Time First (STF/SATF), Aged Shortest Access Time First (ASATF), etc., that schedule requests to minimize the total seek time and rotational latency overhead [43, 76].

2. Disk scheduling algorithms optimized to service requests with real-time deadlines: The simplest of these algorithms is Earliest Deadline First (EDF) [54]. EDF schedules requests in the order of their deadlines but ignores the relative positions of requested data on disk in deriving the access schedule. Hence, it can incur significant seek time and rotational latency overhead. This limitation has been addressed by several disk scheduling algorithms, including Priority SCAN (PSCAN), Earliest Deadline SCAN, Feasible Deadline SCAN (FD-SCAN), SCAN-EDF, Shortest Seek Earliest Deadline by Order/Value (SSEDO, SSEDV) [1, 15, 20, 72], etc. These algorithms start from an EDF schedule and reorder requests so as to reduce the seek and rotational latency overhead without violating request deadlines.

Unlike the systems for which these scheduling algorithms were designed, today's general purpose file and operating systems simultaneously support applications with diverse performance requirements [7, 59]. For instance, a typical file server today services requests from interactive best-effort applications (e.g., word processors); real-time applications (e.g., video and audio players); and file transfer applications (e.g., http servers). Interactive applications require the file server to minimize the average response time of requests. Real-time video playback applications require the file server to retrieve successive video frames prior to their playback instants (i.e., deadlines). However, due to the periodic nature of video playback, these applications do not benefit if the frames are retrieved much prior to their deadlines. Finally, file transfer applications require the server to provide high throughput across several requests, but are less concerned about the response times of individual requests.

With the manifold increase in CPU processing power, network bandwidth, and disk capacity, it is inevitable that general purpose computing environments of the future will support applications of even greater complexity and diversity. We can anticipate that future integrated file systems will support applications that process massive amounts of data for visualization and support real-time interactivity. For instance, a repository of satellite imagery might be accessed and processed by programs for feature extraction and real-time visualization; an application for interactive navigation through virtual environments will issue requests for the storage and retrieval of heterogeneous information objects (e.g., imagery, 3-D models, video, etc.) from distributed file servers under real-time constraints. Since most conventional disk scheduling algorithms are optimized for a single performance criterion, they are ineffective at simultaneously supporting applications with such diverse requirements.

Most of the techniques developed to-date for addressing this problem employ simple adaptations of conventional disk scheduling algorithms. To illustrate, consider a mix of real-time and best-effort applications. A real-time disk scheduling algorithm can be adapted to service these applications by modeling the requests generated by best-effort applications as a periodic task with deadlines [53]. This modeling, however, is non-trivial and introduces artificial constraints that reduce the effectiveness of the system. Another common approach for servicing the mix of real-time and best-effort applications is to employ a scheduler that assigns priorities to application classes and services disk requests in the priority order. Unfortunately, such schedulers may violate service requirements of requests and induce starvation [2]. Finally, simply enhancing a conventional scheduler by allocating time-slices to service requests from different application classes may incur substantial seek and rotational latency overhead (for short time-slices) or yield unacceptable response times (for long time-slices) (see Figure 4.1). Consequently, existing disk scheduling algorithms and their simple adaptations are unsuitable for integrated file systems. The design of a disk scheduling framework that can support applications with diverse performance requirements is the subject matter of this chapter.

The rest of the chapter is organized as follows. We derive requirements for a disk scheduling algorithm suitable for integrated file systems in Section 4.1. Section 4.2 describes and analyzes the Cello disk scheduling framework that we have developed. Section 4.3 presents the results of our experimental evaluation. Section 4.4 presents related work, and finally, Section 4.5 summarizes our results.

4.1 Requirements for a Disk Scheduling Algorithm

To determine a suitable disk scheduling algorithm, consider the requirements imposed by applications likely to be simultaneously supported by integrated file systems:

- *Real-time applications:* These applications require the file system to provide performance guarantees. Depending on the strictness of the requirements, these applications can be classified as either *hard real-time* or *soft real-time* applications. Whereas hard real-time applications require deterministic guarantees for the response time of each disk request, soft real-time applications require statistical guarantees. Request generation in these applications can either be *periodic* or *aperiodic*, and the applications may consume data immediately following its availability or at predefined instants. For example, video playback is a periodic, soft real-time application, in which the accessed video frames are consumed at predefined instants (determined by the video

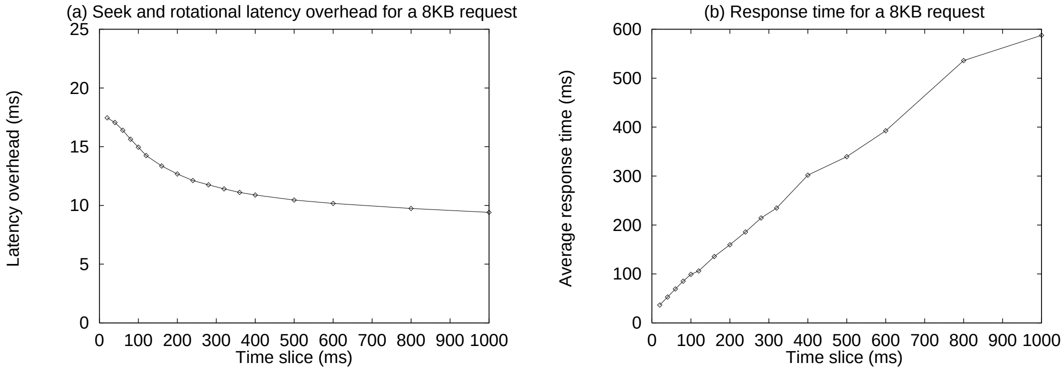


Figure 4.1: Effect of time-slicing on throughput and response times. A scheduler that allocates time-slices to applications in a round-robin manner can incur substantial seek time and rotational latency overhead while switching from one application class to the next. Increasing the duration of time-slices, and thereby servicing multiple requests from each application class within each slice, reduces the fraction of disk bandwidth wasted in switching between application classes. However, this increases the response time for requests.

playback rate and the consumption instants of previous frames). In contrast, applications that support interactive navigation through virtual environments yield real-time requests with low average response time requirements.

EDF and fixed priority schedulers are suitable for hard real-time applications [54], while scheduling algorithms such as FD-SCAN and SSEDV/SSEDO are are suitable for soft real-time applications [1, 20]. Just-in-time schedulers (which schedule requests just prior to their deadlines) are desirable for real-time applications that initiate data consumption at deadlines (e.g., video playback). Finally, algorithms that schedule requests at the earliest possible instants prior to their deadlines, and thereby minimize the response time while meeting the real-time requirements, are suitable for interactive real-time applications.

- *Best-effort applications*: These applications do not need performance guarantees. They can be further classified as either *interactive* or *throughput-intensive*. Interactive applications require low average response times. Throughput-intensive applications require the file system to sustain high throughput across multiple requests, but are less concerned about the response times of individual requests. For instance, word processors are interactive best-effort applications, while file transfer is a throughput-intensive best-effort application.

  Conventional disk scheduling algorithms such as SCAN, SSTF, SATF, etc. are suitable for these applications.

From this, we conclude that different policies are suitable for scheduling disk requests from different application classes. Hence, to align the service provided with the application needs, a disk scheduling framework should employ different policies for different application classes. Furthermore, such a framework should protect application classes from one another. For example, bursty arrival of best-effort requests should not cause deadline violations for real-time requests; and the arrival of a burst of real-time requests should not starve best-effort requests.

These requirements can be met by partitioning disk bandwidth among the application classes, and then employing an application-specific policy to schedule requests within each partition. However, the granularity of partitioning should be chosen such that (1) the seek time and rotational latency overhead incurred while servicing requests is minimized, and (2) the service provided is aligned to the application requirements. Finally, to efficiently utilize disk bandwidth, the framework must be *work-conserving* (i.e., it should utilize the idle disk bandwidth available to one application class to schedule pending requests from another class); and should adapt to changes in the work-load.

In summary, a disk scheduling algorithm suitable for integrated file systems should: (i) align the service it provides with the application needs, (ii) protect application classes from one another, (iii) be work-conserving and adapt to changes in work-load, (iv) minimize the seek time and rotational latency overhead incurred during access, and finally (v) be computationally efficient. In what follows, we present a disk scheduling framework that meets these requirements.

4.2 The Cello Disk Scheduling Framework

4.2.1 Architectural Principles

Cello achieves the above objectives by allocating disk bandwidth to application classes at two time-scales. At the coarse time-scale, it determines the number of requests from each application class to be serviced, and at the fine time-scale, it determines the order for servicing the set of requests from these classes. Whereas the former enables Cello to protect application classes from one another as well as adapt disk bandwidth allocation with changing work-load, the latter enables it to align the service provided to the application requirements while minimizing the seek time and rotational latency overhead.

These two tasks naturally map to a *two-level disk scheduling architecture*, consisting of a *class-independent* and a set of *class-specific* schedulers. The class-independent scheduler governs the *coarse-grain bandwidth allocation* to application classes, while the class-specific schedulers control the *fine-grain interleaving* of requests from the application classes. Moreover, they separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the co-existence of multiple class-specific schedulers. The concepts of allocating disk bandwidth at two time-scales and separating application-independent mechanisms from application-specific policies are the key contributions of the Cello framework.

To service n application classes, Cello uses a class-independent scheduler $\mathcal{C}$ and n class-specific schedulers $\mathcal{S}_i, i \in [1, n]$, and maintains $(n + 1)$ queues: n *pending queues*, one per application class and a *scheduled queue* (see Figure 4.2). Newly arriving requests are placed in the class-specific pending queues, and are eventually moved to the scheduled queue. Requests are dispatched for service from the scheduled queue. The class-independent scheduler determines *when* and *how many* requests are moved from each pending queue to the scheduled queue. To do so, it defines an *interval* (namely, the coarse time-scale), assigns *weights* w_i ($w_i \geq 0$) to the application classes, and during each interval, determines the number of requests to be moved from the pending queues to the scheduled queue such that the disk bandwidth allocated to application classes is in proportion to their weights. The class-specific schedulers utilize the state of the scheduled queue exported by the class-independent scheduler to determine *where* to insert these requests in the scheduled queue.

Observe that the two-level disk scheduling framework of Cello facilitates the development of a *service manager* that allocates disk bandwidth as per the requirements of applications [45]. To illustrate, if an application requests hard (soft) real-time service, then the service manager can use a deterministic (statistical) admission control algorithm that utilizes the disk bandwidth allocated to hard (soft) real-time application class to determine if the request can be satisfied, and if so, assign the request to the appropriate application class. Similarly, the service manager can monitor the workload from each application class, and then adapt the disk bandwidth allocations (namely, the values

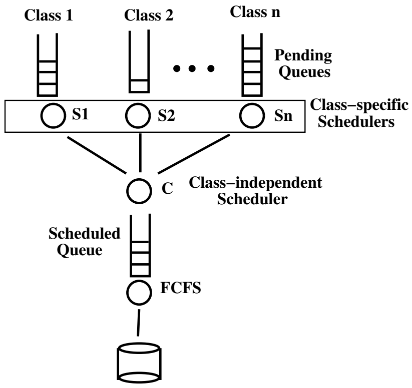


Figure 4.2: The architecture of the Cello disk scheduling framework

of $w_1, w_2, ..., w_n$) accordingly. The development of such techniques is beyond the scope of this dissertation.

In what follows, we describe the design of the class-independent scheduler and present some examples of class-specific schedulers.

4.2.2 The Class-independent Scheduler

The class-independent scheduler performs two functions. First, it determines when and how many requests from each application class should be inserted in the scheduled queue. Second, it exports an interface that enables the class-specific schedulers to determine where to insert requests in the scheduled queue.

4.2.2.1 Allocating Disk Bandwidth to Application Classes

Consider a file server servicing n application classes. Let w_i ($w_i \geq 0$) be the weights assigned to the application classes, and let the duration of an interval be $\mathcal{P}$. The class-independent scheduler $\mathcal{C}$ allocates disk bandwidth to application classes using one of two methods: (i) *proportionate time-allocation*, in which the fraction of an interval for which the disk services requests from an application class is proportional to its weight; and (ii) *proportionate byte-allocation*, in which the amount of data accessed within an interval for an application class is proportional to its weight. Note that since each request may incur a different seek and rotational latency overhead and may access different amounts of data, the above two methods yield different allocations. In what follows, we describe the techniques for achieving proportional time- and byte-allocation; we will discuss their relative merits in Section 4.3.

Proportionate time-allocation

In the proportionate time-allocation method, for each interval, the class-independent scheduler $\mathcal{C}$ maintains the disk idle time $\mathcal{I}$, and the total service time $\mathcal{U}_i, i \in [1, n]$ — which includes seek time, rotational latency, and transfer time — expended for scheduling requests from class i. At the beginning of each interval, $\mathcal{I}$ and $\mathcal{U}_i$ are initialized to 0. Whereas $\mathcal{U}_i$'s are updated when a request is inserted in the scheduled queue, $\mathcal{I}$ is updated every time the disk is idle.

At all times, $\mathcal{C}$ ensures that the following inequality holds:

$$\sum_{j=1}^{n} \mathcal{U}_j + \mathcal{I} \leq \mathcal{P} \tag{4.1}$$

To allocate disk service time to application classes in proportion to their weights, $\mathcal{C}$ operates as follows: For each class i, if

$$\mathcal{U}_i < \frac{w_i}{\sum_{j=1}^{n} w_j} \cdot (\mathcal{P} - \mathcal{I}) \tag{4.2}$$

then $\mathcal{C}$ invokes the class-specific scheduler for class i (namely, $\mathcal{S}_i$). If the pending queue for class i is not empty, then $\mathcal{S}_i$ returns with a request r, and the desired location for inserting r in the scheduled queue[1]. The location is specified by a $[prev, next]$ pair, where $prev$ and $next$ are two successive requests in the scheduled queue.

On receiving this, $\mathcal{C}$ first computes: (i) τ, the total seek time, rotational latency and transfer time incurred in servicing request r if it is inserted after request $prev$;[2] and (ii) τ_{next}^{new}, the time to service request $next$ after servicing request r. $\mathcal{C}$ then verifies that:

1. The insertion of r in the scheduled queue does not increase the total service time expended for class i beyond its proportionate share. That is,

$$\mathcal{U}_i + \tau \leq \frac{w_i}{\sum_{j=1}^{n} w_j} \cdot (\mathcal{P} - \mathcal{I}) \tag{4.3}$$

2. The cumulative service time expended for all classes does not exceed the available interval duration. Since the insertion of r in the scheduled queue changes the service time of $next$, but does not affect the service time of any other requests, this condition is formulated as:

$$\sum_{k=1}^{n} \mathcal{U}_k + \tau + (\tau_{next}^{new} - \tau_{next}) \leq (\mathcal{P} - \mathcal{I}) \tag{4.4}$$

   where τ_{next} is the service time for $next$ prior to the insertion of r.

If (4.3) and (4.4) are satisfied, then $\mathcal{C}$ inserts r in the scheduled queue between $prev$ and $next$, and updates $\mathcal{U}_i$ as:

$$\mathcal{U}_i = \mathcal{U}_i + \tau \tag{4.5}$$

Moreover, if request $next$ belongs to class j, $\mathcal{C}$ updates $\mathcal{U}_j$ as:

$$\mathcal{U}_j = \mathcal{U}_j + (\tau_{next}^{new} - \tau_{next}) \tag{4.6}$$

Specifically, if $\tau_{next}^{new} < \tau_{next}$ (i.e., if insertion of r reduces the service time for request $next$), then (4.6) credits the utilization of class j; and if $\tau_{next}^{new} > \tau_{next}$, then $\mathcal{U}_j$ is incremented appropriately. This symmetric model of update makes $\mathcal{U}$ independent of the order in which requests are inserted in the scheduled queue, and dependent solely on the order in which requests are serviced by the disk.

[1] The techniques for selecting a request and determining the desired location for inserting it in the scheduled queue are discussed in Sections 4.2.2.2 and 4.2.3.

[2] The service time of a request can be computed using empirically derived disk models [51]. These models use logical block numbers of disk requests to compute the service time. Studies have shown that use of logical block numbers, rather than physical disk locations, for disk scheduling does not introduce any significant errors [96].

Once the request is inserted, $\mathcal{C}$ re-evaluates (4.2) for class i and repeats the above procedure. $\mathcal{C}$ switches from one class to another either when the pending queue for the class is empty or when the request selected by the class can't be inserted into the scheduled queue (i.e., either (4.3) or (4.4) are violated). In the former case, class i becomes a member of $\mathcal{E}$, the set of application classes that have not fully used their service time allocation; otherwise, class i becomes a member of $\widehat{\mathcal{E}}$, the set of classes all of whose requests can't be inserted into the scheduled queue due to violation of either (4.3) or (4.4).

To effectively utilize disk bandwidth, $\mathcal{C}$ employs a *lazy* approach to distribute the unused service time allocation of the classes in $\mathcal{E}$ to service pending requests from classes in $\widehat{\mathcal{E}}$. Specifically, if the scheduled queue is empty and if none of the of the classes in $\mathcal{E}$ have a pending request, then $\mathcal{C}$ schedules requests from classes in $\widehat{\mathcal{E}}$ *one at a time*.[3] The selection of a request from classes in $\widehat{\mathcal{E}}$ for insertion into the scheduled queue involves two steps:

1. *Determination of the set of classes $\widehat{\mathcal{E}}'$ ($\widehat{\mathcal{E}}' \subseteq \widehat{\mathcal{E}}$) eligible for utilizing the unused service time*: The eligibility of a class depends on the service time distribution policy: $\mathcal{C}$ can allocate the unused service time among all the classes in $\widehat{\mathcal{E}}$ in proportion of their weights or in a priority order. In the former case, all the classes in $\widehat{\mathcal{E}}$ are eligible, while in the latter case, highest priority classes with non-empty pending queues are eligible. Hybrid policies—in which classes at the same priority level receive service time in proportion to their weights—are also possible.

2. *Selection of a request from one of the eligible classes for insertion into the scheduled queue*: This selection is governed by two requirements:

   (a) Proportionate distribution of unused service time among the eligible classes: For each eligible class $i \in \widehat{\mathcal{E}}'$, $\mathcal{C}$ maintains $\mathcal{X}_i$, which measures the unused service time of classes in $\mathcal{E}$ expended to schedule requests of class i within an interval. At the beginning of each interval, $\mathcal{X}_i$'s are initialized to 0; and $\mathcal{X}_i$ is updated when a class i request is scheduled to utilize unused bandwidth allocation of classes in $\mathcal{E}$. To ensure proportionate distribution of unused service time among the eligible classes, $\mathcal{C}$ selects a request of class i only if

   $$\mathcal{X}_i < \frac{w_i}{\sum_{j \in \widehat{\mathcal{E}}'} w_j} \cdot (t_{end} - t) \tag{4.7}$$

   where t and t_{end} denote the current time and the completion time of the current interval. If none of the classes in $\widehat{\mathcal{E}}'$ meet this requirement, then $\mathcal{X}_i$'s are re-initialized to 0; and the process is repeated.

   (b) Minimizing the seek time and rotational latency overhead incurred in servicing selected requests: To meet this requirement, $\mathcal{C}$ invokes the class specific scheduler of every class in $\widehat{\mathcal{E}}'$ that satisfies (4.7) for a request, and selects a request that is closest to the current disk head position for insertion into the scheduled queue.

To ensure that (4.1) will be satisfied after the insertion, $\mathcal{C}$ verifies that

$$\sum_{j=1}^{n} \mathcal{U}_j + \mathcal{I} + \tau \leq \mathcal{P} \tag{4.8}$$

[3]This enables $\mathcal{C}$ to schedule a request from any of the classes in $\mathcal{E}$ immediately upon its arrival.

where τ is the service time of the request selected for insertion. If (4.8) is satisfied, $\mathcal{C}$ inserts the selected request into the scheduled queue, and updates $\mathcal{X}_i$ as:

$$\mathcal{X}_i = \mathcal{X}_i + \tau \tag{4.9}$$

Moreover, to ensure that the allocation available to classes in $\mathcal{E}$ is reduced appropriately (see 4.2), the service time τ incurred by the inserted request is charged to the idle time $\mathcal{I}$. That is, the idle time is updated as

$$\mathcal{I} = \mathcal{I} + \tau \tag{4.10}$$

Proportionate Byte-allocation

In the proportionate byte-allocation method, for each interval, the class-independent scheduler $\mathcal{C}$, maintains $\mathcal{I}$, the disk idle time $\mathcal{I}$; and $\mathcal{U}_i, i \in [1, n]$, the total service time expended for scheduling requests from class i. Additionally, $\mathcal{C}$ maintains $\mathcal{B}_i, i \in [1, n]$, which denotes the total number of bytes accessed during an interval for class i. Just as in the proportionate time-allocation method, $\mathcal{C}$ ensures (4.1) holds at all times.

To ensure that the amount of data accessed within an interval for an application class is proportional to its weights, the byte-allocation method, unlike the proportionate time-allocation method, charges the same (namely, the average) seek and rotational latency overhead per byte incurred within an interval to all request classes, regardless of the actual latency overhead incurred by individual requests. Specifically, given the values of $\mathcal{U}_i$ and $\mathcal{B}_i$, $\mathcal{C}$ defines $\mathcal{O}$, the average service time per byte incurred within an interval, and $\mathcal{V}_i$, the amount of time already expended for servicing requests of class i as:

$$\mathcal{O} = \frac{\sum_{i=1}^{n} \mathcal{U}_i}{\sum_{i=1}^{n} \mathcal{B}_i}; \qquad \mathcal{V}_i = \mathcal{B}_i \cdot \mathcal{O} \tag{4.11}$$

and then allocates disk bandwidth to classes as follows: For each class i, if

$$\mathcal{V}_i < \frac{w_i}{\sum_{j=1}^{n} w_j} \cdot (\mathcal{P} - \mathcal{I}) \tag{4.12}$$

then $\mathcal{C}$ invokes $\mathcal{S}_i$ to obtain a request r and the desired location for its insertion (specified by a $[prev, next]$ pair). Just as in the time-allocation method, $\mathcal{C}$ determines τ, τ_{next}, and τ_{next}^{new}, and then computes the new value of $\mathcal{O}$ as:

$$\mathcal{O}^{new} = \frac{\sum_{k=1}^{n} \mathcal{U}_k + \tau + (\tau_{next}^{new} - \tau_{next})}{\sum_{i=1}^{n} \mathcal{B}_i + b} \tag{4.13}$$

where b denotes the amount of data (in bytes) accessed by r. It then verifies the following two conditions:

1. The insertion of r in the scheduled queue does not increase $\mathcal{V}_i$ beyond its proportionate share. That is,

$$\mathcal{V}_i^{new} = \mathcal{O}^{new} \cdot (\mathcal{B}_i + b) \leq (\mathcal{P} - \mathcal{I}) \cdot \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{4.14}$$

2. The cumulative service time expended for all classes does not exceed the available interval duration. That is,

$$\mathcal{O}^{new} \cdot \left(b + \sum_{i=1}^{n} \mathcal{B}_i \right) \leq \mathcal{P} - \mathcal{I} \tag{4.15}$$

Observe that substituting (4.13) into (4.15) yields (4.4). If (4.14) and (4.15) are satisfied, then $\mathcal{C}$ inserts r in the scheduled queue between $prev$ and $next$, updates $\mathcal{U}_i$'s using (4.5) and (4.6), and updates $\mathcal{B}_i$ as:

$$\mathcal{B}_i = \mathcal{B}_i + b \tag{4.16}$$

Once the request is inserted, $\mathcal{C}$ re-evaluates (4.12) for class i and repeats the above procedure. To utilize the unused byte allocation of a class to service pending requests from other classes, $\mathcal{C}$ employs a technique similar to the one used in the proportionate time-allocation method.

4.2.2.2 Exporting the State of Scheduled Queue

Best-effort applications are elastic, and can tolerate variations in the response times of individual requests. Real-time applications are more rigid and impose deadline requirements with each request. Consequently, a class-specific scheduler must determine a position for inserting a request that does not inadvertently violate the deadlines of the requests already in the scheduled queue.

To assist a class-specific scheduler in determining the insertion position, the class-independent scheduler must export the state of the scheduled queue along with the constraints imposed by real-time requests. Cello defines these constraints in terms of *slack*.

Definition 4.1 Given the set of requests in the scheduled queue, the slack of a request is defined as the difference between the latest time by which the disk must begin servicing the request and the earliest time at which the disk may begin servicing the request.

The latest time by which the disk must begin servicing a request is governed either by (i) the deadline (if any) associated with the request, (ii) the latest time by which the disk must begin servicing the next request in the queue, or (iii) the completion time of the current interval. Let the scheduled queue contain R requests. Let τ_i denote the service time incurred for the i^{th} request in the scheduled queue, and let d_i denote its deadline. Also, let t_{end} denote the completion time of the current interval. Then the latest time l_i by which the disk must begin servicing the i^{th} request is given by:

$$l_i = \begin{cases} \min(d_R, t_{end}) - \tau_R & i = R \\ \min(d_i, l_{i+1}) - \tau_i & 1 \leq i \leq (R-1) \end{cases} \tag{4.17}$$

Note that if the i^{th} request is a best-effort request, then $d_i = \infty$.

The earliest time at which the disk may begin servicing a request depends on the service time of all the requests preceding it in the queue. Specifically, if t denotes the current time, then the earliest time e_i at which the disk may begin servicing the i^{th} request ($i \in [1, R]$) in the queue is given by:

$$e_i = \begin{cases} t & i = 1 \\ e_{i-1} + \tau_{i-1} & 2 \leq i \leq R \end{cases} \tag{4.18}$$

Given l_i and e_i, the slack of the i^{th} request is defined as:

$$s_i = \max(0, l_i - e_i) \tag{4.19}$$

Thus, s_i defines the duration for which the i^{th} request can be delayed without violating the deadlines of any of the requests in the scheduled queue.

The class-independent scheduler must recompute slack values after insertion of each request. Since this involves recomputing l_i and e_i for each request in the scheduled queue, a naive slack computation algorithm has $O(R)$ complexity. However, the following relationships between the slack values of consecutive requests in the scheduled queue can be utilized to significantly reduce the overhead of slack computation.

Lemma 4.1 The slack of a best-effort request is equal to the slack of the request following it in the scheduled queue.

Proof: If position i in the scheduled queue contains a best-effort request (i.e., $d_i = \infty$), using (4.17), (4.18), and (4.19) we get:

$$l_i - e_i = (l_{i+1} - \tau_i) - e_i = l_{i+1} - e_{i+1} \tag{4.20}$$

Hence, $s_i = \max(0, l_i - e_i) = \max(0, l_{i+1} - e_{i+1}) = s_{i+1}$. ■

The following two corollaries are a consequence of Lemma 4.1.

Corollary 4.1 Consecutive best-effort requests in the scheduled queue have identical slack values.

Corollary 4.2 The slack of a best-effort request is equal to that of the first real-time request following it in the scheduled queue.

Lemma 4.2 If consecutive real-time requests in the scheduled queue have the same deadline, then they have identical slack values.

Proof: Let two real-time requests at positions i and $i+1$ in the scheduled queue have identical deadlines (i.e., $d_i = d_{i+1}$). Then using (4.17) and (4.18), we get:

$$l_i - e_i = (\min(d_i, l_{i+1}) - \tau_i) - e_i = \min(d_{i+1}, l_{i+1}) - e_{i+1} \tag{4.21}$$

Since $l_{i+1} < d_{i+1}$ (from (4.17)), the lemma follows. ■

Lemma 4.3 Slack values increase monotonically in the scheduled queue.

Proof: Using (4.17) and (4.18), we get:

$$l_{i+1} - e_{i+1} = l_{i+1} - (e_i + \tau_i) = (l_{i+1} - \tau_i) - e_i \tag{4.22}$$

Since from the definition of l_i, $l_{i+1} - \tau_i \geq l_i$. Hence, $l_{i+1} - e_{i+1} \geq l_i - e_i$, and the lemma follows. ■

It follows from the above lemmas and corollaries that a sequence of best-effort requests followed by a sequence of real-time requests with the same deadline have identical values of slack. Hence, $\mathcal{C}$ can maintain a single value of slack for each such sequence rather than maintaining it for individual request. This significantly reduces the overhead of slack computation. As we illustrate in the next section, by exporting the values of s_i along with the state of the scheduled queue, the class-independent scheduler enables the class-specific schedulers to determine the position for inserting a new request into the scheduled queue.

The complete algorithm implemented by the class-independent scheduler in Cello is described in Figure 4.3. The figure describes the proportionate time-allocation method, the algorithm for proportionate byte-allocation can be described similarly.

loop forever

At the beginning of each interval, set $\mathcal{I}$, all $\mathcal{U}_i$s , and $\mathcal{X}_i$s to 0

if a class in $\mathcal{E}$ has a non-empty pending queue or a new interval begins

i := choose a class from $\mathcal{E}$ with a non-empty pending queue

$(r, [prev, next])$:= invoke the class specific scheduler S_i for a request and its insert position in the scheduled queue

τ := service time of request r if inserted after request $prev$

τ_{next} := current service time of request $next$

τ_{next}^{new} := new service time of request $next$ if r is inserted

if $\mathcal{U}_i + \tau \leq (\mathcal{P} - \mathcal{I}) \cdot \frac{w_i}{\sum_{j=1}^{n} w_j}$ and $\sum_{k=1}^{n} \mathcal{U}_k + \tau + (\tau_{next}^{new} - \tau_{next}) \leq (\mathcal{P} - \mathcal{I})$ then

Request the class specific scheduler S_i to delete r from its pending queue and insert it into the scheduled queue after request $prev$

Set $\mathcal{U}_i = \mathcal{U}_i + \tau$ and $\mathcal{U}_j = \mathcal{U}_j + (\tau_{next}^{new} - \tau_{next})$

Update slack values

fi

else if the pending queues of all classes in $\mathcal{E}$ are empty and the scheduled queue is empty

Determine the set of eligible classes $\widehat{\mathcal{E}'}$

Invoke the class-specific scheduler of each class in $\widehat{\mathcal{E}'}$ that satisfies $\mathcal{X}_i < \frac{w_i}{\sum_{j \in \widehat{\mathcal{E}'}} w_j} \cdot (t_{end} - t)$ for a request

if none of the classes in $\widehat{\mathcal{E}'}$ satisfy the above condition, then reset $\mathcal{X}_i$ to 0, $i \in \widehat{\mathcal{E}'}$ and repeat the above step

r := choose a request closest to the current disk head in the scan direction

τ := service time of request r

if $\sum_{k=1}^{n} \mathcal{U}_k + \mathcal{I} + \tau \leq \mathcal{P}$

Request the class specific scheduler S_i to delete r from its queue and insert it into the scheduled queue

Update $\mathcal{X}_i := \mathcal{X} + \tau$

Update idle time as $\mathcal{I} := \mathcal{I} + \tau$

fi

else {`* all queues are empty *`}

Sleep until a request arrives or a new round starts

Increment idle time $\mathcal{I}$ by the time for which disk was idle

fi

end loop

Figure 4.3: The class independent scheduling algorithm

4.2.3 The Class-specific Schedulers

Class-specific schedulers perform two functions. First, they order the requests in their pending queues in accordance with the application requirements. Second, when invoked by the class-independent scheduler, they determine the position for inserting the selected request into the scheduled queue and return it with the $[prev, next]$ pair to $\mathcal{C}$. In what follows, we describe the functionality of class-specific schedulers for the interactive best-effort, throughput-intensive best-effort, and a class of real-time applications.

4.2.3.1 Interactive Best-effort Applications

A scheduler for the interactive best-effort application class should minimize the response times observed by requests. It can achieve this objective as follows:

1. Select a request from the pending queue in the FIFO order.
2. Insert the selected request into the scheduled queue using the classic *slack stealing* technique [53]. Specifically, the scheduler inserts the request ahead of the request at position k in the scheduled queue only if the increase in service time yielded by inserting the request is smaller than s_k. This provides low average response time to best-effort requests without violating deadlines of real-time requests. To minimize the seek time and rotational latency overhead, sequences of best-effort requests are maintained in SCAN/SATF order.

   As per the slack stealing technique, if the request can be inserted at more than one location in the scheduled queue, then the scheduler can employ either a *first-fit*, a *best-fit*, or a *hybrid* policy for determining the insertion location. Whereas the first-fit policy minimizes the response time of the request, potentially at the expense of overall disk throughput; the best-fit policy maximizes disk throughput at the expense of a higher response time. A hybrid policy can balance these tradeoffs by selecting the location i for insertion that minimizes $c_i = \beta r_i + (1 - \beta)\tau_i$, $(0 \leq \beta \leq 1)$, where r_i and τ_i, respectively, denote the response time and the service time for the request if it is inserted at location i in the scheduled queue. Note that with $\beta = 1$ the hybrid policy reduces to first-fit, and with $\beta = 0$ it reduces to best-fit.

4.2.3.2 Throughput-intensive Best-effort Applications

Throughput-intensive applications (e.g., ftp) require the disk scheduler to sustain high throughput, but they are less concerned about the response times of individual access requests. A scheduler can meet this requirement as follows:

1. Select a request from the pending queue in FIFO order.
2. Insert the selected request towards the tail of the scheduled queue, and order the sequence of requests from throughput-intensive best-effort applications in SCAN/SATF order.

4.2.3.3 Real-time Applications with Periodic Consumption

Video playback is a periodic, soft real-time application, in which the accessed video frames are consumed at predefined instants (determined by the video playback rate and the consumption instants of previous frames). Hence, it is an example of a real-time application with periodic consumption.

Just-in-time schedulers — that service requests just prior to their deadlines — are desirable for this class of applications. Most of the known real-time disk scheduling algorithms can be adapted to derive a suite of just-in-time schedulers with different properties. In what follows, we derive a just-in-time scheduler from the SCAN-EDF algorithm [72].

The SCAN-EDF real-time disk scheduling algorithm orders requests based on their deadlines, and then schedules requests with the same deadline in the SCAN order. This algorithm can be adapted to achieve just-in-time service as follows:

1. Select a pending request for insertion into the scheduled queue in the earliest deadline first (EDF) order.

2. Determine the set of feasible positions for inserting the selected request in the scheduled queue, and then insert the request at the last feasible position. For a request with deadline d, a position k in the scheduled queue is considered feasible if:

$$t + \tau + \sum_{j=1}^{k-1} \tau_j \leq d \quad \text{and} \quad \forall j \in [1, k-1] : d_j \leq d \tag{4.23}$$

   where τ is the service time incurred by the request if it is inserted at position k in the scheduled queue, and t is the current time. This condition ensures that the requests with deadlines are in the EDF order in the scheduled queue. Moreover, by selecting the largest value of k for inserting the request, the scheduler groups together requests with the same deadline and thereby facilitates the use of SCAN for their service.

4.2.4 Complexity Analysis

For a file server servicing n application classes, the computational complexity of the operations performed by the Cello framework are as follows:

- *Determining the position for inserting a request into the scheduled queue:* The complexity of this operation depends on the insertion policy (e.g., first-fit, best-fit, etc.) employed by the class-specific scheduler. In the worst case, if the scheduled queue contains R requests, then this operation takes $O(R)$ time.

- *Computing slack values:* The class-independent scheduler must recompute slack values after insertion of each request. Since this involves recomputing l_i and e_i for each request in the scheduled queue, a naive slack computation algorithm has $O(R)$ complexity. However, as demonstrated in Section 4.2.2.2, a sequence of best-effort requests followed by a sequence of real-time requests with the same deadline have identical values of slack. Hence, $\mathcal{C}$ can maintain a single value of slack for each such sequence rather than maintaining it for individual request. This significantly reduces the overhead of slack computation.

- *Reassigning idle bandwidth:* To minimize the seek time and rotational latency overhead incurred while reassigning unused disk bandwidth, $\mathcal{C}$ selects for insertion into the scheduled queue a request, from a class in $\widehat{\mathcal{E}}'$, that is closest to the current disk head position. Since the number of request classes is usually small and since, at any time, only one request from each of the pending queues is considered for insertion into the scheduled queue, a linear search through all eligible classes may suffice. If the number of request classes is large, then the algorithm can maintain a binary search tree to efficiently choose the next request to be scheduled. Since the tree contains no greater than n elements, searches, insertions and deletions are $O(\log n)$ operations, while initial construction of the tree is an $O(n \log n)$ operation.

- *Insertions and deletions from the scheduled queue:* Once the position for inserting a request is determined, by maintaining the scheduled queue as a doubly linked list, insertion into the scheduled queue becomes an $O(1)$ operation. Also, since requests are always deleted from the head of a queue, deletions are $O(1)$ operations.

4.2.5 Discussion

The two-levels of the Cello framework separate class-independent mechanisms from class-specific scheduling policies, and thereby facilitate the coexistence of multiple class-specific schedulers. By isolating the implementations of class-specific schedulers, it simplifies the development and modification of policies employed by class-specific schedulers. In what follows, we illustrate that the information hiding essential for achieving flexibility and extensibility in the Cello framework may yield sub-optimal schedules as compared to a monolithic scheduler that utilizes complete knowledge about the requirements of requests from all the application classes. We also discuss the trade-offs of using intervals for scheduling disk requests.

Invocation Order of Class-specific Schedulers

Example 4.1 Consider a file server servicing requests from a real-time and a best-effort class. Let us assume that the real-time class has two pending requests a and b; and the best-effort class has a pending request c. Let the time required to service each requests be 10ms. Let the current time be 0, and the deadline for both a and b be 20ms. Consider the case that the class-independent scheduler $\mathcal{C}$ first invokes the scheduler for real-time class (say S_1) and then the scheduler for the best-effort class (say S_2). Since a and b are inserted first, there is no slack for inserting c before a or b. Hence, the resulting schedule is a, b, c; which meets all the deadlines.

If, instead, $\mathcal{C}$ had invoked S_2 before S_1, then c will be the first request inserted into the scheduled queue. Since the deadline of a can be met even if it serviced after c, S_1 will insert a after c in the scheduled queue. Unfortunately, this results in the slack of a to be equal to 0. Hence, b can't be inserted ahead of a, resulting in the schedule c, a, b. However, this schedule violates the deadline requirements of b. ■

This example demonstrates that the order for invoking the class-specific schedulers impacts the feasibility of the resulting schedule. To minimize the possibility of generating infeasible schedules, $\mathcal{C}$ should invoke class-specific schedulers in the order defined by the strictness of their requirements. For instance, by inserting requests from the real-time class prior to requests from the best-effort class, the scheduler can minimize the possibility of best-effort requests violating the deadlines of real-time requests. Such a policy, however, is only a heuristic.

In general, deriving a schedule that best meets the requirements of application classes while incurring the smallest seek and rotational latency overhead requires complete knowledge about the constraints (e.g., deadlines) and disk location to be accessed for all the requests in the pending queues. Since the layered architecture of Cello restricts $\mathcal{C}$ to accessing only the requests at the head of the pending queues, the resulting schedule may be sub-optimal. However, as we show in Section 4.3, the reduction in disk throughput due to such a sub-optimal schedule is small in practice.

Aggressive versus Lazy Insertion

Once an application class is selected, the class-specific scheduler can employ an *aggressive* policy that inserts as many requests as possible into the scheduled queue. In such a policy, the number of requests of a class inserted into the scheduled queue will be constrained by the bandwidth allocation for the class and the number of requests in its pending queue. As the following example illustrates, such an aggressive policy may violate the requirements of requests.

Example 4.2 Consider a scenario where a large number of real-time requests with late deadlines (i.e., ones that expire in future intervals) arrive before a request with an early deadline (i.e., one that expires in the current interval). The aggressive insertion policy may spend the entire bandwidth allocation to schedule requests with late deadlines. This may prevent the request with early deadline from being inserted into the scheduled queue in the current interval, violating its deadline. ■

Alternately, the class-specific scheduler can employ a *lazy* policy that delays the insertions of requests with late deadlines as long as possible to reduce the possibility of denying service to requests with early deadlines. Implementing such a policy requires the class-independent scheduler to know when to invoke a particular class-specific scheduler such that pending request deadlines are met; and the class-specific scheduler to know when any further delay of insertion will cause its unused allocation to be assigned to other classes. Design of such lazy class-specific schedulers is the subject of future research.

Determining the Interval Length

The interval length places a limit on the maximum number of requests that can be inserted in to the scheduled queue per interval. Use of a large interval enables Cello to batch a large number of requests, and thereby optimize disk seek and rotational latency overheads. In contrast, use of a small interval reduces the maximum number of requests that a class specific scheduler would have to examine to determine the insert position of a request. This reduces the overhead of making scheduling decisions in Cello. The interval length must be chosen to balance these tradeoffs.

Discrete versus Moving Intervals

For efficiency reasons, Cello uses discrete intervals to schedule disk requests. An alternate approach is to allocate bandwidth to application classes over a moving interval. Although intuitively appealing, such an approach can significantly increase scheduling overhead. This is because a class-specific scheduler would need to examine the entire pending queue to determine the next request to be serviced (rather than only the request at the head of the pending queue). Moreover, allocating bandwidth over a moving interval places a limit on the number of requests belonging to any one class that can be batched together in the scheduled queue, which in turn increases the seek and rotational latency overhead.

4.3 Experimental Evaluation

We have built an event-based trace-driven disk simulator called *diskSim* to evaluate the Cello framework. For our simulations, we configured Cello with three applications classes—soft real-time, interactive best-effort, and throughput-intensive best-effort. The scheduler for interactive best-effort class used the first-fit policy to insert requests in the scheduled queue and maintained a sequence of best-effort requests in SCAN order (see Section 4.2.3.1). The scheduler for the throughput-intensive best-effort class inserted requests at the tail of the scheduled queue in

Table 4.1: Disk Parameters of Seagate-Elite3 disk

Disk capacity	2 GBytes
Bytes per sector	512 KB
Sector per track	99
Tracks per cylinder	21
Cylinders per disk	2627
Minimum seek time	1.7 ms
Maximum seek time	22.5 ms
Maximum rotational latency	11.1 ms
Average seek time	11.0
Average Transfer rate	4.6 MB/s

SCAN order (see Section 4.2.3.2), while that for soft real-time requests inserted requests using a just-in-time adaptation of SCAN-EDF (see Section 4.2.3.3). We simulated a Seagate Elite 3 disk that stores text and video files, using block sizes of 8KB and 64KB, respectively. (see Table 4.1 for the characteristics of this disk)

Each text client in our simulations selects a random file and reads it sequentially from beginning to end. Clients access the text files either using an interactive or a throughput-intensive best-effort application. In either case, the size of data accessed by each application request is normally distributed with a mean of 32KB, while the inter-arrival times of requests are exponentially distributed with a mean of 900ms. Each video client in our simulations emulates a video player and reads a randomly selected MPEG-1 file at 30 frames/s. Each MPEG-1 file has an average bit rate of 1.5 Mbit/s. All video clients are serviced in the server-push mode. In the server-push mode, the file server proceeds in terms of periodic rounds and accesses a fixed number of video frames during each round. Requests for all the frames to be accessed in a round are issued at the beginning of each round, and all of these requests have the end of the round as their deadline. These requests are serviced using the soft real-time class of Cello. Both the round duration and the interval length were set to 1000ms in our simulations. The length of each simulation run was such that the 95% confidence intervals are within 5% of the reported values. Since the objective of our study is to evaluate disk scheduling algorithms, we assume in our simulations that the system is limited only by the disk bandwidth (i.e., the cpu, memory, and system bus never become bottlenecks).

4.3.1 Aligning the Service to Application Needs

To demonstrate that a scheduling algorithm that aligns the service provided to application needs performs better than one that uses a single policy to schedule requests from different application classes, we compared the response times yielded by Cello to those obtained using SCAN. For this experiment, we configured Cello with two request classes—interactive best-effort and soft real-time—and assigned the same weight to both classes (i.e., $w_1 : w_2 = 1 : 1$). We varied the video load and the text load and measured the disk utilization, the response time of text requests, and the percentage of deadline violations yielded by Cello and SCAN for each combination of these parameters.

Figure 4.4 plots the disk utilization (i.e., percentage of the time the disk was busy) for different combinations of text and video workloads. Since the utilization is a function of the total workload, different combinations of text and

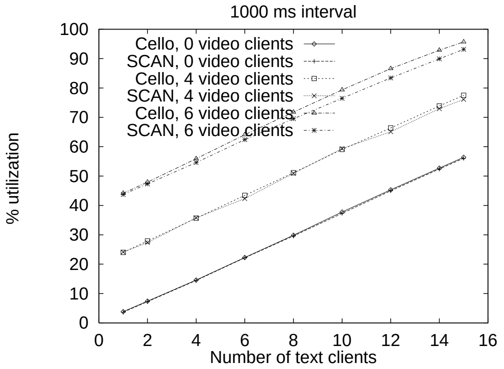


Figure 4.4: Disk utilizations for different video and text loads

video loads can yield the same utilization. In the rest of this section, we present results for specific combinations of text and video loads. Figure 4.4 can then be used to determine the utilization for a particular workload combination, thereby correlating metrics such as response time to disk utilization.

Figure 4.5(a) plots the response time of text requests for different video loads. The figure shows that the response time of text requests significantly increases with increase in video load when SCAN scheduling is used, whereas it is largely independent of the video load when Cello is used. Since SCAN does not distinguish between text and video requests while making scheduling decisions, increasing the video load increases the number of requests in the SCAN schedule, causing response time of text requests to increase. Cello, on the other hand, exploits slack values to schedule text requests ahead of video requests whenever possible. Consequently, it provides low average response time to text requests even at heavy video loads. For instance, with six text and six video clients (i.e., a disk utilization of 60%), the response time provided by Cello is a factor of 2.5 smaller than SCAN. For Cello, the small increase in response time at heavy loads is due to the decreasing availability of slack with increase in video load. Figure 4.5(b) plots the response time of text requests for different text loads and a fixed video load. The figure shows that, the response time provided by Cello is significantly lower than SCAN over a wide range of text workloads.

Figure 4.6(a) plots the percentage of video request deadlines that are violated for different text loads. For SCAN, increasing the text load increases the probability of deadline violations for video requests. For Cello, since the scheduler for the interactive best-effort class determines the position for inserting a request into the scheduled queue such that none of the deadlines are violated, increasing the text load has no effect on the probability of deadline violations.

Since throughput rather than response time is more important to throughput-intensive requests, Cello services these requests at the end of each interval in a single SCAN. This enables Cello to schedule interactive best-effort requests ahead of throughput-intensive requests, and thereby provide better response time to these requests. To demonstrate this behavior, we configured Cello with all three request classes and assigned them equal weights. We then measured the response time of interactive and throughput-intensive requests for a fixed video load. As expected, Cello provided lower response times to interactive requests as compared to throughput-intensive requests (see 4.6(b)).

The above experiments show that a scheduling algorithm designed for best-effort requests is unsuitable for servicing classes with different requirements. To show that this is true for real-time disk scheduling algorithms as well, we repeated the above experiment for the earliest deadline SCAN (D-SCAN) algorithm [1]. The D-SCAN

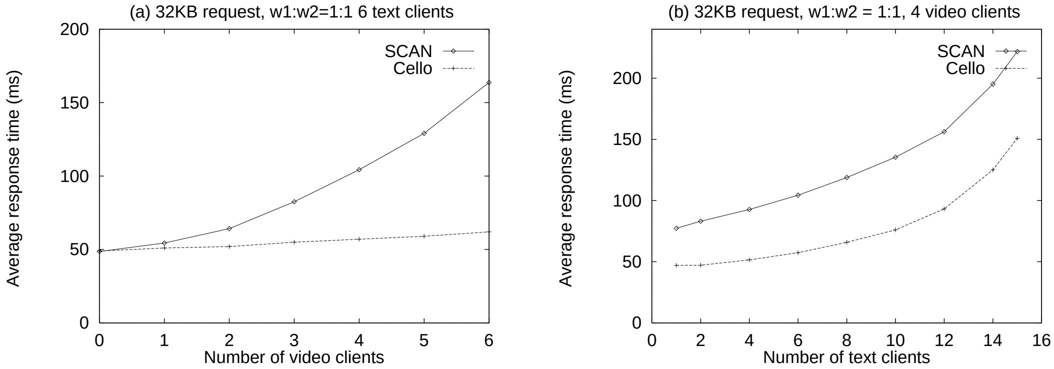


Figure 4.5: Response times of interactive best-effort requests in Cello and SCAN

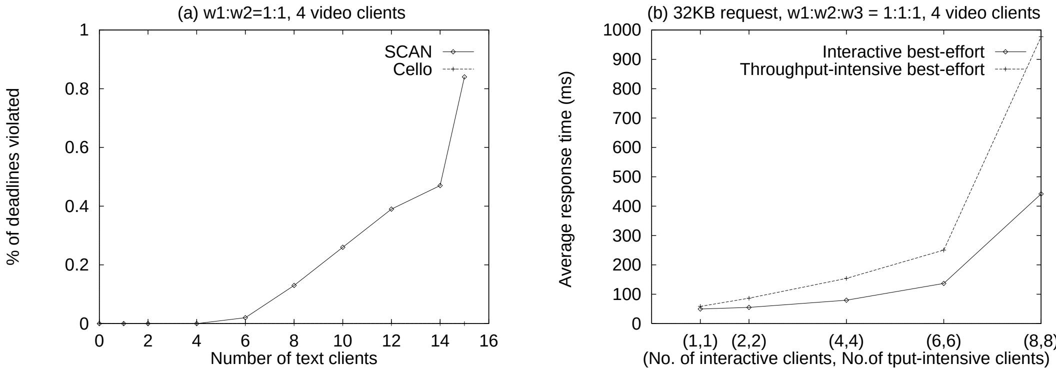


Figure 4.6: Performance of real-time and throughput-intensive best-effort classes.

algorithm determines the scan direction based on the requests with the earliest deadline, and services all requests along the way while scanning to this request. These requests are either best-effort requests or real-time requests with later deadlines. In the absence of any real-time requests, the scan direction is determined as in SCAN.[4] Note that, D-SCAN uses request deadlines only to determine the scan direction and does not distinguish between real-time and best-effort requests while servicing them in SCAN order. Hence, the presence of text requests in the schedule can interfere with video requests and vice-versa. This causes D-SCAN to yield worse performance than Cello (see Figure 4.7).

Together, Figures 4.5, 4.6 and 4.7 demonstrate that existing disk scheduling algorithms are inadequate for servicing application classes with differing requirements; Cello addresses this limitation by: (1) isolating request application from each other, and (2) aligning the service provided to the needs of applications.

4.3.2 Effect of Proportionate Allocation in Cello

Cello allocates the time spent in servicing requests among classes in proportion to their weights. To demonstrate this property, we configured Cello with all three request classes and assigned them equal weights (i.e., $w_1 : w_2 : w_3 = 1 : 1 : 1$). We then measured the time spent in servicing requests of each class. The video and text loads used for

[4]The algorithm as proposed considers only real-time requests. We trivially extended it to service best-effort requests as well.

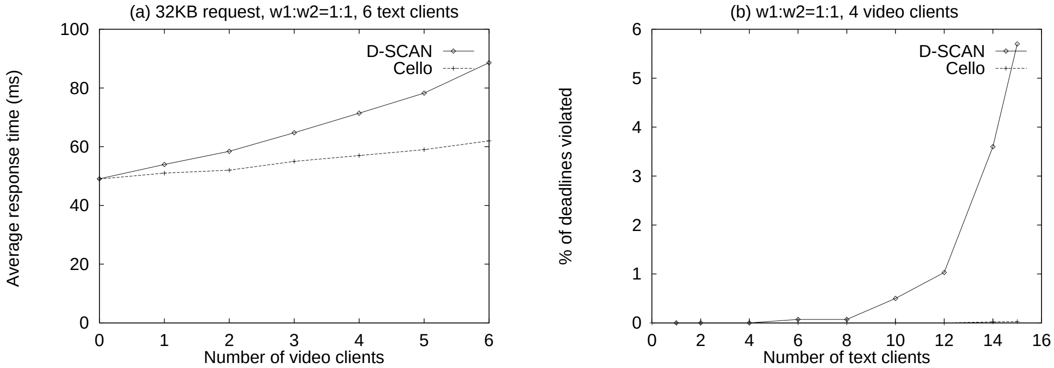


Figure 4.7: Performance of interactive and real-time requests in Cello and D-SCAN.

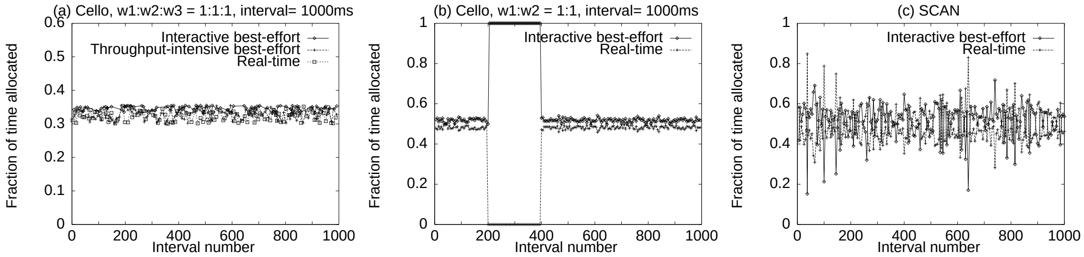


Figure 4.8: Partitioning disk bandwidth between request classes

our experiments were such that the pending queues never became empty, and hence, each class used up its entire allocation in each interval. As shown in Figure 4.8(a), Cello allocated the time spent in servicing requests within each interval equally among the three request classes.

To demonstrate that Cello allocates disk bandwidth unused by a class to classes with pending requests, we configured Cello with the real-time and the interactive best-effort classes and assigned equal weights to these classes. We conducted an experiment in which no video requests arrived from intervals 200 through 400. Consequently, the interactive best-effort class received the entire disk bandwidth in these intervals. In all other intervals, however, Cello allocated the time spent in servicing each requests equally among the two classes. Figure 4.8(b) demonstrates this behavior. Observe that, the actual allocation received by the best-effort class is slightly larger than that of the real-time class. Since the block sizes used for video and text files was 64KB and 8KB, respectively, the average service time of a video request is larger than that of a text request. Towards the end of each interval, any residual allocation that is unutilized by the real-time class (since it is insufficient to service any more video requests) is reassigned to the best-effort class. Since the service time of text requests is smaller than video requests, the reassigned allocation is often sufficient to service a text request, resulting in a larger allocation to the best-effort class.

Finally, Figure 4.8(c) shows that fraction of the time spent by SCAN in servicing requests belonging to the two classes within each 1000ms interval. Since SCAN does not distinguish between text and video requests, the time spent is servicing requests of the two classes fluctuates across intervals. Such fluctuations increase the variation in the response time of text requests and the probability of deadline violations for video requests.

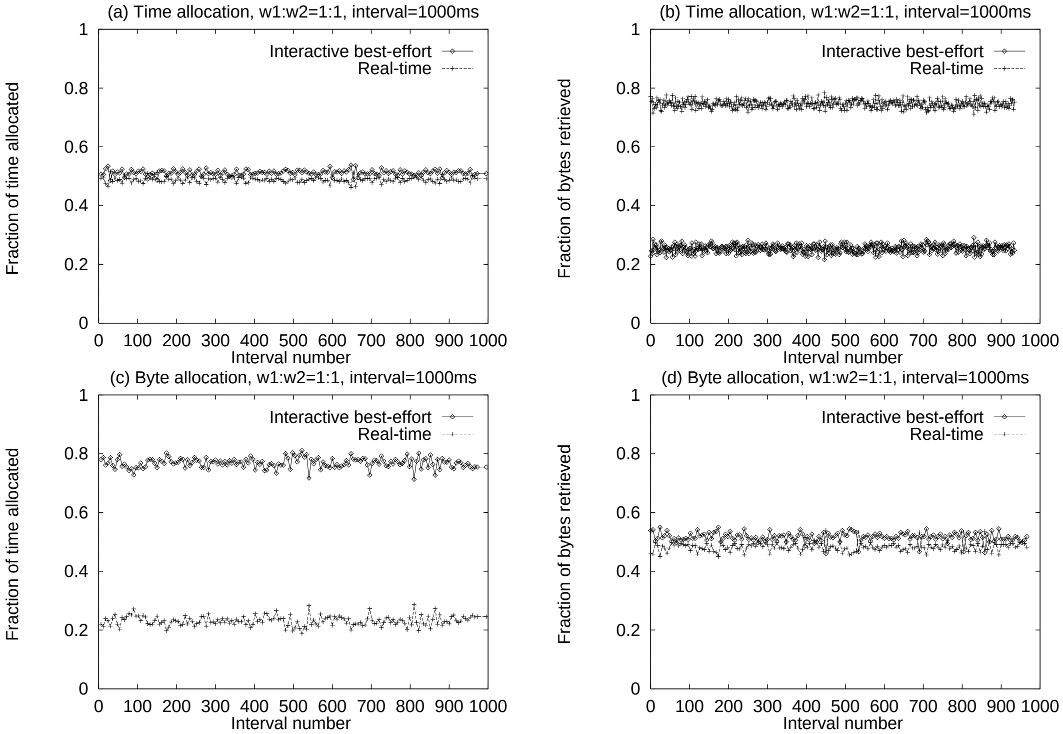


Figure 4.9: Comparison of time-allocation and byte-allocation strategies

4.3.3 Proportionate Time-allocation versus Proportionate Byte-allocation

In proportionate time-allocation, the class independent scheduler allocates the time spent in servicing requests among application classes in proportion to their weights. Since different requests incur different seek and rotational latency overheads, the number of bytes retrieved for classes i and j within an interval may not be in the ratio $\frac{w_i}{w_j}$, even though the time spent in retrieving this data is. The problem is further exacerbated if requests can have different sizes, since the transfer time of requests also differ in addition to the latency overhead. To demonstrate this, we configured Cello with the interactive best-effort and real-time classes, assigned them equal weights, and measured the time allocated and the number of bytes retrieved for each class within an interval. The time-allocation strategy ensures that both the interactive best-effort and real-time classes receive equal durations within each interval. However, the number of bytes retrieved differs significantly for the two classes (see Figures 4.9(a) and (b)). In contrast, the byte-allocation strategy ensures that each class receives equal byte-allocation within each interval, but spends significantly different durations in retrieving this data. (see Figures 4.9(c) and (d)).

Both time-allocation and byte-allocation methods have their advantages and disadvantages. The byte-allocation method provides a direct correlation between the weights assigned to classes and the amount of data retrieved in an interval, making the task of assigning weights simple. However, a limitation of the byte allocation strategy is that each request is charged the same overhead $\mathcal{O}$ per byte, causing requests for large blocks to incur a larger seek and rotational latency overhead than smaller blocks. In reality, the latency overhead depends on the relative positions of requests and is independent of the request size. This penalizes requests for large blocks and benefits requests for small blocks. The time allocation strategy does not suffer from these limitations, since each class is charged the exact

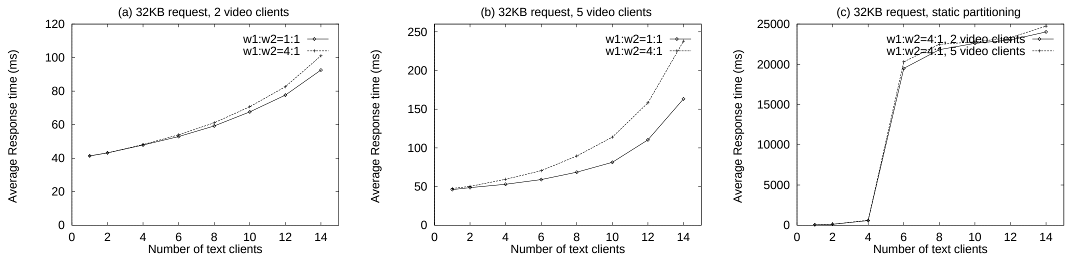


Figure 4.10: Effect of reassigning idle bandwidth on the response time

service times of its requests. Finally, since admission control algorithms for real-time requests (e.g., video) assume a fixed duration allocation for servicing requests, the time-allocation strategy is more suitable for such environments. The byte-allocation strategy is more suitable for environments in which the file server employs a single block size for all files.

4.3.4 Effect of reassigning idle bandwidth

Cello allocates bandwidth unused by a class to service pending requests from other classes. To demonstrate the benefits of reassigning idle bandwidth, we simulated two scenarios. In the first scenario, we assigned equal weights ($w_1 = w_2 = 1$) to the interactive best-effort and soft real-time classes; and in the second scenario, we reduced the allocation of the best-effort class to 20% ($w_1 : w_2 = 4 : 1$). For both scenarios, we measured the response time of text requests. Figures 4.10(a) and (b) show the response time of text requests for two different background video loads. Although the allocation of the best-effort class is only 20% of the total allocation in the second scenario, at light video loads, Cello reassigns bandwidth unused by the real-time class to the best-effort class, thereby providing response times comparable to the first scenario. As the background load increases, the unused allocation of the real-time class decreases, causing the response time of text requests to increase. Finally, Figure 4.10(c) shows the response time of text requests when static partitioning of disk bandwidth is employed. The figure shows that the response times of requests are larger by two orders of magnitudes as compared to Cello. This demonstrates that simple partitioning schemes are ineffective at efficiently utilizing disk bandwidth.

4.3.5 Overheads of Cello

Cello considers the requirements of requests as well as their relative positions on disk while making scheduling decisions. This causes some requests to be serviced out of scan order, increasing the seek and rotational latency overhead incurred by Cello. Figure 4.11 compares the time for which the disk was busy within an interval for a particular workload when Cello and SCAN are used. Although the total time to service a given set of requests is larger if Cello is used, the increase in service time is very small ($< 2\%$). Thus, the loss in disk throughput in Cello is negligible.

We have implemented the Cello disk scheduling framework in a prototype of Symphony. We conducted some preliminary experiments to measure the overheads of making scheduling decisions in Cello and SCAN. Our measurements were made on a 233MHz Intel Pentium II machine running Solaris 2.5. We ran the disk at 90% utilization, resulting in queue lengths of up to 150 requests. Table 4.2 compares the overheads of determining the insert position

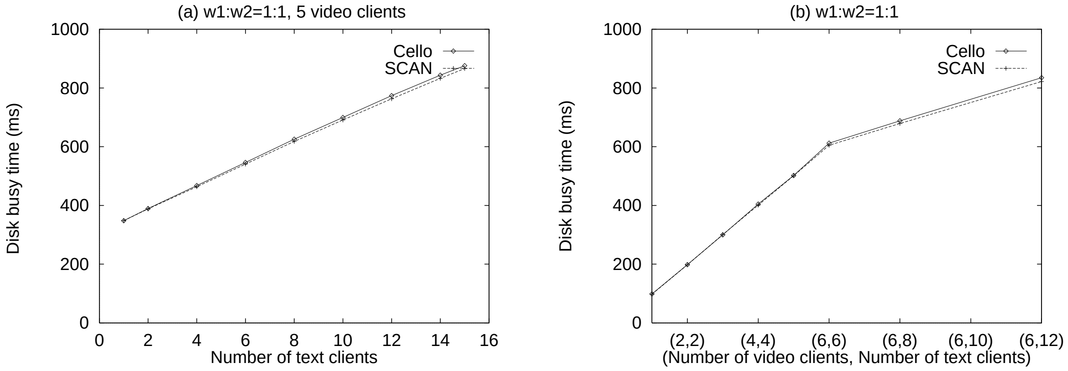


Figure 4.11: Overhead of Cello

	SCAN		Cello	
	Average	Worst-case	Average	Worst-case
Det. insert posn.	9 μs	0.5 ms	65 μs	4.6 ms
Total insert time	129 μs	7.4 ms	180 μs	12 ms

Table 4.2: Implementation Overheads

and the total time for inserting a request into the scheduled queue. The complexity of determining the insert position is $O(\log R)$ in SCAN (since binary search can be used to determine the insert position), while that for Cello is $O(R)$, where R is the number of requests in the scheduled queue. Hence, Cello incurs higher overhead than SCAN. However, the average overhead of inserting a request is small and results in a CPU load of less than 1% for requests rates of up to 50 requests/s. This demonstrates that sophisticated scheduling algorithms such as Cello are feasible in practice.

4.4 Related Work

Recently several research efforts have developed disk scheduling techniques for heterogeneous workloads. The Atropos scheduler in the Nemesis operating system schedules disk requests by allocating a certain fraction of the disk bandwidth to each application [7]. The Real-Time file system [59] employs a combination of admission control and slack stealing techniques to service real-time and best-effort requests. Nerjes et. al. propose priority-based techniques to schedule continuous media and textual requests; the number of requests from a class that are serviced within each interval is limited by a threshold value [62]. Wijayaratne et. al. propose a technique that partitions real-time requests into a number of sub-groups based on their disk locations. Interactive requests are inserted into the closest sub-group, and sub-groups with outstanding interactive requests are given priority by the scheduler [93]. The MARS multimedia server employs a two level scheduler that orders requests in pending queues using class-specific schedulers and employs a deficit round robin algorithm to control the number of requests that are inserted from each pending queue into the scheduled queue [11]. Class-specific schedulers are used only to determine the ordering within the corresponding pending queue and do not determine the interleaving within the scheduled queue; the scheduled queue is always maintained in SCAN order. The key difference between these techniques and Cello is that they employ either techniques that allocate disk bandwidth to application classes or techniques that interleave requests from multiple classes, but not both. In contrast, Cello employs both class-independent and class-specific schedulers to allocate bandwidth to classes as well as determine an interleaving of requests that aligns the service provided with application needs.

4.5 Concluding Remarks

In this chapter, we articulated the disk scheduling requirements imposed by applications likely to be supported by integrated file systems. We then presented the Cello disk scheduling framework for meeting these requirements. Cello assigns weights to the application classes. It services requests from application classes by proceeding in terms of intervals and, during each interval, allocating disk bandwidth to application classes in proportion to their weights. Cello distributes the unused bandwidth allocation of an application class to schedule pending requests from another class. It interleaves requests from application classes such that the service provided is aligned with the application needs (e.g., video requests are serviced just prior to their deadlines, interactive best-effort requests are scheduled such that their response times are minimized, etc.). Finally, since a schedule that aligns the service with the application requirements may be different from one that minimizes disk latency overheads, Cello derives a schedule that balances these tradeoffs.

Cello efficiently performs these functions by employing a two-level disk scheduling architecture, consisting of a class-independent scheduler and a set of class-specific schedulers. The two levels of the framework allocate disk bandwidth to application classes at two time-scales: the class-independent scheduler governs the coarse-grain bandwidth allocation to application classes; while the class-specific schedulers control the fine-grain interleaving of requests from the application classes to align the service provided with the application requirements. The two levels of the architecture separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the coexistence of multiple class-specific schedulers.

Our experiments demonstrate that:

- Existing disk scheduling algorithms are inadequate for servicing application classes with diverse requirements.
- Cello yields substantial performance improvements over existing disk scheduling algorithms. For instance, with six text and video clients (i.e., a disk utilization of 60%), Cello yields a factor of 2.5 improvement in response time over SCAN, while continuing to meet the deadlines of all real-time requests.
- Cello is suitable for servicing application classes with diverse requirements since: (i) it aligns the service provided with the application requirements, (ii) it protects application classes from one another, (iii) it is work-conserving and can adapt to changes in work-load, (iv) it minimizes the seek time and rotational latency overhead incurred during access, and (v) it is computationally efficient.

Chapter 5

Striping Techniques

Stripes are in.

—The New York Times Fashion Review

Due to the large storage space and bandwidth requirements of data types such as continuous media, integrated file systems are founded on disk arrays. To efficiently utilize a disk array, the file system stripes each file across disks in the array. The performance of striped disk arrays is governed by two parameters: the *stripe unit size*, which denotes the maximum amount of logically contiguous data stored on a single disk; and the *degree of striping*, which refers to the number of disks across which a particular file is striped. Depending on the striping policy, successive blocks of a file can be stored on the same or different locations on consecutive disks.

Recently, techniques for determining the stripe unit size and the degree of striping for workloads consisting of textual and numeric data accesses have been proposed [17, 18, 51]. However, these techniques are not directly applicable to file servers storing continuous media due to the following fundamental characteristics:

- *Real-time requirements of continuous media*: Textual and numeric data accesses require good response times but no absolute performance guarantees. In contrast, due to its real-time nature, continuous media accesses require the file server to provide bounds on response times. Hence, a stripe unit size that minimizes the average response time is considered optimal for textual and numeric data [17], while a stripe unit size that minimizes the tail of the response time distribution (possibly at the expense of an increased average response time) is more desirable for continuous media data.

  This fundamental difference in the optimization criterion has a significant impact on the selection of stripe unit size. To illustrate, consider Figure 5.1(a), which depicts the histogram of the response time observed for two different stripe unit sizes (obtained using a workload of 60 video clients accessing an array of 16 Seagate Elite3 disks). It shows that stripe unit sizes of 32KB and 64KB yield average response times of 30ms and 32ms, respectively. The figure also shows that the histogram for the 32KB stripe unit size has a longer tail. If data accesses do not impose any real-time constraints, 32KB would be chosen as the appropriate stripe unit size. For accesses with real-time constraints, a stripe unit size of 64KB would be more desirable. As shown in Figure 5.1(b), the block size that minimizes the average response time continues to differ from one that minimizes the 90^{th} percentile of the response time (i.e., the tail of the histogram) over a wide range of client workloads.

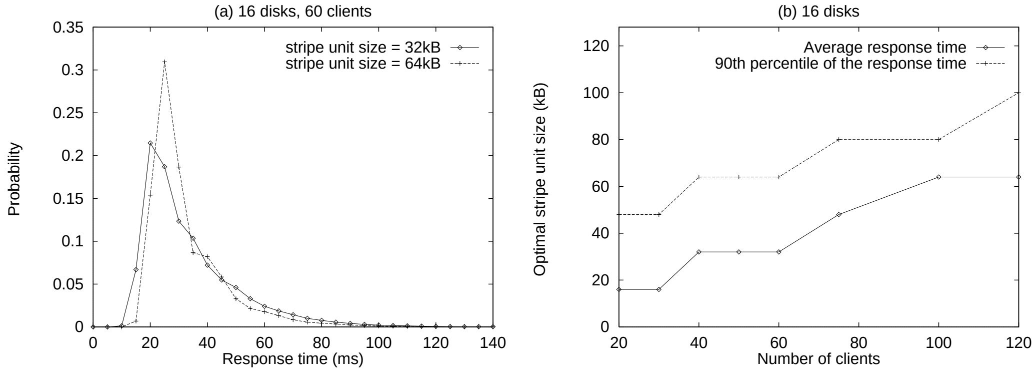


Figure 5.1: Effect of different metrics on the stripe unit size.

- *Periodic and sequential nature of continuous media:* In general, textual and numeric data accesses consist of aperiodic reads and writes, while continuous media workloads consist of reads and writes that are periodic and sequential. These differences in access characteristics not only affect the optimal stripe unit size and degree of striping, but also result in a fundamentally different mode of data access. Specifically, due to the sequential and periodic nature of data accesses, integrated file systems service continuous media requests by periodically accessing and transmitting data, without an explicit request from the client for each access. Such a *server-push* mode of retrieval is markedly different from the *client-pull* mode employed for servicing conventional applications (in which data is accessed by the server only in response to an explicit client request). Differences in the retrieval mode also affect the stripe unit size selection process.

- *Large data rate requirements of continuous media:* Typically textual files are a few kilobytes in size, whereas continuous media files are orders of magnitude larger (e.g., an one hour MPEG-1 video is over 1GB in size). Moreover, continuous media clients have large data rate requirements (MPEG-2 clients can have data rate requirements of 4MB/s). Hence, a large stripe unit size is more desirable for continuous media files so as to reduce disk seek and rotational latency overheads and improve server throughput, while a small stripe unit size is more suitable for textual files.

Due to these differences, novel techniques that optimize the performance of disk arrays storing continuous media data must be developed. Techniques for determining the optimal stripe unit size and the degree of striping for continuous media files constitute the subject matter of this chapter.

The rest of this chapter is organized as follows. In Section 5.1, we present techniques for determining the optimal stripe unit size for continuous media files. Section 5.2 describes techniques for determining the degree of striping. Section 5.3 presents related work, and finally, Section 5.4 summarizes our results.

5.1 Determining the Stripe Unit Size

Consider an integrated file system that interleaves continuous media files across disks by storing successive blocks of a file on consecutive disks in a round-robin manner. The unit of interleaving, referred to as a *media block* or a *stripe unit*, denotes the maximum amount of logically contiguous data stored on a single disk.[1] Due to the periodic nature of continuous media playback, the file server services multiple clients by proceeding in periodic *rounds*. During each round, the server retrieves a fixed number of *media units* (e.g., video frames or audio samples) for each client. To ensure continuous playback, the number of media units accessed for a client must be sufficient to sustain its playback rate, and the service time (i.e., the total time spent in retrieving media units during a round) must not exceed the duration of a round.

If each continuous media file is compressed using a variable bit rate (VBR) compression algorithm, then the sizes of successive media units within a file will vary. Although each client accesses a fixed number of media units in each round, due to variable media unit sizes, the number of blocks requested by the client can *vary from one round to another*. The server can service such clients either by retrieving a variable number of blocks across rounds, or by retrieving a fixed number of blocks across rounds and employing prefetching and buffering schemes to smooth out the variations. Depending on the amount of variation in the bit rate, the latter approach can substantially increase the initiation latency (since sufficient amount of data must be prefetched before the client can initiate playback). Our experiments with MPEG-1 video clients indicate that, for a round duration of 1s, accessing data at the average bit rate can cause the initiation latency to be more than 20s. Although latencies of tens of seconds are acceptable for video-on-demand environments, they are unsuitable for general purpose integrated file systems. In contrast, since no smoothing is performed when a variable number of blocks are accessed, the clients can initiate playback without any delay. However, accessing a variable number of blocks can cause load imbalances across disks in the array and can reduce the number of clients supported by the file server. A key challenge is to devise striping techniques that reduce such load imbalances and maximize the number of clients supported. In this chapter, we assume that the file server services clients by accessing a variable number of blocks across rounds, and determine the stripe unit size and the degree of striping that achieves the above objective.

Since the file server accesses a variable number of blocks per client, the set of disks accessed by different clients during a round are different, and hence, the total number of blocks accessed can vary from one disk to another. Since some disks are more heavily loaded than others, the service time of some of these disks may occasionally exceed the round duration, causing playback discontinuities at client sites. To minimize the frequency of such playback discontinuities, the server must minimize the service time of the most heavily loaded disk in the array. The service time of the most heavily loaded disk depends on the media block size. To observe this, consider a small media block size. Such a block size increases the number of blocks accessed from the array during a round, thereby distributing the load across disks and reducing the load imbalance. However, it also increases the overhead due to seek and rotational latency, thereby increasing the service time of the most heavily loaded disk. In contrast, a large block size reduces the overhead of seek and rotational latency, but increases the load imbalance, and hence, the service time of the most heavily loaded disk. The server must select a media block size that balances these tradeoffs and *minimizes the service time of the most heavily loaded disk in the array.*

In what follows, we present an analytical model that uses the characteristics of the workload and the configuration of the file server to predict the service time of the most heavily loaded disk in non-redundant and redundant disk arrays. By computing this service time over a range of block sizes, a media block size that minimizes it can be chosen.

[1]We shall use the terms media block and stripe unit interchangeably in this dissertation.

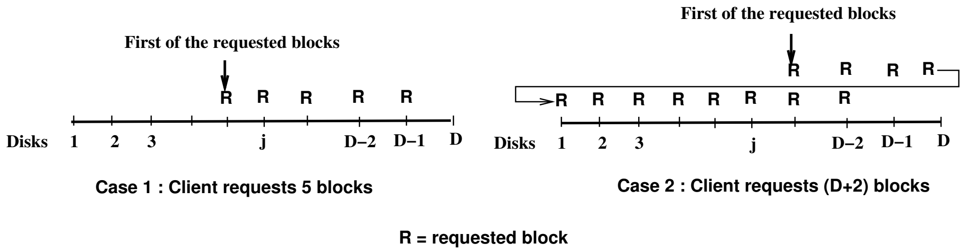


Figure 5.2: Different scenarios in which client i accesses a block from disk j.

5.1.1 Analytical Models for Determining the Load on the Array

5.1.1.1 A Model for Non-redundant Arrays

Consider an integrated file system that interleaves continuous media files across a disk array. Given the configuration of the server (e.g., number of disks, their physical characteristics, the round duration, etc.) and the client characteristics (e.g., number of clients, trace of the media unit sizes for each client, playback rate, etc.), the service time of the most heavily loaded disk in redundant and non-redundant disk arrays can be computed as follows:

1. Compute the distribution of the number of blocks accessed from a disk by each client during a round using a trace of media unit sizes.
2. Compute the distribution of the total number of blocks accessed from a disk by summing the number of blocks requested by each client from that disk.
3. Compute the distribution of the number of blocks accessed from the most heavily loaded disk.
4. Given the distribution of the number of blocks accessed from the most heavily loaded disk, compute the service time distribution for the disk using a disk model.

To derive the model for non-redundant arrays, consider a server that employs an array of D disks for interleaving files. Let n clients access the server, each retrieving a media stream,[2] and let B denote the media block size. Since the server accesses a fixed number of media units for each client during a round, the distribution of the number of blocks accessed by the client during a round can be determined from a trace of the media unit sizes. Let b_i^k, obtained from this distribution, denote the probability that client i accesses k blocks from the array in a round, and let p_{ij}^k denote the probability that client i accesses k blocks from disk j in a round. To compute p_{ij}^1, observe that client i will access exactly one block from disk j in a round if: (1) it requests m blocks ($1 \leq m \leq D$) from the array and the first of these blocks is stored either on disk j or any of the previous $m-1$ disks; or (2) it requests $D+m$ blocks ($1 \leq m < D$) from the array and the first of these block is stored any disk *other than* disk j or any of the previous $m-1$ disks. Figure 5.2 illustrates these cases. Due to the VBR nature of continuous media, the number of blocks accessed by a client varies from one round to another. Hence, after a small number of rounds, the first block is equally likely to be accessed from any of the disks in the array. Consequently,

$$p_{ij}^1 = \sum_{m=1}^{D} b_i^m \cdot \frac{m}{D} + \sum_{m=1}^{D-1} b_i^{D+m} \cdot \frac{D-m}{D} \tag{5.1}$$

[2]Since continuous media requests are dominated by read requests, we confine our focus to read requests.

Generalizing, client i will access k blocks ($k = 1, 2, 3...$) from disk j if: (1) it requests $(k-1) \cdot D + m$ blocks ($1 \leq m \leq D$) from the array and the first of these blocks is stored on disk j or any of the previous $m-1$ disks; or (2) it requests $k \cdot D + m$ blocks ($1 \leq m < D$) from the array and the first of these blocks is stored on any disk other than disk j or any of the previous $m-1$ disks. Hence,

$$p_{ij}^k = \sum_{m=1}^{D} b_i^{(k-1)\cdot D+m} \cdot \frac{m}{D} + \sum_{m=1}^{D-1} b^{k\cdot D+m} \cdot \frac{D-m}{D} \tag{5.2}$$

Lastly, the probability that client i does not access disk j is $p_{ij}^0 = 1 - \sum_{k=1}^{\infty} p_{ij}^k$.
Let $\mathcal{X}_{ij}$ be a random variable denoting the number of blocks accessed by client i from disk j during a round. Then,

$$P(\mathcal{X}_{ij} = k) = p_{ij}^k \tag{5.3}$$

Then, the total number of blocks accessed from disk j during a round, $\mathcal{N}_j$, can be computed as

$$\mathcal{N}_j = \sum_{i=1}^{n} \mathcal{X}_{ij} \tag{5.4}$$

Due to the VBR nature of continuous media, the number of blocks accessed by clients from the array are independent of each other. Thus, $\mathcal{X}_{1j}, \mathcal{X}_{2j}, ..., \mathcal{X}_{nj}$ are independent random variables, and hence, the distribution of $\mathcal{N}_j$ can be obtained by applying the the z-transform [3] to (5.4). That is,

$$Z(\mathcal{N}_j) = \prod_{i=1}^{n} Z(\mathcal{X}_{ij}) \tag{5.5}$$

where

$$Z(\mathcal{X}_{ij}) = p_{ij}^0 + z\ p_{ij}^1 + z^2\ p_{ij}^2 + z^3\ p_{ij}^3 + \cdots \tag{5.6}$$

Then, the number of blocks accessed from the most heavily loaded disk is given by

$$\mathcal{N}_{\max} = \max(\mathcal{N}_1, \mathcal{N}_2, \cdots, \mathcal{N}_D) \tag{5.7}$$

Due to the round robin nature of file placement, the number of blocks accessed from a disk is not independent of the load on its neighboring disks. Since the precise dependence of these random variables on each other is difficult to characterize, and since the maximum of D dependent random variables is difficult to compute, as an approximation we assume that $\mathcal{N}_j$s are independent of each other. Later in this section, we demonstrate that this approximation does not cause any inaccuracies in the predictions of the model. Then, the distribution of $\mathcal{N}_{\max}$ can be computed as

$$F_{\mathcal{N}_{\max}}(x) = F_{\mathcal{N}_1}(x) \cdot F_{\mathcal{N}_2}(x) \cdots F_{\mathcal{N}_D}(x) \tag{5.8}$$

where $F_{\mathcal{N}_j}$ is the cumulative probability distribution function of the random variable $\mathcal{N}_j$ [66].

[3] The z-transform of a random variable $\mathcal{U}$ is the polynomial $Z(\mathcal{U}) = a_0 + za_1 + z^2a_2 + \cdots$ where the coefficient of the i^{th} term in the polynomial represents the probability that the random variable equals i. That is, $P(\mathcal{U} = i) = a_i$. If $\mathcal{U}_1, \mathcal{U}_2, ..., \mathcal{U}_n$ are n independent random variables, and $\mathcal{Y} = \sum_{i=1}^{n} \mathcal{U}_i$, then $Z(\mathcal{Y}) = \prod_{i=1}^{n} Z(\mathcal{U}_i)$. The distribution of $\mathcal{Y}$ can then be computed using a polynomial multiplication of the z-transforms of $\mathcal{U}_1, \mathcal{U}_2, \cdots, \mathcal{U}_n$ [66].

Having determined the distribution of the number of blocks accessed from the most heavily loaded disk, the service time of the disk can then be computed by using a disk model. We use one such model that has been proposed in the literature [51, 92]. The service time to access $\mathcal{N}_{\max}$ blocks of size B as predicted by the disk model is:

$$\tau_{\max} = \mathcal{N}_{\max} \cdot (t_s + t_r) + \mathcal{N}_{\max} \cdot B \cdot t_t \tag{5.9}$$

where t_s and t_r denote the seek time and rotational latency incurred while accessing a block from disk and t_t denotes the transfer time for a unit amount of data. Assuming that $\mathcal{N}_{\max}$ blocks are uniformly distributed across the $\mathcal{C}$ cylinders of a disk, the distance between two consecutive blocks is $\lfloor \frac{\mathcal{C}}{\mathcal{N}_{\max}+1} \rfloor$ cylinders. Hence, we define $t_s = t_{seek}\left(\lfloor \frac{\mathcal{C}}{\mathcal{N}_{\max}+1} \rfloor\right)$, where $t_{seek}(x)$ is the time to move the disk head across x consecutive cylinders and is computed as:

$$t_{seek}(x) = \begin{cases} 0 & \text{if } x = 0 \\ a\sqrt{x-1} + b(x-1) + c & \text{otherwise} \end{cases}$$

where a, b, and c are constants (determined using physical characteristics of a disk) [51]. The rotational latency, t_r, is defined to be half of the maximum rotational latency

Thus, given the file server configuration and the workload characteristics, the model computes the service time distribution of the most heavily loaded disk for a particular block size. Moreover, the model also yields the distribution of the number of blocks accessed from a disk with average load (i.e., $\mathcal{N}_j$). The service time of such a disk can then be computed using the disk model.

5.1.1.2 A Model for Redundant Arrays

Since disk arrays are highly susceptible to disk failures, integrated file systems employ redundancies in data storage to guarantee high availability of data. Most redundant arrays are based on the *Redundant Array of Independent Disks (RAID)* architecture [19, 67]. RAID arrays compute redundant blocks (referred to as *parity*) by taking an exclusive-or operation over data blocks stored on $G - 1$ disks, where $G > 2$, and store it on another disk. The parity block together with all the data blocks over which parity is computed is referred to as a *parity group*. In the presence of a disk failure (also referred to as the degraded mode), the server reconstructs a block stored on the failed disk by accessing the parity block and data blocks of the parity group stored on surviving disks. A commonly used RAID architecture is RAID-5 which uses block-interleaved parity and uniformly distributes parity blocks across disks in the array. The multiple RAID-5 architecture is an extension of the RAID-5 array in which the array is partitioned into clusters of disks, with each cluster independently computing parity information [19]. In the rest of this section, we assume a multiple RAID-5 architecture for our model. However, the basic approach used in our model is applicable to other RAID architectures as well.

Consider a integrated file system servicing n clients from a RAID-5 array consisting of D disks. Let G denote the parity group size, where $G \leq D$. Then the array contains $P = D/G$ clusters. Let us assume that the server computes parity blocks over a sequence of successive blocks from the file (i.e., all data blocks of a parity group are consecutive blocks of the same file). Consequently, the server stores successive blocks of a file on disks containing data blocks of the parity group and skips over disks storing the parity blocks. Since each of the P clusters contains a disk storing a parity block, a request for more than $D - P$ consecutive blocks causes a disk to be reaccessed.

Fault-free Case

To compute the service time of the most heavily loaded disk in the fault-free mode, let b_i^k denote the probability that client i accesses k blocks from the array during a round, and let p_{ij}^k denote the probability that client i accesses k blocks from disk j during a round. To compute p_{ij}^1, note that client i will access disk j only if disk j stores a data block (i.e., does not store a parity block). Moreover, client i will access a block from disk j if: (1) it requests m blocks ($1 \leq m \leq D-P$) from the array and the first of these blocks is stored on disk j or any of the previous $m-1$ disks storing data blocks; or (2) it requests $D-P+m$ blocks ($1 \leq m < D-P$) from the array and the first of these blocks is stored on any disk storing data blocks other than disk j or any of the previous $m-1$ disks. Since parity blocks are uniformly distributed across disks, one out of every G blocks stored on a disk is a parity block. Hence, the probability that disk j stores a data block is $(1-1/G)$. Due to the VBR nature of continuous media, the first block is equally likely to be accessed from any of the $D-P$ disks storing data blocks. Hence, we get

$$p_{ij}^1 = (1-\frac{1}{G}) \cdot \left(\sum_{m=1}^{D-P} b_i^m \cdot \frac{m}{D-P} + \sum_{m=1}^{D-P-1} b_i^{(D-P)+m} \cdot \frac{D-P-m}{D-P} \right) \tag{5.10}$$

Generalizing, the probability that client i accesses k blocks from disk j is

$$p_{ij}^k = (1-\frac{1}{G}) \cdot \left(\sum_{m=1}^{D-P} b_i^{(k-1)\cdot(D-P)+m} \cdot \frac{m}{D-P} + \sum_{m=1}^{D-P-1} b_i^{k\cdot(D-P)+m} \cdot \frac{D-P-m}{D-P} \right) \tag{5.11}$$

where $k = 1, 2, 3, \ldots$. Since $P = \frac{D}{G}$, $(1-\frac{1}{G})$ can be rewritten as $\frac{D-P}{D}$. Substituting this value in the above equation and simplifying, we get

$$p_{ij}^k = \sum_{m=1}^{D-P} b_i^{(k-1)\cdot(D-P)+m} \cdot \frac{m}{D} + \sum_{m=1}^{D-P-1} b_i^{k\cdot(D-P)+m} \cdot \frac{D-P-m}{D} \quad (k = 1, 2, 3, \ldots) \tag{5.12}$$

Let $\mathcal{X}_{ij}$ be the random variable representing the number of blocks accessed by client i from disk j during a round. Then $P(\mathcal{X}_{ij} = k) = p_{ij}^k$. Using this distribution of $\mathcal{X}_{ij}$, the distributions of the number of blocks accessed and the service time of the most heavily loaded disk in the fault-free state can be derived using the method presented in Section 5.1.1.1.

Failure Case

To compute the service time of the most heavily loaded disk in degraded mode, assume that disk f in the array experiences a failure, where $1 \leq f \leq D$. Since each cluster independently computes parity, disks that do not belong to the cluster containing disk f are unaffected by this failure, and hence, for these disks, the number of blocks accessed in a round is the same as that in the fault-free state. All disks belonging to the cluster containing disk f, however, will experience an increase in load whenever a client accesses a block from disk f. To compute the number of blocks accessed by client i from disk j belonging to the cluster containing disk f, let δ denote the number of disks storing data blocks contained between disks j and f (including disk j), and let Δ denote the number of disks storing data blocks not contained between disks j and f. Observe that, if no parity block is stored on a disk between disks j and f, then $\delta = \mid j - f \mid$. Otherwise $\delta = \mid j - f \mid -1$. In either case, $\Delta = D - P - \delta$.

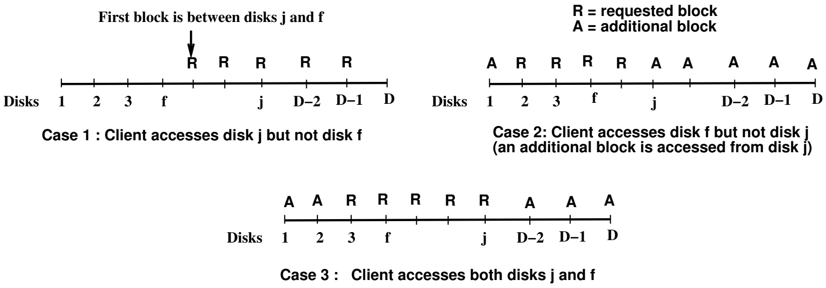


Figure 5.3: Different scenarios in which client i accesses a block from disk j in degraded mode.

To compute p_{ij}^1, note that client i will access exactly one block from disk j if it requests m blocks from the array and one of the following three conditions hold: (1) a block is requested from disk j but not from disk f, or (2) a block is requested from disk f but not from disk j (and hence, a block must be accessed from disk j to reconstruct the block on disk f), or (3) a block is requested from both disks j and f and both blocks belong to the same parity group (and hence, no additional block needs to be accessed from disk j). Figure 5.3 illustrates these cases for an array with $G = D$.

To compute the probability that client i accesses disk j but not disk f, let us first consider the case when $f < j$. Client i will access disk j only if disk j stores a data block of the parity group. Moreover, client i will access a block from disk j but not disk f if: (1) it requests m blocks ($1 \leq m \leq \delta$) from the array and the first of these blocks is stored on disk j or any of the previous $m-1$ disks; or (2) it requests $\delta + m$ blocks ($1 \leq m \leq \Delta - \delta$) from the array and the first of these blocks is stored on disk j or any of the previous $\delta - 1$ disks; or (3) it requests $\Delta + m$ blocks ($1 \leq m \leq \delta - 1$) from the array and the first of these blocks is stored on disk j or any of the previous $\delta - m - 1$ disks. A similar argument holds for the case when $f > j$, except that we must consider the last block accessed by the client instead of the first block. Since the first (last) block is equally likely to be stored on any of the $D - P$ disks storing data blocks, and since the probability that disk j stores a data block is $(1 - \frac{1}{G})$, we get

$$p' = (1 - \frac{1}{G}) \cdot (\sum_{m=1}^{\delta} b_i^m \cdot \frac{m}{D-P} + \sum_{m=1}^{\Delta-\delta} b_i^{\delta+m} \cdot \frac{\delta}{D-P} + \sum_{m=1}^{\delta} b_i^{\Delta+m} \cdot \frac{\delta - m}{D-P}) \tag{5.13}$$

Substituting $\frac{D-P}{D}$ for $(1 - \frac{1}{G})$ in the above equation and simplifying, we get

$$p' = \sum_{m=1}^{\delta} b_i^m \cdot \frac{m}{D} + \sum_{m=1}^{\Delta-\delta} b_i^{\delta+m} \cdot \frac{\delta}{D} + \sum_{m=1}^{\delta} b_i^{\Delta+m} \cdot \frac{\delta - m}{D} \tag{5.14}$$

By symmetry, the probability that client i accesses disk f but not disk j is the same as the probability that it accesses disk j but not disk f.

To compute the probability that client i accesses a block from both disks j and f, observe that the client must request at least δ blocks from the array (see Figure 5.3). Moreover, to be able to access disk j and f both disks j and f must store data blocks. Hence, the client accesses blocks belonging to the same parity group from disks j and f if (1) it requests $(m + \delta)$ blocks from the array, $(0 \leq m \leq \Delta)$ and the first of these blocks is stored on a disk not contained between disks j and f; or (2) it accesses $(D - P + m)$ blocks and the first of these blocks is stored on a disk such that only one block is accessed from disks j and f. Since two out of every G parity groups will store a parity block on disks j or f, the probability that neither disk j nor disk f stores a parity block is $(1 - \frac{2}{G})$. Hence, the probability of accessing blocks belonging to the same parity group from disks f and j is

$$p'' = (1 - \frac{2}{G}) \cdot \left(\sum_{m=0}^{\Delta} b_i^{m+\delta} \cdot \frac{m+1}{D-P} + \sum_{m=1}^{\Delta} b_i^{D-P+m} \cdot \frac{\Delta - m + 1}{D-P} \right) \quad (5.15)$$

Hence, summing the probability of the three cases, we get $p_{ij}^1(\delta, \Delta) = 2 \cdot p' + p''$, That is,

$$\begin{aligned} p_{ij}^1(\delta, \Delta) &= 2 \cdot \left(\sum_{m=1}^{\delta} b_i^m \cdot \frac{m}{D} + \sum_{m=1}^{\Delta-\delta} b_i^{\delta+m} \cdot \frac{\delta}{D} + \sum_{m=1}^{\delta} b_i^{\Delta+m} \cdot \frac{\delta - m}{D} \right) + \\ & (1 - \frac{2}{G}) \cdot \left(\sum_{m=0}^{\Delta} b_i^{m+\delta} \cdot \frac{m+1}{D-P} + \sum_{m=1}^{\Delta} b_i^{D-P+m} \cdot \frac{\Delta - m + 1}{D-P} \right) \end{aligned} \quad (5.16)$$

The value of p_{ij}^1 computed in the above equation is a function of parameters δ and Δ. Depending on whether or not a parity block is stored on a disk between disks j and f, we have two cases. If a parity block is stored on a disk between disks j and f, then we get $\delta_1 =\mid j - f \mid -1$ and $\Delta_1 = D - P - \delta_1$. Since parity blocks are uniformly distributed across disks in the array, and the probability that of this case is is $\frac{\delta_1}{G}$. If no parity block is stored between disks j and f, then we get $\delta_2 =\mid j - f \mid$ and $\Delta_2 = D - P - \delta_2$, and the probability of this case is $(1 - \frac{\delta_1}{G})$.

Hence, the overall probability that client i accesses one block from disk j is

$$p_{ij}^1 = \frac{\delta_1}{G} \cdot p_{ij}^1(\delta_1, \Delta_1) + (1 - \frac{\delta_1}{G}) \cdot p_{ij}^1(\delta_2, \Delta_2) \quad (5.17)$$

Generalizing, the probability that client i accesses k blocks from disk j is

$$p_{ij}^k = \frac{\delta_1}{G} \cdot p_{ij}^k(\delta_1, \Delta_1) + (1 - \frac{\delta_1}{G}) \cdot p_{ij}^k(\delta_2, \Delta_2) \quad (5.18)$$

where

$$\begin{aligned} p_{ij}^k(\delta, \Delta) &= 2 \cdot \left(\sum_{m=1}^{\delta} b_i^{m+\alpha} \cdot \frac{m}{D} + \sum_{m=1}^{\Delta-\delta} b_i^{\delta+m+\alpha} \cdot \frac{\delta}{D} + \sum_{m=1}^{\delta} b_i^{\Delta+m+\alpha} \cdot \frac{\delta - m}{D} \right) + \\ & (1 - \frac{2}{G}) \cdot \left(\sum_{m=0}^{\Delta} b_i^{m+\delta+\alpha} \cdot \frac{m+1}{D-P} + \sum_{m=1}^{\Delta} b_i^{D-P+m+\alpha} \cdot \frac{\Delta - m + 1}{D-P} \right) \end{aligned} \quad (5.19)$$

and $\alpha = (k-1) \cdot (D-P)$. Let $\mathcal{X}_{ij}$ be the random variable representing the number of blocks accessed by client i from disk j during a round. Then $P(\mathcal{X}_{ij} = k) = p_{ij}^k$. Then, using this distribution of $\mathcal{X}_{ij}$, the distribution of the number of blocks accessed and the service time of the most heavily loaded disk in the degraded mode can be derived in a manner similar to that in Section 5.1.1.1.

5.1.2 Validation of the Models

To validate our models, we have built an event-based, trace-driven disk array simulator called *diskSim*. We digitized a number of traces and used these traces to run simulations over a wide range of system parameters (e.g., different number of clients, different number of disks, different round durations, etc.). The characteristics of the traces are

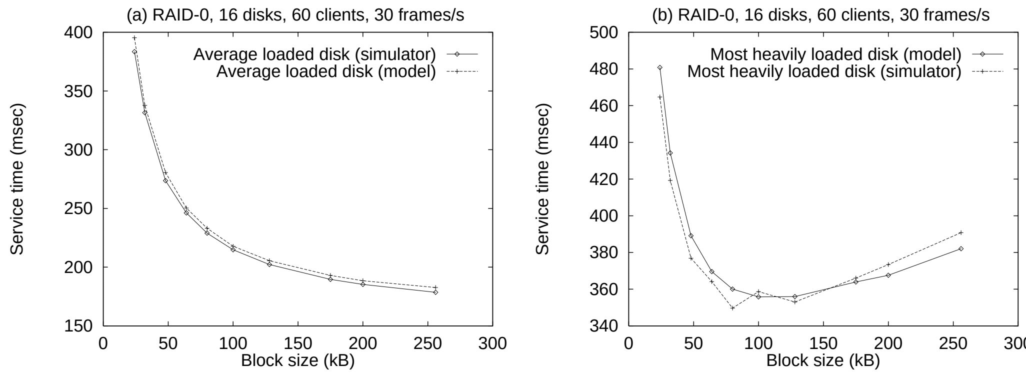


Figure 5.4: Variation in the service time of the average loaded disk and the most heavily loaded disk.

listed in Table 5.1. For each combination of parameters, we conducted multiple simulation runs and computed the 95% confidence intervals of the expected number of blocks accessed and the expected service time of the most heavily loaded disk. To validate the model for non-redundant arrays, we computed the expected number of blocks accessed and the expected service time of the most heavily loaded disk for each workload. The values predicted by the model were found to be within the 95% confidence intervals obtained from simulations. Figures 5.4(a) and (b) plot these values for one such workload. Similar results were obtained for the model for redundant arrays. Thus, the simulation results validate the predictions made by our analytical models over a large parameter space.

The service time graphs of the average loaded disk and the most heavily loaded disk in Figure 5.4 lead us to the following observations:

- As shown in Figure 5.4(a), the service time of the average loaded disk decreases monotonically with increasing block size. This is because increasing the block size decreases the number of blocks accessed from the disk, thereby reducing disk seek and rotational latency overheads.

- The service time of the most heavily loaded disk, on the other hand, decreases initially and then starts increasing with increase in block size (see Figure 5.4(b)). To explain this behavior, let us first introduce some terminology. Let $\widehat{\mathcal{N}}_{\max}$ and $\widehat{\tau}_{\max}$, respectively, denote the expected number of blocks accessed from the most heavily loaded disk and the expected service time of the most heavily loaded disk during a round, and let $\widehat{\tau}_{\text{avg}}$ denote the expected service time of the average loaded disk. Then, the imbalance in the service times of the most heavily loaded disk and the average loaded disk $\mathcal{I}_s$ (referred to as the load imbalance) is defined as

$$\begin{aligned}\mathcal{I}_s &= \frac{\widehat{\tau}_{\max} - \widehat{\tau}_{\text{avg}}}{\widehat{\tau}_{\max}} \\ &= 1 - \frac{\widehat{\tau}_{\text{avg}}}{\widehat{\tau}_{\max}} \qquad (5.20)\end{aligned}$$

From (5.9), the portion of the service time spent in disk seek and rotational latency is $\widehat{\mathcal{N}}_{\max} \cdot (t_s + t_r) = \widehat{\tau}_{\max} - \widehat{\mathcal{N}}_{\max} \cdot B \cdot t_t$. Hence, the overhead due to seek and rotational latency $\mathcal{O}$ can be defined as:

$$\begin{aligned}\mathcal{O} &= \frac{\widehat{\tau}_{\max} - \widehat{\mathcal{N}}_{\max} \cdot B \cdot t_t}{\widehat{\tau}_{\max}} \\ &= 1 - \frac{\widehat{\mathcal{N}}_{\max} \cdot B \cdot t_t}{\widehat{\tau}_{\max}} \qquad (5.21)\end{aligned}$$

Table 5.1: Characteristics of Video Traces

MPEG File	Encoding Pattern	Length (frames)	Frame rate	Bit rate Mb/s
Frasier	$I(BBP)^3BB$	5960	30	1.49
Newscast	$I(BBP)^3BB$	9000	30	2.33
Flintstones	$I(BBP)^3BB$	9000	30	1.67

Assuming a fixed server configuration and workload characteristics, increasing the block size decreases the number of blocks accessed from the array. The smaller the number of blocks being accessed, the smaller is the probability of achieving equitable distribution of load across disks (since the array becomes sparsely loaded). Hence, increasing block size yields an increase in the load imbalance $\mathcal{I}_s$. On the other hand, increasing the block size causes the seek and rotational latency overhead to decrease. Figure 5.5 shows these variations in $\mathcal{I}_s$ and $\mathcal{O}$.

For each media block size, the service time of the most heavily loaded disk is governed by the relative values of $\mathcal{I}_s$ and $\mathcal{O}$. As shown in Figure 5.5, at small block sizes, the latency overhead dominates, and hence the service time decreases with increase in block size. At large block sizes, the load imbalance dominates the latency overhead, and causes the service time to increase with increase in block size. Consequently, the service time of the most heavily loaded disk decreases initially and then starts increasing with increase in block size.

From the above analysis, we conclude that minimizing the service time of the average loaded disk requires the server to choose a block size that is as large as possible. In contrast, minimizing the service time of the most heavily loaded disk requires the server to choose a block size that minimizes the combined effects of $\mathcal{I}_s$ and $\mathcal{O}$. To maximize the number of clients supported for best-effort workloads, the server must minimize the service time of the average loaded disk, while for continuous media workloads, minimizing the service time of the most heavily loaded disk is more desirable. Hence, the optimal block size obtained for the two environments can differ significantly.

The precise value of the optimal block size for a continuous media workload depends on the quality of service requirements of clients and the values of various system parameters (such as the number of clients, their playback rate, the number of disks, etc.). In what follows, we examine the effect of these factors on the optimal block size. For each parameter, we also compute the range of block sizes that yields a service time within $x\%$ of the minimum. The upper and lower bounds of this set of block sizes define the $x\%$ *optimal envelope* for the workload [17, 92]. By choosing a block size that is contained within the $x\%$ optimal envelope of all values of the parameter, the server can ensure performance that is within $x\%$ of the optimal regardless of the workload.

5.1.3 Factors Affecting the Optimal Block Size

5.1.3.1 Effect of Quality of Service

Observe that the model yields a distribution of the service time of the most heavily loaded disk in the array. To determine the optimal block size, the server must first choose a particular percentile of the service time as the metric and then compute the block size that minimizes that percentile. For instance, the server can choose the the expected value of the service time (which, in our experiments, approximately corresponds to the 70^{th} percentile of

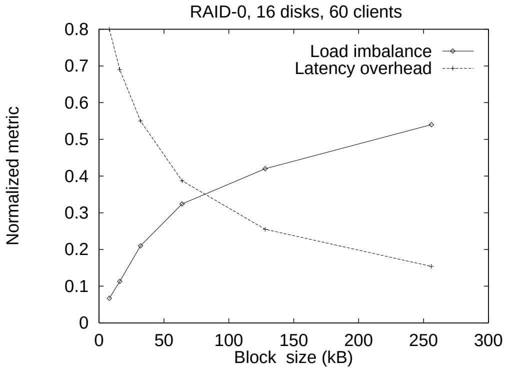


Figure 5.5: Variation in the load imbalance and the latency overhead.

the service time distribution) to determine the block size. In such a scenario, there is a 30% chance that the actual value of the service time during a round will exceed its expected value. If clients have stringent quality of service (QoS) requirements (i.e., they can tolerate only rare discontinuities in playback), then the server must choose higher percentiles of the service time to provide the desired performance guarantees. For example, by choosing the 95^{th} percentile of service time distribution of the most heavily loaded disk, the server can ensure that the service time does not exceed its estimated value in more than 5% of the rounds. Since different percentiles of the service time yield different optimal block sizes (see Figure 5.6(a)), the server must carefully choose an appropriate percentile of the service time as the metric based on the QoS requirements of clients.

Figure 5.6(b) shows the variation in optimal block size and the 5% optimal envelope for different percentiles of the service time. Larger percentiles of the service time correspond to more stringent QoS requirements. To provide stringent QoS, the server must minimize the variation in service times of the most heavily loaded disk across rounds. This can be achieved by selecting a block size which reduces the load imbalance. Since the load imbalance decreases with decrease in the block size (Figure 5.5), a small block size yields better performance for more stringent QoS requirements. Hence, the optimal block size and the 5% optimal envelope decrease with increase in percentile of the service time.

Observe from Figure 5.6(a) that the service time of the most heavily loaded disk increases slowly for block sizes larger than the optimal block size. This might lead us to believe that choosing a block size that is larger than the optimal will yield near optimal performance, while reducing disk latency overheads. However, Figure 5.6(b) demonstrates that choosing the largest possible block size contained in the optimal envelope for a particular QoS degrades performance for more stringent QoS. For instance, choosing the upper 5% optimal envelope of the 70^{th} percentile (i.e., 256KB) as the block size will cause a loss in performance for the 95^{th} percentile (since 256KB is not contained in the 5% optimal envelope of the 95^{th} percentile). This argument also shows that ad-hoc techniques that select a large block size (e.g., selecting the track size as the block size) can significantly affect the server performance, and hence, the number of clients supported. To achieve good performance over a range of QoS requirements, a block size that is contained within the $x\%$ optimal envelope of a wide range of percentiles must be chosen.

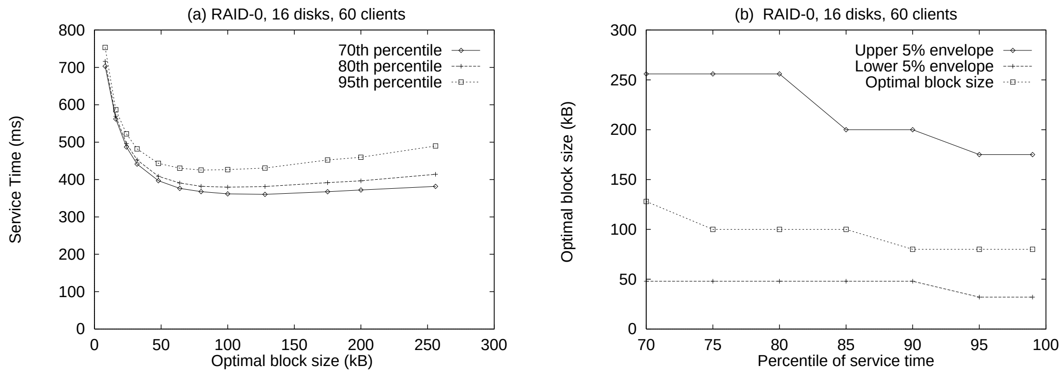


Figure 5.6: Effect of Quality of Service

5.1.3.2 Effect of system parameters

The model can also be used to study the effect of various system parameters on the optimal block size. Since the service time of the most heavily loaded disk is minimized when the combined effects of $\mathcal{I}_s$ and $\mathcal{O}$ are minimized, the effect of varying a system parameter on the optimal block size can be analyzed by studying its effect on $\mathcal{I}_s$ and $\mathcal{O}$. We can intuitively understand the effect of a parameter on the optimal block size by assuming that the point of intersection of $\mathcal{I}_s$ and $\mathcal{O}$ governs the minimum of the service time curve. Then, if a change in the value of the system parameter increases the number of blocks accessed from the array, it increases the probability of achieving equitable load distribution across disks, and hence, reduces $\mathcal{I}_s$. Such a reduction causes the $\mathcal{I}_s$ curve to shift downward. This shifts the point of intersection of $\mathcal{I}_s$ and $\mathcal{O}$ (and hence, the minima of the service time curve) to the right, thereby increasing the optimal block size. On the other hand, if a change in the value of the parameter causes a decrease in the number of blocks per disk, then the load imbalance increases. Such an increase causes the point of intersection of the $\mathcal{I}_s$ and $\mathcal{O}$ curves to shift to the left, thereby reducing the optimal block size. To illustrate, consider the effect of variation in the number of clients on the optimal block size. For a fixed server configuration, an increase in the number of clients increases the number of blocks accessed from the disk array, and thereby increases the probability of achieving equitable distribution of load across disks. This reduces the load imbalance $\mathcal{I}_s$, causing the $\mathcal{I}_s$ curve to shift downwards. In contrast, the latency overhead curve, which is governed mostly by the physical characteristics of disks, shifts only marginally. This shifts the point of intersection of $\mathcal{I}_s$ and $\mathcal{O}$ curves to the right (see Figure 5.7(a)). Hence, the optimal media block size increases with an increase in the number of clients accessing the server (see Figure 5.7(b)). The 5% optimal envelope also increases with increase in number of clients for similar reasons.

We have determined the effect of various system parameters, such as the number of disks, their physical characteristics, the playback rate of clients, the round duration, etc., on the optimal block size. The effect of all of these parameters on the optimal block size can be explained using arguments similar to those presented above. In what follows, we discuss our results in detail (Table 5.2 summarizes these results).

Number of Disks

For a fixed number of clients, increasing the number of disks in the system decreases the number of blocks accessed per disk. This decreases the probability of achieving equitable distribution of load across disks, and hence, increases the load imbalance $\mathcal{I}_s$. An increase in $\mathcal{I}_s$ causes the $\mathcal{I}_s$ curve to shift upwards and the point of intersection of $\mathcal{I}_s$ and

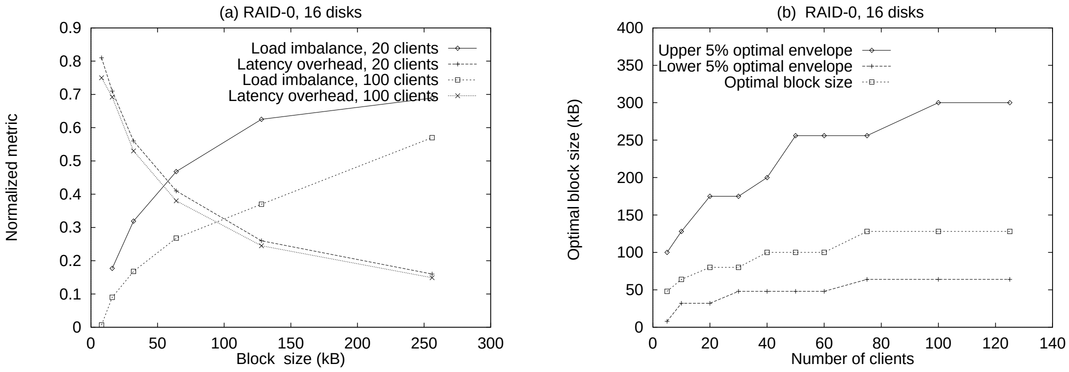


Figure 5.7: Effect of number of clients on the optimal block size.

Table 5.2: Effect of various parameters on the block size

Parameter	Effect of increase in parameter on optimal block size
Number of clients	Block size increases
Playback rate	Block size increases
Quality of Service (QoS)	Block size decreases
Number of disks	Block size decreases
Round duration	Block size increases
Disk zones	Block size increases from inner zones to outer zones
Parity Group Size	Block size increases

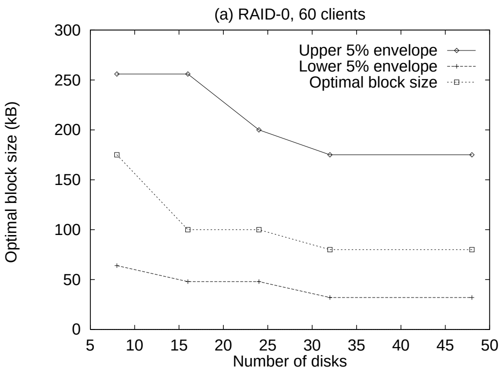


Figure 5.8: Effect of the number of disks on the optimal block size.

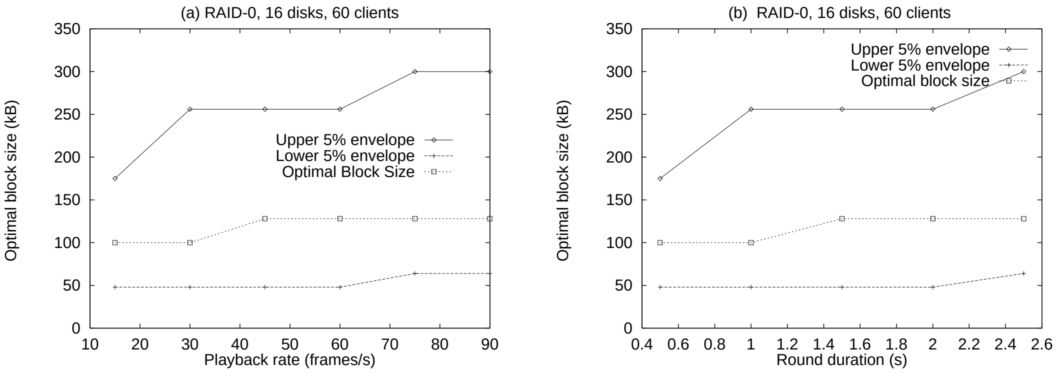


Figure 5.9: Effect of the playback rate of clients and the round duration on the optimal block size.

$\mathcal{O}$ to shift to the left. Thus, the optimal block size decreases with an increase in the number of disks (see Figure 5.8).

Playback Rate and Round Duration

Assuming a fixed round duration (playback rate), increasing the playback rate (round duration) causes a client to request a proportionately larger amount of data per round to sustain continuous playback. This causes a larger number of blocks to be accessed from the array, thereby spreading the load across disks and reducing the load imbalance. Consequently, the optimal block size and the 5% optimal envelope increase with increase in playback rate (round duration). (see Figures 5.9(a) and 5.9(b)).

Disk Characteristics

To evaluate the effect of varying disk characteristics on the optimal block size, we first define the *work coefficient* of a disk [17]:

Definition 5.1 *The work coefficient of a disk is defined as*

$$W = \frac{\textit{time to transfer unit amount of data}}{\textit{average seek} + \textit{average rotational latency}} \tag{5.22}$$

Table 5.3: Characteristics of various Seagate Disks

Model	Abbreviation	Capacity (MB)	Average seek (ms)	Avg Rotational latency (ms)	Transfer rate (MB/s)	Work Coefficient
Medalist	M	631	14	7.87	4.875	9.2 x 10^{-3}
Hawk	H	1050	9	5.54	6.9	9.7 x 10^{-3}
Barracuda1	B1	2150	8	4.17	7.5	11 x 10^{-3}
Barracuda2	B2	4294	8	4.17	7.5	11 x 10^{-3}
Elite9	E9	9090	11	5.56	6.8	8.7 x 10^{-3}

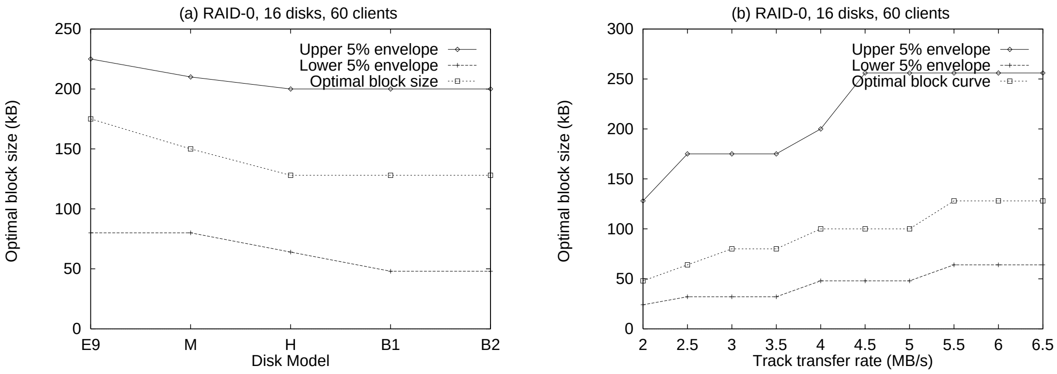


Figure 5.10: Variation in the optimal block size with disk characteristics. Figure (a) compares the optimal block size for different Seagate disks. Disks used in the experiment are Elite9, Medalist, Hawk, Barracuda1, and Barracuda2. Figure (b) shows the variation in the optimal block size for different transfer rates. Lower transfer rates represent inner zones.

The work coefficient measures the relative variation in the latency overheads and transfer times of disks. Table 5.3 shows the characteristics of various Seagate disks and their work coefficients.

Recall from (5.9) that

$$\widehat{\tau}_{\max} = \widehat{\mathcal{N}}_{\max} \cdot (t_s + t_r) + \widehat{\mathcal{N}}_{\max} \cdot B \cdot t_t$$

Hence, from the definition of $\mathcal{O}$, we get:

$$\mathcal{O} = 1 - \frac{\widehat{\mathcal{N}}_{\max} \cdot B \cdot t_t}{\widehat{\tau}_{\max}} = \frac{(t_s + t_r)}{(t_s + t_r) + B \cdot t_t} = \frac{1}{1 + B \cdot \frac{t_t}{(t_s + t_r)}} = \frac{1}{1 + B \cdot W}$$

Hence, for a particular block size, increasing W decreases $\mathcal{O}$. This causes the point of intersection of the $\mathcal{I}_s$ and $\mathcal{O}$ curves to shift to the left. This indicates that the optimal block size varies inversely with the work coefficient. Figure 5.10(a) and Table 5.3 demonstrate this behavior for different Seagate disks.

Zoned Disks

Our experiments thus far assumed a single transfer rate for the entire disk. However, modern disks are partitioned into zones, with outer zones having higher recording densities and larger data transfer rates as compared to inner

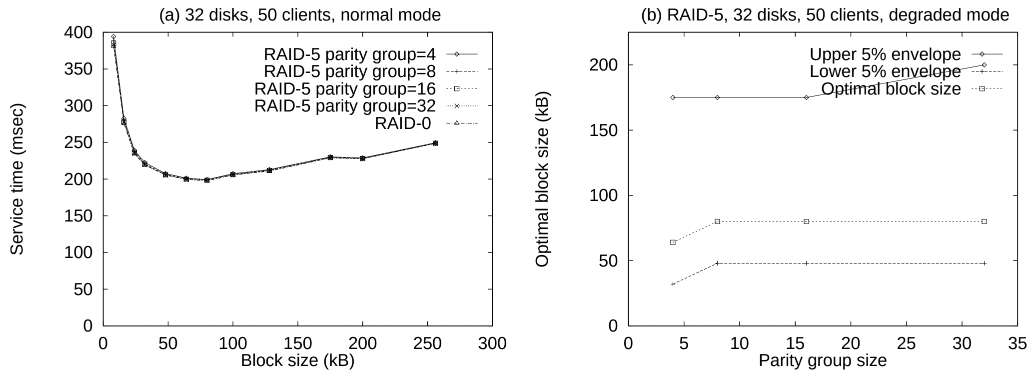


Figure 5.11: Effect of parity group size.

zones. Due to larger transfer rates (and hence, smaller transfer times), outer zones have a smaller work coefficient. Consequently, the optimal block size and the 5% optimal envelope for a zone increases as we proceed from inner zones to outer zones (see Figure 5.10(b)).

Since the optimal block size varies across zones, an integrated file system can either choose different block sizes for different zones, or choose a single block size for all zones. Whereas the former policy complicates storage space management due to the need for managing multiple block sizes, the latter policy can cause an increase in the service time of the most heavily loaded disk. To minimize the increase in the service time in the latter policy, the server must select a block size that is contained within the $x\%$ optimal envelope of all zones. This ensures that the service time of the most heavily loaded disk is always within $x\%$ of the minimum. Observe that these policies form two ends of a spectrum. The server can simplify storage space management and reduce loss in performance by choosing an intermediate policy that groups consecutive zones and selects a single block size for each group.

Parity Group Size

Since non-redundant arrays do not maintain any parity information, the parity group size is a parameter that is relevant only to redundant disk arrays. Figure 5.11(a) depicts the service time of the most heavily loaded disk in a RAID-5 array in the absence of a failure. It demonstrates that, in the absence of a disk failure, the service time of the most heavily loaded disk in a RAID-5 array is independent of the parity group size, and in fact, almost identical to that for an equivalent RAID-0 array. Consequently, so is the optimal block size.

Next consider the RAID-5 array with a single disk failure. Let G denote the parity group size. In such a scenario, whenever a client accesses a block stored on the failed disk, the server must access the remaining blocks of the parity groups stored on the surviving $G - 1$ disks to reconstruct the requested block. Hence, with increase in parity group size, the number of additional blocks that must be accessed to reconstruct a block on the failed disk increases, increasing the load on surviving disks. This results in an *effective* increase in the playback rate of clients. As explained in earlier in this section, increasing the playback rate of clients causes an increase in the optimal media block size and the 5% optimal envelope. Hence, the optimal block size and the optimal envelope in the degraded mode increase with increase in the parity group size (see Figure 5.11(b)).

5.1.4 Selecting an Optimal Block Size

Having examined the effect of the server configuration and the workload characteristics on the block size, we now present procedures for selecting an optimal block size. This procedure depends on the system design goals, which in turn are dictated by the operating environment. To illustrate, in integrated file systems employed for providing commercial services (e.g., movie-on-demand, online news, etc.), the primary goal is to maximize revenue by maximizing the number of clients that can be supported by the server. In contrast, integrated file systems employed in general purpose computing environments service clients with heterogeneous QoS, and hence, the number of clients supported depends on the exact workload mix (i.e., the proportion of clients with different requirements). Since the workload mix can vary over time, the goal for such servers is to provide the best possible performance over a wide range of workloads. Differing design goals may require the system designer to choose completely different media block sizes.

To determine a block size that maximizes the number of clients supported, let us assume that all parameters determining the file server configuration (i.e., the number of disks, their physical characteristics, the round duration, etc.) are known at design time. Also, assume that the data rate of clients and their QoS requirements are known. Then, a block size that maximizes the number of clients supported can be computed by the following two step procedure: (1) For a given number of clients, n, determine the service time of the most heavily loaded disk for different block sizes and select the block size that minimizes the service time; (2) If the service time of the most heavily loaded disk for this block size is less than the round duration, then increment n and repeat step (1). The block size that is obtained when the service time of the most heavily loaded disk equals the round duration maximizes the number of clients supported by the server.

In general computing environments, due to the heterogeneous nature of the workload, some of the workload characteristics may be unknown at design time (e.g., the number of clients accessing the server). In such a scenario, a block size that yields good performance over a wide range of workloads must be chosen [17]. For every parameter that is unknown at design time, the range over which the parameter is likely to vary must first be estimated. The optimal block size and the $x\%$ optimal envelope for each combination of these parameters is then computed using the model. Let $\mathcal{S}_1, \mathcal{S}_2, \cdots$ denote sets, each containing the $x\%$ optimal envelope for a particular combination of these parameters. Then, the set of block sizes that yields service times within $x\%$ of the minimum over all possible combinations of these parameters is $\mathcal{S} = \mathcal{S}_1 \cap \mathcal{S}_2 \cap \ldots$. If $\mathcal{S}$ is empty, then the entire procedure must be repeated for a larger values of x until a non-empty set of block sizes is obtained. Figure 5.12 illustrates the process of computing a feasible solution (i.e., a non-empty set $\mathcal{S}$) over a range of client workloads.

5.2 Determining the Degree of Striping

In addition to determining the stripe unit size, defining a placement policy requires the determination of degree of striping. An integrated file system can either stripe a file across all disks in the array or across a subset of the disks. Whereas the former policy is referred to as *wide striping*, the latter policy is referred to as *narrow striping* [31].

To evaluate the relative merits of these policies, consider an integrated file system that employs wide striping to interleave continuous media files across disks in the array. Let us assume that the performance of the server is measured in terms of the maximum number of clients that it can support. In an ideal scenario, an increase in the number of disks in the system should result in a linear increase in the number of clients that can be supported by

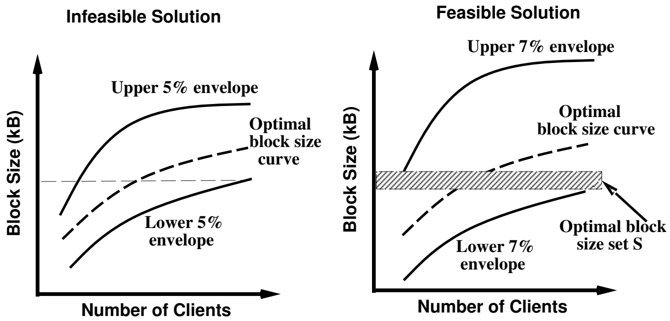


Figure 5.12: Selecting a block size that yields near-optimal performance, regardless of the number of clients accessing the server. The shaded region denotes the set of block sizes $\mathcal{S}$ that yield service times within 7% of the minimum for all workloads.

the server. That is, the number of clients supported by a disk array consisting of D disks should be D times the number of clients that can be supported by a single disk. However, as shown in Figure 5.13(a), the number of clients supported by the server increases sub-linearly with an increase in the number of disks. This can be attributed to the following two reasons:

- *Real-time requirements of clients*: Due to the real-time requirements of clients, the number of clients supported by the server is constrained by the most heavily loaded disk. Specifically, the number of clients accessing the server reaches its maximum value when the service time of the most heavily loaded disk equals the round duration. At this point, however, the service time of a disk with average load is smaller than the round duration. The resulting load imbalance causes most of the disks in the array to be under-utilized.

- *Reduction in optimal block size*: As explained in Section 5.1.3.2, an increase in the number of disks in the system causes the load imbalance $\mathcal{I}_s$ to increase. An increase in the number of disks also increases the number of clients that can be supported by the server. The larger the number of clients accessing the server, the smaller the load imbalance $\mathcal{I}_s$. Thus, the combined effect of increasing the number of disks and the number of clients accessing the server governs the actual value of $\mathcal{I}_s$. Figure 5.13(b) plots the variation in imbalance $\mathcal{I}_s$ against the (number of disks in the system, maximum number of clients supported) pairs. It illustrates that the increase in $\mathcal{I}_s$ due to an increase in the number of disks dominates the decrease in $\mathcal{I}_s$ due to an increase in the number of clients, causing the actual imbalance to increase. Hence, a small block size must be chosen to compensate for the increased imbalance, causing a decrease in the optimal block size (see Figure 5.13(c)). Since a small block size imposes a larger latency overhead, the overall throughput of the array decreases, causing a reduction in the number of clients that can be supported.

To minimize the impact of these factors, a server should: (1) partition the disk array into mutually exclusive groups of disks, and (2) stripe each file only within a partition. Since each partition acts as an independent disk array and the number of disks per partition is small, such an approach: (1) reduces the load imbalance within each partition, and (2) increases the optimal block size for a partition (and thereby reduces the latency overhead). In such partitioned arrays, load imbalances can occur if clients are not equitably distributed among all the partitions. Hence, the partition size must be chosen so as to simultaneously minimize the impact of load imbalance across partitions and

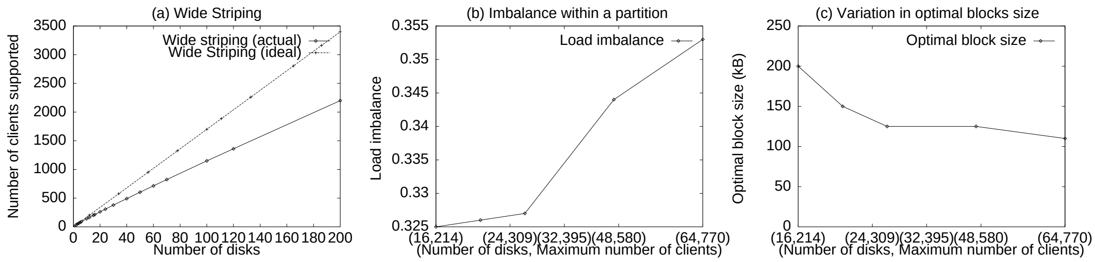


Figure 5.13: Loss in the number of clients supported in large disk arrays and factors contributing to this loss.

the load imbalance within a partition. In what follows, we first present a model for determining the load imbalance across partitions, and then describe a procedure for determining the a partition size that maximizes the number of clients supported.

5.2.1 Modeling the Imbalance Across Partitions

To compute the load imbalance across partitions, consider a disk array consisting of D disks that is partitioned into groups of d disks each. Assume that the file server employs a placement policy that assigns files to partitions such that each partition is equally likely to be accessed by a new request [25, 95]. That is, the probability that a newly arriving client accesses a partition is $q = d/D$. In such a scenario, if n clients access the server, then the probability that m clients access the j^{th} partition is binomially distributed, and is given as:

$$P(\mathcal{Y}_j = m) = \binom{n}{m} \cdot q^m \cdot (1-q)^{n-m} \tag{5.23}$$

where $\mathcal{Y}_j$ is a random variable representing the number of clients accessing the j^{th} partition. Then the number of clients accessing the most heavily loaded partition is

$$\mathcal{Y}_{\max} = \max(\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_{\frac{D}{d}}) \tag{5.24}$$

Since the load on a partition is independent of other partitions, $\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_{\frac{D}{d}}$ are independent random variables. Hence, the distribution of $\mathcal{Y}_{\max}$ can be computed as:

$$F_{\mathcal{Y}_{\max}}(x) = F_{\mathcal{Y}_1}(x) \cdot F_{\mathcal{Y}_2}(x) \cdots F_{\mathcal{Y}_{\frac{D}{d}}}(x) \tag{5.25}$$

where $F_{\mathcal{Y}_j}$ is the cumulative probability distribution function of the random variable $\mathcal{Y}_j$ [66].

Given the distribution of $\mathcal{Y}_i$ and $\mathcal{Y}_{max}$, we can compute the expected number of requests on the average and the most heavily loaded partitions (denoted by $\widehat{\mathcal{Y}}$ and $\widehat{\mathcal{Y}}_{max}$, respectively). Using these values, we can define the the load imbalance across partitions (denoted by $\mathcal{I}_p$) as:

$$\mathcal{I}_p = \left(1 - \frac{\widehat{\mathcal{Y}}}{\widehat{\mathcal{Y}}_{max}}\right) \tag{5.26}$$

Thus, given the number of disks in the array and the partition size, we can compute the load imbalance across partitions.

5.2.2 Determining the Partition Size

For a fixed number of disks, increasing the partition size increases the load imbalance $\mathcal{I}_s$ within a partition (Figure 5.13(b)), while decreasing the load imbalance $\mathcal{I}_p$ across partitions (Figure 5.14). Moreover, as shown in Figure 5.13(c), increasing the partition size results in a reduction in the optimal block size (thereby increasing the seek and rotational latency overhead). Consequently, the server must determine the degree of striping (i.e., partition size) that balances these tradeoffs.

Given the models for predicting: (1) the load imbalance across partitions (Section 5.2.1), (2) the load imbalance within a partition (Section 5.1.1.1), a procedure for choosing a partition size that maximizes the number of clients supported by the server is as follows:

Procedure ComputePartitionSize

1. Choose an initial partition size of d=1.
2. Using the model presented in Section 5.1.1.1, compute the maximum number of clients, n', that can be supported by a single partition of size d (i.e., the number of clients at which the service time of the most heavily loaded disk equals the round duration).
3. Assuming that n clients access the array, using the model presented in Section 5.2.1, compute the expected number of clients, $\widehat{\mathcal{Y}}_{max}$, accessing the most heavily loaded partition.
4. If $\widehat{\mathcal{Y}}_{\max} < n'$, then increment n and repeat step (3). When $\widehat{\mathcal{Y}}_{\max} = n'$, then n denotes the maximum number of clients that can be supported by the array with a partition size of d.
5. Increment the partition size d, and repeat steps (2) thorough (4) until no further improvements in the number of clients is obtained (i.e., until n starts decreasing with increase in d). This yields a partition size that maximizes the number of clients that can be supported.

In the above procedure, note that the limit on the number of clients that can be supported by the entire array is reached when the most heavily loaded partition reaches its maximum capacity. However, at this point, the number of clients accessing other partitions is less than their maximum capacity. Hence, the total number of clients that can be supported by the array does not increase linearly with number of partitions (i.e., $n < n' \cdot \frac{D}{d}$).

Figure 5.15(a) illustrates the result of executing this iterative procedure for an array of 120 disks. Since the number of clients that can be supported by the array is maximized at $d = 10$, the array should be partitioned into 12 partitions of 10 disks each for optimal performance. Figure 5.15(b) demonstrates the variation in the optimal partition size with increase in the number of disks in the array. Finally, Figure 5.15(c) illustrates the improvement in the number of clients supported due to partitioning. For small disk arrays, since wide striping is close to the ideal case, the additional gains due to partitioning are small. For large disk arrays, however, partitioning yields a approximately a 10% increase in the number of clients supported as compared to the wide striping. Figure 5.15(c) also demonstrates that partitioning coupled with static load balancing algorithms does not completely bridge the gap between the number of clients supported by the array in the ideal case (i.e., when the number of clients increases linearly with array size) and that obtained using wide striping. To further reduce the loss in the number of clients supported, the server must replicate files across partitions and employ dynamic load balancing schemes. The improvement in performance yielded by such a scheme is at the expense of higher storage space requirement and more complex

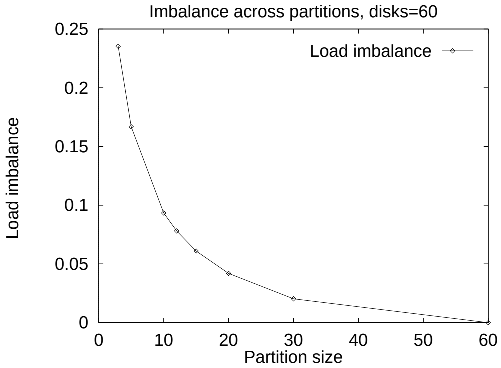


Figure 5.14: Variation in the imbalance across partitions with increase in the partition size

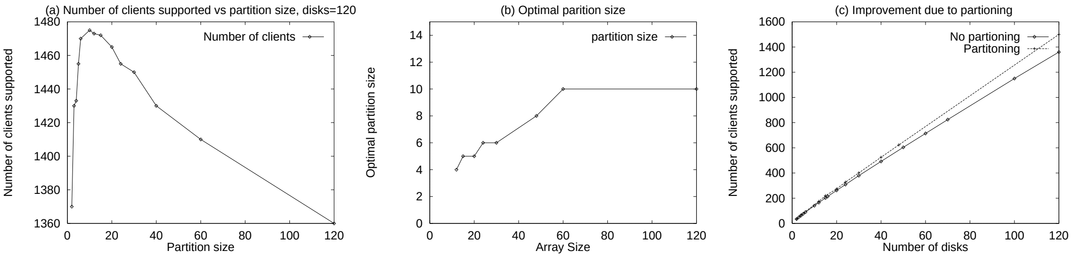


Figure 5.15: Partitioning large disk arrays.

storage space management algorithms. Detailed cost-performance tradeoffs of such an approach are beyond the scope of this dissertation.

5.3 Related Work

Several research projects have developed simulation and analytical techniques for optimizing the performance of striped disk arrays for conventional workloads [17, 51]. However, due to the real-time nature of continuous media accesses, these techniques are not directly applicable for optimizing performance for continuous media workloads.

The problem of determining the optimal stripe unit size for non-redundant arrays storing continuous media was studied in [92]. A model that predicts the service time of the most heavily loaded disk for non-redundant arrays (henceforth referred to as the VRG model) was also proposed. The VRG model uses worst case assumptions about the number of blocks accessed by a client during a round to compute the service time of the most heavily loaded disk. In contrast, our model uses actual distributions of the number of blocks accessed by a client during a round to compute the service time of the most heavily loaded disk. Due to worst-case assumptions, the service time predicted by the VRG model is higher than the actual service time of the most heavily loaded disk (see Figure 5.16(a)). Since the VRG model is conservative, as illustrated in 5.16(b), the optimal block size computed using the VRG model will cause the server to support a smaller number of clients. To derive this graph, we first computed the optimal block size using both models, and then determined the number of clients supported by the server using our model. If the VRG model were to be used to determine the number of clients supported by the server (in addition to using the model to compute the optimal block size), then the number of clients supported would be even lower. The problem

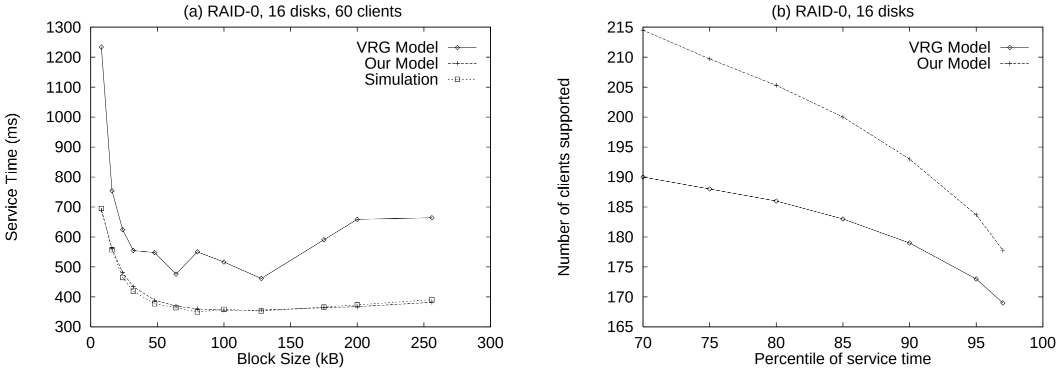


Figure 5.16: Comparison with the VRG model.

of determining block size in redundant arrays or determining the degree of striping was not addressed in the paper.

The problem of determining the degree of striping has not received much attention in the literature. A comparison of wide and narrow striping schemes was presented in [31]. The focus of their effort was to evaluate the effect of replicating media streams across array partitions on the response time. The problem of determining the partition size was not addressed in the paper. The problem of assigning media streams to array partitions subject so as to balance the load across partitions has been dealt in [25, 95]. These efforts complement our work since they do not deal with the issue of determining an optimal partition size for large disk arrays.

5.4 Concluding Remarks

In this chapter, we described techniques for determining the stripe unit size and the degree of striping for continuous media data. To determine the optimal stripe unit size, we presented an analytical model that uses the server configuration and the workload characteristics to predict the load on the most heavily loaded disk in redundant and non-redundant arrays. We used the model to evaluate the effect of various parameters on the optimal block size. We also demonstrated that employing wide striping causes the number of clients supported to increase sub-linearly with increase in the number of disks. To maximize the number of clients supported in large arrays, we proposed a scheme that partitions such arrays and stripes each file across a single disk partition. Since load imbalances can occur in partitioned arrays, we presented a model to determine the imbalance across partitions and described a procedure for determining a partition size that maximizes the number of clients supported by the array. The analytical models presented in this chapter are the first to accurately characterize the load on the disk array for VBR continuous media. The only previously known model for VBR continuous media [92] uses worst case assumptions, and hence, yields sub-optimal results. Furthermore, our models can also be used by integrated file systems to compute the maximum number of clients that can be supported, which can then be used for admission control.

Chapter 6

Failure Recovery Techniques

Pretty daring of you to be storing important files on a UNIX file system.

—Robert E. Seastrom

Integrated file systems employ disk arrays for storing data. Disk arrays connect several disks together, and thereby extend the cost, power, and size advantages of small disks to high capacity configurations [14]. A fundamental tradeoff, though, is that large disk arrays are highly susceptible to disk failures [19]. To illustrate, although the mean time to failure for a single disk is 300,000 hours, an array of 1000 disks will experience a failure every 12 days. To minimize down time due to frequent disk failures, an integrated file system should employ failure recovery techniques that enable it to mask such failures from users.

Recently several research projects have investigated the design of fault-tolerant storage systems [14, 57, 94]. Most of these approaches are based on the Redundant Array of Independent Disks (RAID) architecture [19, 67], which achieves fault tolerance either by *mirroring* or *parity encoding*. Mirroring (also referred to as RAID level 1) achieves fault tolerance by duplicating data on separate disks [10, 16, 24]. By maintaining two copies of the data, mirrored arrays achieve fault-tolerance by accessing backup blocks if the primary blocks are unavailable due to a disk failure. Even in the fault-free state, such arrays service read requests from the disk with the lighter load, thereby yielding better performance. A limitation of mirroring though is that it incurs a 100% storage space overhead.

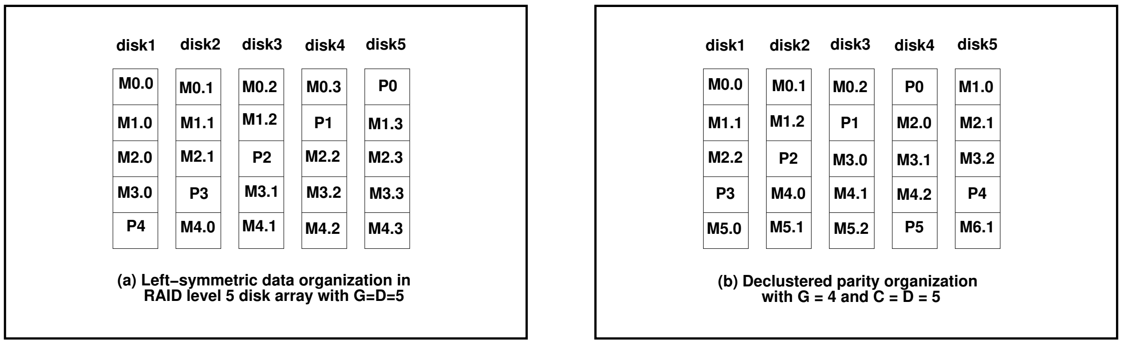


Figure 6.1: Left-symmetric and declustered parity organizations in parity-based arrays. $M_{i.j}$ and P_i denote data and parity blocks, respectively, and $P_i = M_{i.0} \oplus M_{i.1} \cdots \oplus M_{i.(G-2)}$

Parity encoding techniques, employed in RAID levels 3, 4, and 5, reduce this overhead by employing error correcting codes [34, 67]. In a disk array consisting of D disks, parity is computed by an exclusive-or operation over data stored across $(D - 1)$ disks, and is stored on another disk [35, 50, 67]. The parity block together with all the data blocks over which parity is computed are referred to as a *parity group*. In RAID level 3 or *bit-interleaved parity*, data is interleaved bit-wise over the data disks and a dedicated disk stores all parity information. In RAID level 4 or *block-interleaved parity*, data is interleaved in fixed-size blocks rather than in bits, with a dedicated disk storing parity blocks. Since parity must be updated on every write request, the parity disk can become a bottleneck in RAID levels 3 and 4. Hence, in RAID level 5 or *distributed block-interleaved parity*, parity blocks are uniformly distributed across all the disks (e.g., the left-symmetric parity assignment shown in Figure 6.1(a)). In parity-based RAID arrays, if one of the disks fail, the data on the failed disk is recovered by an exclusive-or operation over the data and the parity blocks stored on surviving disks. That is, a read access to a block on the failed disk causes one request to be sent to each surviving disk. Thus, if the system load is balanced prior to a disk failure, the surviving disks would observe twice as many read requests in the presence of a failure, causing a 100% increase in the load [39].

Several approaches that address this limitation by trading storage capacity for reduced failure recovery overhead have been proposed. In the *multiple* RAID architecture, an array of D disks is partitioned into clusters of C disks $(C \leq D)$ with each cluster independently computing parity information [18, 19]. Whereas a standard RAID array can tolerate a single disk failure, such an architecture can tolerate a failure in each cluster without losing data. Furthermore, in the presence of a failure, only disks within the cluster containing the failed disk see an increased load while disks in other clusters continue to see their normal workload. The *declustered parity* array (also referred to as *clustered RAID*) further reduces the overhead of failure recovery by constraining the number of blocks protected by parity to $(G - 1)$ instead of $(C - 1)$, $G < C$ [38, 58, 61]. By appropriately distributing the blocks of a parity group across the C disks in a cluster, such a policy ensures that the load on each surviving disk in a cluster increases only by $(G-1)/(C-1)$ instead of $(C-1)/(C-1) = 100\%$ for read requests. This is illustrated in Figure 6.1(b) where $G = 4$ and $C = D = 5$.

In summary, several techniques for designing fault-tolerant disk arrays have been recently proposed [19]. Most of these failure recovery techniques assume a conventional workload consisting of best-effort data accesses. However, the large recovery overhead imposed by these techniques can saturate the server, thereby adversely affecting real-time performance guarantees provided to continuous media applications. A simple approach that overcomes this limitation is to operate the server at low levels of utilization in the fault-free state, thereby preventing saturation after a failure. This results in under-utilization of resources in the fault-free state. Consequently, existing failure recovery techniques or their simple adaptations are unsuitable for continuous media applications

All existing failure recovery techniques, including those proposed for continuous media [9, 23, 60, 82, 84], treat data as an uninterpreted sequence of bits and do not exploit any of its characteristics. By exploiting the characteristics of continuous media, an integrated file system can significantly lower the overhead of online failure recovery. For instance, the file server can exploit the sequential nature of continuous media access to compute and prefetch parity information, and thereby reduce the overhead of failure recovery. Similarly, instead of perfectly recovering video data stored on the failed disk using error-correcting codes, the server can exploit human perceptual tolerances and the inherent redundancies in video files to *approximately* reconstruct lost image data. Failure recovery techniques that exploit the characteristics of stored data to reduce the recovery overhead constitute the subject matter of this chapter.

The rest of this chapter is organized as follows. Section 6.1 describes failure recovery techniques that exploit the inherent redundancy in video files to reduce recovery overhead, and presents instantiations of these techniques for JPEG and MPEG compression algorithms. Failure recovery techniques that exploit the sequential nature of continuous media accesses are presented in Section 6.2. We analytically compare our techniques to conventional recovery techniques in Section 6.3. Section 6.4 presents the results of our experimental evaluation. Section 6.5 presents related work, and finally, Section 6.6 summarizes our results.

6.1 Exploiting Inherent Redundancy of Video Streams

Conventional parity-based recovery techniques use error correcting codes for *exact* reconstruction of data stored on a failed disk. Since human perception is tolerant to minor distortions in video playback [69], an integrated file system can exploit the inherent spatial and temporal redundancies within video files to *approximately* reconstruct lost images,[1] and thereby substantially reduce the recovery overhead. To illustrate, let I denote an image in the video sequence, where

$$I = \{\ p(x,y)\ \mid\ p(x,y)\ \text{ is the pixel value at } (x,y)\}$$

Let each image I be partitioned into several sub-images $I_1, I_2, \cdots, I_N$ such that $I_i \subset I, \quad 1 \leq i \leq N$, and $I_1 \cup I_2 \cup \cdots \cup I_N = I$. If these sub-images are stored on different disks, then a single disk failure will result in the loss of a fraction of each image. If the sub-images are created such that none of the immediate neighbors of a pixel in the image belong to the same sub-image, then even in the presence of a single disk failure, all the neighbors of the lost pixels will be available. In this case, the high degree of correlation between neighboring pixels will make it possible to reconstruct a reasonable approximation of the original image. Moreover, no additional information will have to be retrieved from any of the surviving disks for recovery. Since the images are partitioned in the pixel domain (i.e., prior to compression), we refer to the process as *pre-compression partitioning*.

Although conceptually elegant, such pre-compression image partitioning techniques significantly reduce the correlation between the pixels assigned to the same sub-image, and hence adversely affect image compression efficiency [70, 86]. The resultant increase in the bit-rate requirement may impose a higher load on each disk even in the fault-free state, and thereby reduce the number of clients that can be supported by the file server. Alternatively, the server can employ *post-compression* partitioning techniques which partition each image into several sub-images *after* the redundancies within the video file have been exploited by the compression algorithm. The key challenge in designing post-compression partitioning schemes is to create sub-images that facilitate effective and efficient recovery of lost image data without significantly affecting the compression efficiency. Most compression algorithms use a transform function (such as the *discrete cosine transform (DCT)* or the *wavelet transform*) that converts an image from the pixel domain to the frequency domain by packing most of the spectral energy into a small number of coefficients, thereby achieving compression. Consequently, post-compression partitioning depends on the characteristics of the transform function used to encode video files. In what follows, we first describe the characteristics of the DCT, and then present a partitioning scheme that exploits these characteristics to effectively reconstruct lost transform coefficients. The partitioning scheme can be used with any compression algorithm using the discrete cosine transform. We illustrate our method by presenting failure resilient schemes for two popular compression algorithms used for

[1] We shall use the terms image and frame interchangeably in this chapter.

video sequences (namely, motion JPEG and MPEG). By integrating these schemes with placement techniques, we derive a new disk array architecture for storing video files.

6.1.1 Characteristics of the Discrete Cosine Transform

Compression algorithms based on the discrete cosine transform partition each image into nxn pixel blocks and independently apply the DCT to each block. The DCT uncorrelates each pixel block into an array of n^2 coefficients such that most of the spectral energy is packed into a small number of low frequency coefficients. Whereas the lowest frequency coefficient (referred to as the *DC coefficient*) captures the average brightness and color of the spatial block, the remaining set of $(n^2 - 1)$ coefficients (referred to as the *AC coefficients*) capture the details within the nxn pixel block. The coefficients produced by the DCT have the following characteristics:

- Since the DC coefficients capture the average brightness and color of each 8x8 pixel block and since the average brightness and color of pixels gradually change within a image, the DC coefficients of neighboring nxn pixel blocks are correlated. Consequently, the value of the DC coefficient of a block can be reasonably approximated from the DC coefficients of the neighboring blocks.

  To formally capture this observation, consider an image containing NCOL $\times$ NROW blocks of nxn pixels each. Let us define the *8-neighborhood* of a block at location (x, y) (denoted by $B(x, y)$) as the set:

  $$\mathcal{N}_8(B(x,y)) = \{B(i,j) \mid |x-i| \leq 1 \text{ AND } |y-j| \leq 1\} - \{B(x,y)\} \tag{6.1}$$

  Then, the DC coefficient of $B(x, y)$ can be approximated as:

  $$\mathrm{DC}_{B(x,y)} = \frac{1}{8} * \sum_{B_{(i,j)} \in \mathcal{N}_8(B(x,y))} \mathrm{DC}_{B(i,j)} \tag{6.2}$$

  where $\mathrm{DC}_{B(i,j)}$ denotes the DC coefficient of block $B(i, j)$.

- Due to the very nature of DCT, the set of AC coefficients yielded for each nxn block are uncorrelated. Moreover, since DCT packs the most amount of spectral energy into a small number low frequency coefficients, quantizing the set of AC coefficients (by using a user-defined normalization array) yields many zeroes, especially at higher frequencies. Consequently, recovering a block by simply substituting a zero for each of the lost AC coefficients is generally sufficient to obtain a reasonable approximation of the original image (at least as long as the number of lost coefficients are small and are scattered throughout the block).

6.1.2 Image Partitioning Fundamentals

To precisely describe the partitioning process, let $\mathcal{I}$ denote the frequency domain image obtained by applying the DCT to the original image I. Then, the partitioning algorithm exploits the characteristics of the DCT by proceeding in two steps:

1. *Scrambling:* In this step, the partitioning algorithm scrambles image $\mathcal{I}$ by uniformly distributing the AC coefficients of a DCT block across multiple blocks.

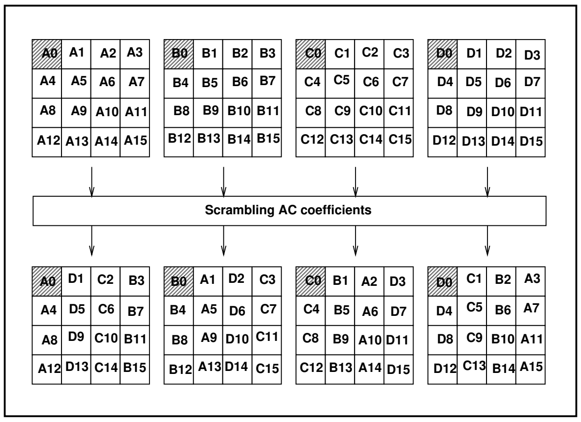


Figure 6.2: Scrambling AC coefficients. Here A_0, B_0, C_0 and D_0 denote DC coefficients, and A_i, B_i, C_i and D_i $(1 \leq i \leq 15)$ represent AC coefficients.

2. *Sub-image creation:* The scrambled image $S(\mathcal{I})$ is then partitioned into a set of N sub-images such that none of the DC coefficients in the 8-neighborhood of a block belong to the same sub-image. If $\mathcal{I}_1, \mathcal{I}_2, \cdots, \mathcal{I}_N$ denote the sub-images, then the partitioning algorithm also ensures that: (i) $\mathcal{I}_j \subset S(\mathcal{I}),\ 1 \leq j \leq N$, and (ii) $\mathcal{I}_1 \cup \mathcal{I}_2 \cup \cdots \cup \mathcal{I}_N = S(\mathcal{I})$.

Note that this partitioning process is distinct from pre-compression image partitioning since the sub-images are created in the frequency domain as opposed to in the pixel domain. In fact, as we shall demonstrate later, it is this feature of the partitioning algorithm that enables reasonable failure recovery without incurring any significant degradation in compression efficiency. We present our partitioning algorithm by first describing a method for scrambling AC coefficients and then describing the sub-image creation process.

6.1.2.1 Scrambling AC Coefficients

Although substituting lost AC coefficients by zeroes yields a reasonable approximation of the original image, the degradation in image quality can be minimized by reducing the number of lost coefficients. The number of lost coefficients can be minimized by scattering the AC coefficients of a block amongst multiple sub-images. To achieve this objective, the partitioning algorithm employs a scrambling function f which when given a set of M DCT blocks, creates a new set of M blocks such that the AC coefficients from each of the input blocks are uniformly distributed amongst all of the output blocks. Furthermore, to prevent scrambling from adversely affecting compression efficiency, the scrambling function must ensure that the relative positions of AC coefficients in scrambled blocks is the same as that in the input blocks. Any scrambling function that satisfies these requirements can be used by the partitioning algorithm.

To describe one such scrambling function f, let $\mathcal{B}_0, \mathcal{B}_1, \cdots \mathcal{B}_{M-1}$ denote the DCT blocks of the original image. Then,

$$f(\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{M-1}) = \{\mathcal{B}'_0, \mathcal{B}'_2, \cdots, \mathcal{B}'_{M-1}\}$$

where $\mathcal{B}'_0, \mathcal{B}'_1, \cdots \mathcal{B}'_{M-1}$ denote the scrambled DCT blocks. Let us assume that the AC coefficients are numbered from left-to-right in a row-major order and that $\mathrm{AC}^k_{\mathcal{B}_i}$ denotes the k^{th} AC coefficient ($k \in [1, n^2 - 1]$) of block $\mathcal{B}_i$.

The scrambling function f assigns $\mathrm{AC}^k_{\mathcal{B}_i}$ to be the k^{th} coefficient of block $\mathcal{B}'_j$ where $j = (i+k) \bmod M$. Thus, each resulting block contains $\frac{n^2}{M}$ coefficients of each of the original blocks. Specifically, one of the blocks contain the DC coefficient and $\left(\frac{n^2}{M} - 1\right)$ AC coefficients, and all the remaining $(M-1)$ blocks contain $\left(\frac{n^2}{M}\right)$ AC coefficients. Figure 6.2 illustrates the scrambling of AC coefficients for four 4x4 blocks.

6.1.2.2 Creating Sub-images

Having scrambled the image, it must then be partitioned into sub-images such that none of the DC coefficients in the 8-neighborhood of a block belongs to the same sub-image. The minimum value of the degree of image partitioning N (i.e., the number of sub-images) that satisfies this requirement is determined by the following theorem:

Theorem 6.1 To ensure that none of the blocks contained in a sub-image are in the 8-neighborhood of each other in the original image, the image must be partitioned into at least 4 sub-images.

Proof: We demonstrate that partitioning an image into 4 sub-images is both necessary and sufficient.

- *Necessity*: Consider a DCT block $\mathcal{B}(x,y)$ as well as all of the blocks in its 8-neighborhood. Let us partition the set of blocks into four groups G_1, G_2, G_3, and G_4, such that:

$$\begin{aligned}
G_1 &= \{\mathcal{B}(x,y)\} \\
G_2 &= \{\mathcal{B}(x-1,y), \mathcal{B}(x+1,y)\} \\
G_3 &= \{\mathcal{B}(x-1,y-1), \mathcal{B}(x-1,y+1), \mathcal{B}(x+1,y-1), \mathcal{B}(x+1,y+1)\} \\
G_4 &= \{\mathcal{B}(x,y-1), \mathcal{B}(x,y+1)\}
\end{aligned}$$

  By the definition of 8-neighborhood, it is clear that none of the groups contain any two blocks that are in the 8-neighborhood of each other. Moreover, none of the groups can be merged together without violating this condition. Hence, to ensure that no sub-image contains two blocks that are in the 8-neighborhood of each other, an image must be partitioned into at least 4 sub-images.

- *Sufficiency*: To prove the sufficiency, let us partition an image into 4 sub-images (denoted by $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$, and $\mathcal{I}_4$) as follows:

$$\begin{aligned}
\mathcal{I}_1 &= \{\mathcal{B}(i,j) \mid (i \bmod 2) = 0 \text{ AND } (j \bmod 2) = 0\} \\
\mathcal{I}_2 &= \{\mathcal{B}(i,j) \mid (i \bmod 2) = 1 \text{ AND } (j \bmod 2) = 0\} \\
\mathcal{I}_3 &= \{\mathcal{B}(i,j) \mid (i \bmod 2) = 0 \text{ AND } (j \bmod 2) = 1\} \\
\mathcal{I}_4 &= \{\mathcal{B}(i,j) \mid (i \bmod 2) = 1 \text{ AND } (j \bmod 2) = 1\}
\end{aligned}$$

  Since these definitions cover blocks in both odd and even numbered rows as well as columns, the 4 sub-images together contain all the blocks within the original image. Moreover, each block within the original image is a member of of exactly one of the sub-images.

To demonstrate that none of the blocks contained in a sub-image belongs to the 8-neighborhood of each other, without loss of generality, consider two distinct blocks $\mathcal{B}(i_1, j_1)$ and $\mathcal{B}(i_2, j_2)$ that belong to a sub-image. Then, by the definition of sub-images, either $|i_1 - i_2| \geq 2$ or $|j_1 - j_2| \geq 2$, or both. Hence, by definition (see Equation (6.1)), blocks $\mathcal{B}(i_1, j_1)$ and $\mathcal{B}(i_2, j_2)$ do not belong to the 8-neighborhood of each other. Thus, the definitions of sub-images $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$, and $\mathcal{I}_4$ denote a scheme for partitioning an image which guarantees that no sub-image contains blocks that are in the 8-neighborhood of each other. ■

6.1.2.3 Determining Image Partitioning Parameters

Before partitioning an image, the algorithm must first choose the values of parameters M and N (i.e., the number of DCT blocks produced by each invocation of the scrambling function and the degree of image partitioning, respectively). To enable good recovery of DC coefficients, the algorithm must choose $N \geq 4$, where the exact value of N is governed by the required quality of the reconstructed image. To minimize the impact of lost AC coefficients on the visual quality of an image, the partitioning algorithm must assign no more than one block from every set of M scrambled blocks to each sub-image. This requires that M be at most N (i.e., $M \leq N$). Furthermore, since each scrambled block consists of $1/M$th of the AC coefficients from each of M input blocks, the number of lost AC coefficients can be minimized by choosing M as large as possible. Hence, the algorithm can minimize the degradation in image quality due to a failure by choosing $M = N$.

If an image is partitioned into 4 sub-images, then each sub-image will contain 25% of the image data in the frequency domain. Consequently, if the information contained in a sub-image is not available, the image will have to be reconstructed from the remaining 75% of the data. Since the quality of the reconstructed image is dependent on the amount of original image data available for reconstruction, increasing the degree of image partitioning improves the quality of the reconstructed images. However, as we shall point out later, this increase also decreases the correlation between the DC coefficients of blocks assigned to the same sub-image, and thereby deteriorates compression efficiency. Hence, the degree of image partitioning must be chosen to balance these concerns. In what follows, we show how scrambling and sub-image creation can be combined to derive loss-resilient versions of JPEG and MPEG.

6.1.3 Loss-Resilient JPEG (LRJ) Algorithm

The JPEG compression algorithm groups image data into a sequence of 8x8 pixel blocks. Each pixel block is then subjected to the DCT which yields a DC coefficient and 63 AC coefficients. The DC coefficients of successive blocks are difference encoded using a DPCM scheme independent of the AC coefficients. Within each block, the AC coefficients are quantized to remove high frequency components, scanned in a zig-zag manner to obtain an approximate ordering from lowest to highest frequency, and finally run length and entropy encoded. Figure 6.3 depicts the main steps involved in the JPEG compression algorithm [68]. The motion-JPEG algorithm applies the JPEG algorithm to a sequence of images yielding a compressed video file.

The *Loss-Resilient JPEG (LRJ)* algorithm is an enhancement of the JPEG compression standard, and uses the partitioning technique presented in the Section 6.1.2. Given that each image in a video file is being partitioned into $N (N \geq 4)$ sub-images, the LRJ compression algorithm is as follows:

1. *Scrambling:* The algorithm selects N consecutive DCT blocks from the same row of the original image and scrambles the AC coefficients within the blocks to create a new set of N blocks. Since each invocation of the

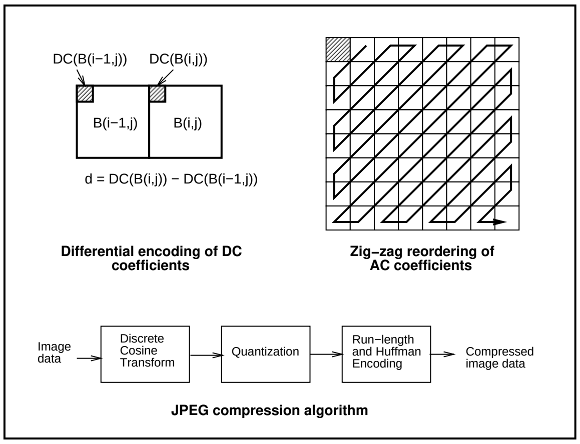


Figure 6.3: JPEG compression algorithm

scrambling function requires N blocks from the same row, N is chosen such that it divides the total number of blocks in a row. In the event that this is not possible (e.g., when the number of blocks in a row are prime), each row of blocks is padded with additional "zero" blocks such that the number of blocks in a row becomes an integral multiple of N. Since all coefficients in such blocks are zeroes, they can be efficiently run-length and huffman encoded without significant degradation in compression efficiency.

2. *Sub-image Creation:* Blocks obtained from each invocation of the scrambling function are then assigned to sub-images (one block per sub-image) such that none of the DC coefficients contained in a sub-image belong to blocks that are in the 8-neighborhood of each other. Since $N \geq 4$, the latter objective can be achieved by assigning the scrambled blocks belonging to a row to sub-images in a round-robin manner, and by ensuring that the assignment of the first block from each row is offset by 2 sub-images from the corresponding block in the previous row. That is, if block $\mathcal{B}'(i,j)$ is assigned to sub-image $\mathcal{I}_k$, then block $\mathcal{B}'(i+1,j)$ is assigned to sub-image $\mathcal{I}_{(k+1) mod N}$, $0 \leq i \leq$ NCOL, $0 \leq j \leq$ NROW. Moreover, if block $\mathcal{B}'(i,j)$ is assigned to sub-image $\mathcal{I}_k$, then block $\mathcal{B}'(i,j+1)$ is assigned to sub-image $\mathcal{I}_{(k+2) mod N}$.

Once all the blocks within the image have been processed, each of the N sub-images can be independently encoded. Specifically, the DC coefficients within each sub-image are encoded with a lossless DPCM scheme using the DC coefficient from the previous block assigned to the sub-image as a 1-D predictor. Similarly, the 2-D array of 63 AC coefficients within each block is formatted as a 1-D vector using a zigzag reordering, and then run-length and huffman encoded. Note that the huffman tables utilized for this purpose can either be optimized over each individual sub-image or over the entire image. Whereas the former approach will require a huffman table to be stored with each sub-image, the latter requires a single huffman table to be stored for an entire image. However, in such a scenario, the huffman table must be replicated across multiple sub-images to guarantee its availability even when one or more of the sub-images are not available.

At the time of decompression, once each sub-image has been run-length and huffman decoded, the LRJ algorithm repeatedly selects a block from each of the N sub-images (referred to as the *merging* step) and uses an unscrambling function to obtain blocks of the original image. In the event that the information contained in a sub-image is not available, the unscrambling function also performs a predictive reconstruction of the lost DC coefficients from the

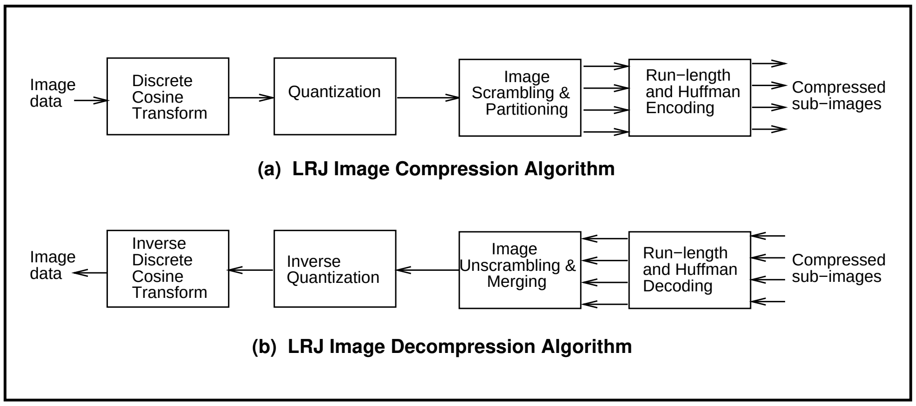


Figure 6.4: LRJ compression and decompression algorithms

Procedure LRJ-Compress
begin
  Input: Image consisting of NROW $\times$ NCOL
      8x8 pixel blocks and the value of N;
  Perform DCT on each 8x8 pixel block;
  Quantize the coefficients using a user-defined matrix;
  NCOL$'$ $=$ NCOL rounded to the next higher
      multiple of N;
  Pad each row of the transformed image with
    $(\text{NCOL}' - \text{NCOL})$ zero blocks;
  for ($i = 0$ to NROW $- 1$) do
    OFFSET $= (2 * i) \bmod N$;
    for ($j = 0$ to NCOL$' - 1$) do
      $k = (j + \text{OFFSET}) \bmod N$;
      Assign $\text{DC}_{B(i,j)}$ to sub-image k;
      for ($m = 1$ to 63) do
        Assign $\text{AC}^m_{B(i,j)}$ to sub-image
          $(k + m) \bmod N$;
      end for
    end for
  end for
  for each sub-image do
    DPCM encoding of the DC coefficients;
    Run-length encode the AC coefficients;
    Huffman encode the resultant stream;
  end for
end.

Procedure LRJ-Decompress
begin
  Input: N sub-images;
  for each sub-image do
    Huffman decode the bit stream;
    Run-length decode AC coefficients;
    Inverse DPCM for DC coefficients;
  endfor
  for ($j = 0$ to NROW $- 1$) do
    OFFSET $= (2 * j) \bmod N$;
    for ($i = 0$ to NCOL$' - 1$) do
      $k = (i + \text{OFFSET}) \bmod N$;
      if sub-image k is available
        $\text{DC}_{B(i,j)} \leftarrow$ next DC coefficient
            from sub-image k;
      else
        Derive $\text{DC}_{B(i,j)}$ from Equation 6.2;
      fi
      for ($m = 1$ to 63) do
        if sub-image $(k + m) \bmod N$ is available
          $\text{AC}^m_{B(i,j)} \leftarrow m^{th}$ AC coefficient from
          sub-image $(k + m) \bmod N$;
        else
          $\text{AC}^m_{B(i,j)} \leftarrow 0$;
        fi
      end for
    end for
  end for
  Perform inverse quantization of each 8x8 block;
  Perform inverse DCT on each 8x8 block;
end.

Figure 6.5: LRJ compression and decompression algorithms

DC coefficients of the neighboring 8x8 blocks. Lost AC coefficients, on the other hand, are replaced by zeroes. Since the scrambling function employed by the encoder ensures that each scrambled block contains coefficients from several blocks of the original image, the artifacts yielded by such a recovery mechanism are dispersed over the entire reconstructed image, thereby significantly improving the visual quality of the image. Figure 6.4 depicts the various modules involved in the LRJ compression and decompression algorithm, and Figure 6.5 describes both of these algorithms in detail.

As a final note, observe that since successive blocks within a sub-image do not belong to the 8-neighborhood of each other, the correlation between their DC coefficients is smaller than the neighboring DC coefficients in the original image. The reduced correlation diminishes the efficiency of DPCM encoding of DC coefficients, and hence increases the total size of the compressed image (as compared to the corresponding JPEG image). Scrambling AC coefficients of a block across several sub-images, on the other hand, does not have any significant impact on the compressed image size. This is because, due to the very nature of DCT, AC coefficients are uncorrelated. Moreover, quantization yields a large number of zero coefficients. Since the scrambling algorithm does not alter the relative position of an AC coefficient within the zig-zag ordering, the effect of scrambling on the efficiency of run-length and huffman encoding is minimal. Thus, the increase in compressed image size yielded by the LRJ algorithm can be mostly attributed to the need for replicating huffman tables and the reduced correlation of successive DC coefficients.

6.1.4 Loss-Resilient MPEG (LRM) Algorithm

The MPEG compression standard exploits the large temporal and spatial redundancies present within a video file to achieve a high degree of compression [32, 42]. MPEG supports three kinds of frames : (1) I frames (intra-coded without any reference to other frames), (2) P frames (predictively coded using an I or P frame), (3) B frames (bidirectionally interpolated from both the previous and the following I and/or P frame). To derive these types of frames, MPEG groups image data into 16x16 pixel areas called macro blocks. Macro blocks belonging to I frames are independently encoded. Macro blocks belonging to B and P frames, on the other hand, are temporally interpolated from corresponding reference frame(s), and the error macro block is computed as the difference between the actual and interpolated blocks. Macro blocks for which the encoder is unable to find a good reference block are intra-coded. The interpolation process also produces up to two motion vectors for each macro block, which denote the positions of the interpolated macro blocks in the reference frames. Regardless of the frame type, each macro block is then partitioned into six 8x8 pixel blocks — four luminance and two chrominance blocks. Each 8x8 pixel block is transformed into the frequency domain using the DCT. The DC coefficients of successive blocks are difference encoded. The AC coefficients within a block are quantized, scanned in a zig-zag manner, and finally, run-length and entropy encoded. The motion vectors in P and B frames are also difference and entropy encoded.

Thus, while JPEG exploits only the spatial redundancies present within images, MPEG exploits both temporal and spatial redundancies present in image sequences. The key difference between the loss-resilient MPEG (LRM) algorithm and the LRJ algorithm is that LRM must also recover lost motion vectors of a macro-block in addition to lost DCT blocks. In order to recover lost DCT blocks, the LRM algorithm uses a partitioning algorithm similar to that in the LRJ algorithm. Since the least unit of encoding in MPEG is a macro block, the partitioning algorithm operates on macro blocks rather than DCT blocks. To precisely describe the LRM algorithm, let $\mathcal{B}^i(x,y)$ denote the i^{th} DCT block, $0 \leq i \leq 5$, in the macro block located at (x,y), and let the first four DCT blocks with each macro blocks denote luminance blocks. The algorithm proceeds in two steps:

1. *Scrambling:* Given N macro blocks from the same row of the image, the scrambling function takes the i^{th} DCT block of each macro block and scrambles them such that the AC coefficients of each block are uniformly distributed among the output blocks. That is,

$$f(\mathcal{B}^i(x,y), \mathcal{B}^i(x+1,y), \cdots \mathcal{B}^i(x+N-1,y)) = \mathcal{B'}^i(x,y), \mathcal{B'}^i(x+1,y), \cdots \mathcal{B'}^i(x+N-1,y)$$

2. *Sub-image creation:* The partitioning algorithm assigns DCT blocks to sub-images such that none of the DC coefficients in the 8-neighborhood of a block belong to the same sub-image. Since each macro block contains blocks from the luminance as well as both chrominance components, the above property must hold for all three components of an image. To ensure this property, the partitioning algorithm partitions each image as follows: (1) block $\mathcal{B'}^0(x,y)$ is assigned to sub-image $\mathcal{I}_{(2x+4y)\ mod\ N}$, (2) if block $\mathcal{B'}^0(x,y)$ is assigned to sub-image $\mathcal{I}_k$, then block $\mathcal{B'}^i(x,y)$ is assigned to sub-image $\mathcal{I}_{(k+i)\ mod\ N}$, $0 \le i \le 3$, and (3) blocks $\mathcal{B'}^4(x,y)$ and $\mathcal{B'}^5(x,y)$ are assigned to sub-images $\mathcal{I}_{(x+2y)\ mod\ N}$ and $\mathcal{I}_{(x+2y+1)\ mod\ N}$, respectively. Figure 6.6 illustrates the sub-image creation process.

While a lost DC coefficient can be extrapolated from its neighboring DCT blocks, extrapolating a lost motion vector from neighboring macro blocks yields poor results due to the small correlation between motion vectors of adjacent macro-block within an image. Furthermore, if the MPEG encoder is unable to temporally interpolate a macro block from its reference frames, then the block is intra-coded. Thus, if the neighboring macro blocks of a macro block are intra-coded, there won't be any motion vectors to extrapolate from. Hence, to enable recovery even in such scenarios, the loss-resilient MPEG (LRM) algorithm must replicate motion vectors of macro blocks. In our algorithm, the motion vectors of a macro block stored in sub-image $\mathcal{I}_k$ are replicated in sub-image $\mathcal{I}_{k+1}$. Since MPEG allows applications to store any application-specific data in the user-defined section of the MPEG stream, such replication can be achieved without violating the syntax of MPEG. The primary motion vectors, on the other hand, are stored with their respective macro blocks, as dictated by the syntax of MPEG. Besides replicating motion vectors, the LRM algorithm also replicates the header information of each macro block in the user-defined section of a sub-stream. The macro block header contains information such as whether the macro block is intra-coded, predictively coded, or bidirectionally interpolated, which is needed by the decoder to decode the macro block. Such header information can be efficiently replicated since only a few bits per macro-block are needed to encode the information.

In the event of a disk failure, the recovery process operates as follows: (1) DC coefficients of lost DCT blocks are extrapolated from neighboring DCT blocks, (2) lost AC coefficients are substituted by zeroes, and (3) lost motion vectors are recovered from the replicas stored in the user-defined sections of the surviving sub-streams. Observe that, since neighboring macro blocks can be encoded differently, a DC coefficient must be extrapolated from only those neighboring DCT blocks with the same encoding type. For instance, a DC coefficient contained in an intra-coded macro block must be extrapolated from only those neighboring DCT blocks belonging to intra-coded macro blocks. Thus, just as in the LRJ algorithm, a reasonable reconstruction of the image can be obtained by exploiting the inherent redundancies within the video file. However, due to the dependencies between frames in MPEG, errors due to imperfect recovery in a frame can get propagated to other frames. That is, imperfect recovery of an I frame macro block can cause artifacts to appear in the blocks within the B and P frames which have been interpolated from it. The impact of such error propagation on the visual quality can be reduced by choosing a larger value of N, thereby reducing the fraction of the data lost per image. As we shall see in Section 6.4, the loss in compression efficiency caused by choosing larger values of N (as compared to LRJ) is not high.

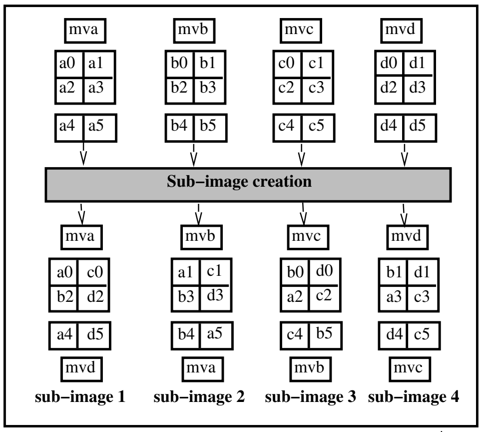


Figure 6.6: Sub-image creation in LRM: $a0$ through $a5$ represent the six DCT blocks of macro block a and mva represents its motion vector. Successive DCT blocks within the same row of the luminance component (e.g., $a0, a1, b0, b1, ...$) are assigned to successive sub-images in a round-robin manner. Successive blocks within the same column (e.g., $a0$ and $a2$) are assigned to sub-images that are offset by 2 from each other. Successive blocks of each chrominance component (e.g., $a4, b4, c4...$) are mapped to sub-images in a round-robin manner.

6.1.5 Inherently Redundant Array of Disks (IRAD)

The LRJ or the LRM compression algorithms, when applied to a sequence of images constituting a video stream yield N sequences of sub-images. We refer to each such sequence as a *sub-stream*. For effective recovery, the integrated file system must ensure that the disks over which the sub-streams are striped do not overlap (i.e., even in the presence of a single disk failure, at least $(N-1)$ sub-streams are available). This can be achieved by striping each sub-stream over a mutually exclusive subset of disks in the array. We refer to a disk array architecture that employs such a placement strategy as an *Inherently Redundant Array of Disks (IRAD)*. In the event of a disk failure, clients use the LRJ or the LRM algorithm to approximately reconstruct lost image data. A careful analysis of this process of recovering from disk failures illustrates the following salient characteristics of the IRAD architecture:

- Since each image in the video file is reconstructed by extrapolating information retrieved from the surviving disks, the failure recovery process does not impose any additional load on the disk array. Consequently, the number of clients that can be serviced simultaneously by an integrated file system will be constrained solely by the playback rate requirements of video clients during the fault-free state. Moreover, operating the server at very high levels of utilization during the fault-free state will not present any risk of saturation in the presence of failure.

- Since the recovery of lost image data is integrated with the decompression algorithm, the reconstruction process is carried out at client sites. This is an important departure from the conventional RAID architectures — distributing the functionality of failure recovery to client sites will significantly enhance the scalability of integrated file systems.

- Since the recovery process only exploits the inherent redundancy in imagery, client sites will be able to reconstruct a video file even in the presence of multiple disk failures. The quality of the reconstructed image, albeit, will degrade with increase in the number of simultaneously failed disks (i.e., IRAD supports graceful degradation in the image quality with increase in the number of failed disks).[2]

- Since the cause of the data loss is irrelevant to the recovery algorithm, the unscrambling algorithms in LRJ and LRM can be adapted to mask packet losses due to network congestion as well.[3]

Thus, the IRAD architecture provides an integrated, scalable, end-to-end solution for failure recovery.

In case of a disk failure, a redundant array must: (1) perform on-line reconstruction and thereby provide uninterrupted service to user requests, and (2) rebuild the failed disk onto a spare disk so that the array can revert back to the normal operating mode. Whereas the LRJ and LRM algorithms can be used to achieve the former objective in IRAD, rebuild can be accomplished either by restoring the failed disks from tertiary storage (i.e., tape backups), or by employing parity-based techniques. To rebuild failed disks from tertiary storage, the controller must allow a user to backup and restore each disk individually. Since rebuild of a failed disk from tertiary storage can operate independently of other disks in the array, the rebuild operation does not impose any additional load on the surviving disks. Alternatively, if parity information is used to rebuild the contents of the failed disk, then on-line rebuild onto a spare disk can proceed simply by issuing low-priority read requests to access media blocks from each of the surviving disks [39]. Note that the array controller still depends on the LRJ/LRM algorithms for *online reconstruction* of user requested images while the parity information is used for *online rebuild* of the failed disk. The choice of a particular rebuild technique is environment dependent: whereas rebuild from tertiary storage will suffice for predominantly read-only environments, parity-based rebuild will be required for environments with frequent writes.

6.2 Exploiting Sequentiality of Continuous Media Retrieval

In the previous section, we presented failure techniques that approximately reconstruct data stored on failed disks. While approximate reconstruction is adequate for most continuous media applications, certain demanding continuous media applications require perfect reconstruction of data (e.g., a video clip showing a CAT scan of a brain tumor). For such applications, we present a failure recovery scheme that exploits the sequential nature of continuous media accesses to perfectly recover data stored on failed disks without imposing a large overhead on an integrated file system.

6.2.1 Parity-based Reconstruction

Consider an integrated file system that stripes continuous media files across a disk array. In addition to a sequence of data blocks for each continuous media file, to recover from disk failures, the file server maintains parity blocks on the array. Parity blocks are computed by an exclusive-or operation over all data blocks within the parity group.

To describe the fault-free operation, consider a scenario where the server is servicing n clients, each retrieving a continuous media file (say $S_1, S_2, ..., S_n$, respectively). Let $\mathcal{R}_i$ denote the playback rate of file S_i, $1 \le i \le n$. Due to the periodic nature of continuous media playback, an integrated file system services these clients by proceeding in *rounds*. During each round, the server retrieves a fixed number of media units for each client. Thus, if $\mathcal{T}$ denotes the

[2]Observe that, to tolerate multiple disk failures, multiple copies of huffman tables and motion vectors would have to be maintained.

[3]Several techniques have been proposed which scramble video files prior to network transmission to enable approximate reconstruction in case of packet losses. [27, 70]. The efficacy of these techniques validates our claim.

duration of a round, then to ensure continuous playback, the number of media units of files $S_1, S_2, ..., S_n$ retrieved during each round is given by: $f_i = \mathcal{T} * \mathcal{R}_i; \quad \forall i \in [1, n]$. To access $f_1, f_2, ..., f_n$ media units, the server will be required to retrieve $k_1, k_2, ..., k_n$ data blocks, respectively, of files $S_1, S_2, ..., S_n$,. If each file is encoded using a variable bit rate (VBR) compression algorithm, then the sizes of media units will vary. Hence, the mapping from f_i to k_i may vary from one round to another. For files encoded using a constant bit rate (CBR) compression algorithm, on the other hand, the mapping from f_i to k_i is fixed. Regardless of the actual compression algorithm, during the fault-free state, a server only retrieves data blocks and skips over all the parity blocks.

In the presence of a disk failure, when a client requests the retrieval of a block stored on the failed disk, the server must access the parity and data blocks stored on the surviving disks to recover the lost information. In the simplest case, the server retrieves blocks of the parity group to recover lost information, transmits the set of requested blocks to the client site, and then discards the additional blocks. Although relatively simple to implement, the transient increase in load on disks induced by such a scheme may yield service times (i.e., the total time spent in retrieving images during a round) that exceed $\mathcal{T}$, resulting in playback discontinuities at client sites.

Alternatively, consider a file server that always computes parity over a sequence of $(G-1)$ data blocks from the *same* continuous media file. That is, all $(G-1)$ data blocks within a parity group are successive blocks of the same file. In such a scenario, the server can exploit sequentiality of continuous media playback to minimize the increase in load due to a failure by using data blocks retrieved during playback for failure recovery and vice-versa. Observe that the server can recover blocks stored on the failed disk either in the round in which they are accessed, or at least one round prior to their access. These recovery policies are referred to as *lazy* and *eager* failure recovery, respectively. In what follows, we present these policies in detail and discuss their tradeoffs.

To describe the lazy failure recovery algorithm, consider a disk array with a parity group size of G. Let the set of disks within a parity group be denoted as $1, 2, ..., G$, and let us assume that, disk $i (i \leq G)$ fails in round r. In such a scenario, for each block accessed from the failed disk during round r, the server will access an additional block from each of the surviving $(G-1)$ disks in the parity group. Then, due to sequentiality of continuous media playback, a subset of the $(G-1)$ blocks (namely, blocks from disks $(i+1)$ through G) accessed for recovering the lost block will be requested by the client within the next few rounds. The server can buffer these blocks and service requests for their access from the buffer. Consequently, an increase in load on disks $(i+1)$ through G during round r due to a client will be followed by a corresponding reduction in load due to that client in the next few rounds. Observe that, while blocks from disks $(i+1)$ through G are reused for servicing retrieval requests, blocks from disks 1 through $(i-1)$ are not. Hence, these blocks must be retrieved for reconstructing lost blocks and subsequently discarded. However, due to the sequential nature of playback, the client would have accessed blocks on disk 1 through $(i-1)$ in the past few rounds for normal playback. Hence, the server can further reduce the overhead of failure recovery by maintaining an exclusive-or of these blocks when the client accesses them for playback. The server can then use the result of this exclusive-or computation for reconstructing the block on disk i, instead of retrieving these blocks again from the array. We refer to the result of this exclusive-or computation as *online parity*. The server can maintain online parity only in the event of a disk failure or even in the fault-free state. Since a disk failure can not be anticipated in advance, maintaining online parity even in the fault free state minimizes recovery overhead. On the other hand, since parity computations can be expensive, maintaining parity only in the event of a disk failure and only for the parity group containing the failed disk significantly reduces parity computation overheads. However, this approach yields a transient increase in the load on disks 1 through $(i-1)$ immediately after a disk failure, since additional blocks must be accessed from these disks to reconstruct lost blocks.

```
Procedure OnlineParity
begin
    Let R_j denote the set of k_j blocks to be retrieved for client j.
    for every block b in set R_j do
        if b belongs to the same parity group as the blocks over which
        online parity p_j has been computed then
            p_j = p_j ⊕ b
        else
            p_j = b {* new parity group starts with b *}
        fi
    end for
end.
```

Figure 6.7: Procedure for computing online parity

The exact algorithms for maintaining online parity and lazy failure recovery are described in Figures 6.7 and 6.8, respectively. The lazy recovery algorithm presented in Figure 6.8 assumes that online parity is maintained even in the fault-free state, and can be easily modified if online parity is maintained only in presence of disk failures.

In the eager failure recovery algorithm, on the other hand, instead of recovering blocks stored on a failed disk only when they are accessed, the file server exploits the sequentiality of continuous media playback to prefetch data blocks so that blocks on the failed disk are recovered sufficiently prior to their access. To precisely describe the recovery algorithm, let $k_j^{\max}$ and $k_j^{\min}$, respectively, denote the maximum and the minimum number of media blocks requested by client j in a round. Then the following theorem determines the number of blocks that must be prefetched to reconstruct blocks on the failed disk prior to their access.

Theorem 6.2 By maintaining $(k_j^{\max} + G - 2)$ blocks per client in the buffer, the server can ensure the reconstruction of lost data blocks at least one round prior to their access.

Proof: To reconstruct a data block stored on the failed disk, all the blocks in its parity group must be accessed. Thus, in the worst case, if $k_j^{\max}$ blocks are requested by client j from the server during a round, and if the last of these blocks is stored on the failed disk and none of the remaining $(k_j^{\max} - 1)$ requested blocks belong to its parity group, then the server will require $(G - 1)$ additional blocks to recover the lost block. Hence, if the server ensures that $\Big((k_j^{\max} - 1) + (G - 1)\Big) = (k_j^{\max} + G - 2)$ blocks have been prefetched into its buffer, then it can reconstruct any lost block at least one round prior to its access. ■

In the simplest case, regardless of the presence or absence of failures, an integrated file system can prefetch $(k_j^{\max} + G - 2)$ blocks per client prior to initiating playback. In such a scenario, if a client requests the retrieval of k_j blocks in a round, then the server transmits these blocks from its buffer and retrieves the next k_j blocks from the array so that $(k_j^{\max} + G - 2)$ blocks are always in its buffer. By maintaining $(k_j^{\max} + G - 2)$ blocks per client, the server ensures that an entire parity group is retrieved and buffered at least one round before any of its blocks are accessed. Hence, data blocks on the failed disk can always be reconstructed prior to their access. The disadvantage of this approach is the increase in initiation latency experienced by clients due to the prefetch operation. Alternatively,

```
Procedure LazyRecovery
begin
  Let R_j denote the set of k_j blocks to be retrieved for client j and p_j denote
  its online parity block.
  for each client j do
     if R_j consists of a block on the failed disk i then
        Let b_l denote the lost block
        Let Q_j consist of data blocks from disks (i + 1) through G for client j
        Retrieve blocks R_j − {b_l}.
        Retrieve blocks Q_j − R_j. {* Get remaining blocks of the
                                     parity group *}
        Retrieve the parity block p
        b_l = p_j ⊕ p ⊕ blocks of the parity group in Q_j ∪ R_j
        Buffer blocks in Q_j − R_j for future rounds.
     else
        Retrieve blocks in R_j (do not retrieve blocks present in the buffer).
     fi
     Compute online parity p_j over R_j
     Schedule blocks of R_j for transmission to client j.
  end for
end.
```

Figure 6.8: Lazy Failure Recovery Algorithm

Procedure EagerRecovery
begin
for each client j do
  (1) Suspend transmission of blocks, and prefetch $(k_j^{\max} + G - 2)$ media blocks
  (2) If the first requested block in R_j is on the m^{th} disk of the parity group of the failed disk and $m \leq i$ then
      Retrieve additional data blocks from disks 1 though $m - 1$.
      Retrieve the parity block and reconstruct block on disk i.
      Discard blocks retrieved from disks 1 through $m - 1$.
    end if.
  (3) Resume transmission of data blocks to the user.
  (4) If in a round, k_j prefetched blocks are transmitted to the user then
      Retrieve the next k_j blocks from the disk array so that $(k_j^{\max} + G - 2)$ blocks are always in the buffer.
      If a requested block belongs to a failed disk then
        Retrieve the corresponding parity block instead.
      end if.
    end if.
  (5) Reconstruct lost blocks as soon as all remaining blocks within its parity group have been read into the buffer.
end for.
end.

Figure 6.9: Eager failure recovery algorithm

the server can maintain $(k_j^{\max} + G - 2)$ blocks per client only in the event of a disk failure. In this case, assuming that the i^{th} disk $(i \leq G)$ of the parity group fails in round r, the server suspends transmission of data blocks to clients until $(k_j^{\max} + G - 2)$ blocks are prefetched for each client. Observe that, if the server is lightly loaded, then the duration for which transmission (and hence, the playback) is required to be suspended is relatively short. In the worst case, however, the prefetch operation may take up to $\left\lceil \frac{k_j^{\max} + G - 2}{k_j^{\min}} \right\rceil$ rounds. In addition to these blocks, if the first of the requested blocks for a client in round r is stored on disk m of the parity group, and if $m \leq i$, then the server must retrieve additional blocks from disks 1 to $(m - 1)$ to reconstruct the lost block. These additional blocks are discarded by the server after reconstructing the block on disk i. On resuming transmission (and hence, playback), the server retrieves sufficient number of blocks in each round so as to ensure that the $(k_j^{\max} + G - 2)$ blocks per client are always in buffer. Observe that this approach shifts the latency from playback initiation time to the time when the server experiences a disk failure. Since disk failures are infrequent events, most clients will not experience this latency in the common case. However, a disadvantage of the approach is the temporary pause in playback that clients experience immediately after a disk failure. The precise eager failure recovery algorithm is described in Figure 6.9. The algorithm presented in Figure 6.9 assumes that media blocks are prefetched only in the event of a disk failure, and can be easily modified if blocks are prefetched prior to initiating playback.

Table 6.1: Overhead of the lazy/eager algorithms

	Standard Array	Array with lazy/ eager recovery
RAID level 4	Data Disks: 100 % increase Parity Disk: same load as the failed disk prior to fault	Data Disks: No increase Parity Disk: same load as the failed disk prior to the fault
RAID level 5	$\frac{G-1}{G-1} = 100$ % increase	$\frac{1}{G-1}$
Declustered parity	$\frac{G-1}{C-1}$	$\frac{1}{C-1}$
Flat parity	Data Disks: 100% increase Parity Disk: $\frac{1}{C-(G-1)}$	Data Disks: No increase Parity Disk: $\frac{1}{C-(G-1)}$

6.2.2 Failure Recovery Overheads

Whereas in standard RAID each data block within the parity group would be accessed twice in the worst case, once for playback and once for reconstructing the lost block, in the lazy and eager schemes each media block is accessed precisely once. Thus, the only additional load on the disks is due to the retrieval of parity blocks, thereby considerably reducing the overhead of failure recovery. Recall that, the lazy scheme causes a sequence of load fluctuations since an increase in load due a client (caused by accessing a block on the failed disk) is followed by a reduction in load due to that client in the following rounds. Hence, the fundamental difference between the lazy and the eager recovery algorithms is that the latter trades buffer space to replace a sequence of load fluctuations possible in the former by a constant increase in the load.

Observe that, both the lazy and the eager recovery schemes make no assumptions about the array architecture, and hence can be used with many different architectures. In what follows, we analyze the overhead of these schemes for various architectures including RAID level 4, RAID level 5, declustered parity, etc., and compare it with standard recovery techniques.

Assuming that the load on each disk is balanced prior to a failure, the recovery overhead for different architectures is shown in Table 6.1. Since the only additional blocks that are retrieved by the lazy and eager schemes are parity blocks, the data disks in a RAID level 4 array experience no increase in the load. The parity disk, however, experiences a load equal to that of the failed disk prior to failure (since every access to the failed disk causes an access on the parity disk). For RAID level 5 arrays, since parity blocks are uniformly distributed among all disks in the parity group (see Figure 6.1(a)), the recovery overhead is $1/(G-1)$. This is a significant reduction over the 100% load increase seen by each disk in standard RAID level 5 arrays. In declustered parity arrays, since parity blocks are uniformly distributed across the $(C-1)$ surviving disks within the cluster, the recovery overhead is $1/(C-1)$. Lastly, consider a uniform flat parity placement scheme in which: (1) each cluster is partitioned into groups of $(G-1)$ disks (i.e., $C = n \cdot (G-1),\ n = 1, 2, ..$), and (2) each group of $(G-1)$ disk stores the data blocks of a parity group with the parity block uniformly distributed among the remaining $C-(G-1)$ disks within the cluster. In the presence of a failure, while the $(G-1)$ disks storing data blocks see no increase in the load, the remaining $(C-(G-1))$ disks see a load increase due to retrieval of parity blocks. Thus, the overhead of failure recovery on these disks is $1/(C-(G-1))$.

6.2.3 Discussion

Recently a scheme similar to the lazy recovery algorithm was proposed in [9]. In this scheme, on experiencing a disk failure, the cluster with a failed disk switches to the degraded mode with requests reading and buffering entire parity groups at a time. Thus, the entire cluster acts like a single logical disk. In the lazy recovery algorithm, accessing a block on the failed disk i causes blocks from disk $(i+1)$ through G to be retrieved. Thus, disks $(i+1)$ through G act as a single logical disk, and only blocks of the parity group retrieved from these disks need to be buffered. Whereas both schemes have identical worst case buffer requirements, the lazy scheme has a lower average case buffer requirement.

Recall that, both the lazy and the eager failure recovery algorithms require that all data blocks contained within a parity group belong to the same file. However, most of the existing array controllers provide the abstraction of a single large disk addressable by logical blocks numbers to the operating system software. Thus, details such as the logical to physical block mapping, membership of a parity group, etc., are implemented by the controller logic and are hidden from the operating system. Without these details, an integrated file system can not determine the block numbers of data blocks constituting a parity group, and hence can not control the membership of media blocks within a parity group. Consequently, to implement our failure recovery algorithms, conventional array controllers must be suitably extended. Specifically, if the controller can implement a function that takes a logical block number as its input and returns a list of logical block numbers of all blocks which belong to its parity group, then an integrated file system can exercise precise control over membership of blocks within a parity group.

Since a continuous media workload is dominated by read requests, the preceding discussion focussed on the overhead of read requests and ignored write requests. We assume that standard techniques such as full-stripe writes in which entire parity groups are written at a time will be used for the large sequential writes seen in a continuous media workload. Regardless of the presence or absence of failures, in full-stripe writes, the server computes the new parity information and writes it to the appropriate disk along with the data blocks in the parity group. In case of a disk failure, the server either discards the block to be stored on the failed disk, or writes it to a replacement disk. Since the array operation is unchanged in the presence of failures, these full-stripe writes impose no extra overhead on the server as compared to the fault-free state.

6.3 Comparative Evaluation

A comparison of the schemes presented in this chapter with standard RAID level 5 and declustered parity arrays is shown in Table 6.2. The array architectures are compared with respect to their storage space overhead, overhead imposed by to failure recovery, buffer space requirements, and mean time to data loss (MTTDL).

Storage Space Overhead

To compute the storage space required to maintain parity information, consider a disk array with a cluster size of C. Then, the storage space overhead of a RAID level 5 array is $1/C$, while that of a declustered parity array is $1/G$. Since $G < C$, declustered parity arrays have a higher overhead as compared to RAID level 5 arrays. For the IRAD architecture, the storage space overhead is primarily due to the loss in compression efficiency during image partitioning in the LRM and LRJ algorithms. The replication of motion vectors adds to this overhead in the LRM algorithm. The overhead increases initially with increase in the degree of image partitioning, and then becomes

Table 6.2: Comparative evaluation of fault-tolerant schemes

	RAID-5/Declustered Parity (Standard)	Declustered Parity (Lazy/Eager)	IRAD
Storage Overhead	RAID-5: $1/C$ Declustered: $1/G$	$1/G$	depends on N
Load Overhead	$(G-1)/(C-1)$	$1/(C-1)$	0
Buffer (fault-free)	$nbk^{\max}$	eager: $nbk^{\max}$ lazy: $nb(k^{\max}+1)$	$nb'Nk'^{max}$
Buffer (fault)	$nb[I(k^{\max}+G-1)$ $+(D-I)k^{\max}]/D$	eager: $nb(k^{\max}+G-1)$ lazy: depends on $k^{\min}$	$nb'Nk'^{\max}$
MTTDL	$\frac{MTTF^2}{D(C-1)MTTR}$	$\frac{MTTF^2}{D(C-1)MTTR}$	depends on the rebuild algorithm

independent of the degree of image partitioning for large values of N. Finally, if the IRAD architecture rebuilds failed disks using parity information rather than from tertiary storage, then it incurs an additional overhead of $1/C$.

Failure Recovery Overhead

Recall that, the recovery overhead due to read requests for RAID level 5 and declustered parity arrays is $(G-1)/(G-1)$ and $(G-1)/(C-1)$, respectively. As shown in Section 6.2, if the lazy and eager recovery algorithms are used, this overhead reduces to $1/(G-1)$ and $1/(C-1)$ for RAID level 5 and declustered parity arrays, respectively. For the IRAD architecture, on the other hand, since the array operation is unchanged even in the presence of a failure, there is no overhead due to failure recovery. However, this is at the expense of a slightly higher load in the fault free state caused by the degradation in compression efficiency due to image partitioning in LRJ and LRM.

Buffer Requirements

To compute the buffer overhead, let us assume that the load on each disk in the array is balanced in the fault-free state. Let n denote the total number of clients accessing the disk array, and let b denote the media block size. Since in the worst case, each client accesses $k^{\max}$ blocks in the fault-free state, the total buffer requirement for RAID level 5 and declustered parity arrays is $nbk^{\max}$. Assuming that the eager recovery scheme prefetches media blocks only in the event of a failure, its fault-free buffer requirement is $nbk^{\max}$. Similarly, assuming that the lazy scheme maintains online parity even in the fault free state, it incurs a buffer requirement of $nb(k^{\max}+1)$. In the IRAD architecture, assuming that each client accesses $k'^{\max}$ blocks of size b' from each of the N sub-streams during a round, the total buffer required is $nb'Nk'^{\max}$ ($k'^{\max}$ and b' can be distinct from those for parity-based arrays). While choosing $b'=b$ yields an array utilization that is comparable to parity-based arrays, it increases the buffer space requirement. On the other hand, choosing $b'<b$ lowers the buffer required for IRAD architectures at the expense of a lower array utilization. Hence, the block size must be chosen to balance these tradeoffs.

To compute the buffer requirement in the presence of disk failures, let us assume that I clusters in the array have experienced a single disk failure ($I \leq D/C$). Further, assume for simplicity, that a client accesses at most one failed disk in each round. For RAID level 5 and declustered parity arrays, in the worst case, each client accessing a block on the failed disk would access $(G-1)$ additional blocks. Since each disk is accessed by n/D clients and the array contains I failed disks, the total buffer required is $nb(k^{\max}+G-1)I/D+nbk^{\max}(D-I)/D$. In the eager recovery scheme, the server buffers $(k^{\max} + G - 2)$ blocks per client and requires an additional block per client to store the reconstructed block. Hence, the total buffer required is $nb(k^{\max} + G - 1)$. The buffer requirement for lazy recovery is dictated by the following lemma:

Lemma 6.1 The worst case buffer requirement for the lazy failure recovery algorithm is

$$\begin{array}{ll} nb(k^{\max} + G - 1)I/D + nbk^{\max}(D - I)/D, & \text{if } (G-2) \leq k^{\min} \\ ((G-2)(j+1) + k^{\max} - k^{\min} j(j-1)/2)Inb/D + & \\ (n - (j+1)In/D)bk^{\max} & \text{otherwise} \end{array}$$

where $k^{\min}$ denotes the minimum number of blocks accessed by a client in a round, $k^{\min} \geq 1$, and $j = \lfloor \frac{G-2}{k^{\min}} \rfloor$.

Proof: In the lazy scheme, when a client requests a block on the failed disk i, the server accesses and buffers blocks stored on disks $(i+1)$ through G of the parity group. Then in the worst case, the server will buffer $(G-2)$ data blocks per client and these blocks will be used to service requests in the following $(\frac{G-2}{k^{\min}})$ rounds. If $(G-2) \leq k^{\min}$, then all the $(G-2)$ buffered blocks would be accessed by the client in the following round. In such a scenario, the total buffer required would be the same as that in a RAID 5 array. That is, total buffer required is $nb(k^{\max} + G - 1)I/D + nbk^{\max}(D - I)/D$.

On the other hand, if $(G-2) > k^{\min}$, then in the worst case, the server uses the buffered blocks to service client requests for up to $j = \lfloor \frac{G-2}{k^{\min}} \rfloor$ rounds after the client has accessed a block on the failed disk. To compute the buffer requirement in this scenario, consider the set of clients who have accessed a failed disk in the current round or the any of previous j rounds. The buffer required for these clients is

$$[(k^{\max} + G - 2) + (G-2) + (G - 2 - k^{\min}) + (G - 2 - 2k^{\min}) \cdots + (G - 2 - (j-1)k^{\min})]Inb/D$$

Simplifying, this yields

$$[k^{\max} + (G-2)(j+1) - k^{\min}(1 + 2 + \cdots + (j-1))]Inb/D$$

or

$$[k^{\max} + (G-2)(j+1) - k^{\min} j(j-1)/2]Inb/D$$

The buffer required for the remaining clients is $[n - (j+1)In/D]bk^{\max}$. Thus, when $k^{\min} < (G-2)$, the total buffer required is

$$[k^{\max} + (G-2)(j+1) - k^{\min} j(j-1)/2]Inb/D + [n - (j+1)In/D]bk^{\max} \tag{6.3}$$

■

Lastly, for the IRAD architecture, since no additional blocks are accessed by a client in the presence of failures, there is no increase in the buffer requirement.

Mean Time To Data Loss

RAID level 5, declustered parity, and the lazy/eager recovery based disk arrays experience data loss if a disk within a cluster fails while another disk within that cluster is being rebuilt. The mean time to data loss (MTTDL) for these architectures is given as

$$MTTDL = \frac{MTTF^2}{D(C-1)MTTR} \tag{6.4}$$

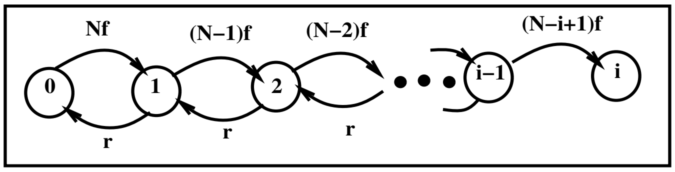


Figure 6.10: Markov model for determining MTTS. State j represents j failed disks in a cluster. With j failed disks in a cluster, the failure rate is $(N-j)f$, and the repair rate is r. The system is taken offline when it reaches state i.

where $MTTF$ is the mean time to failure for an individual disk, and $MTTR$ is the mean time to rebuild a failed disk [19]. To illustrate, consider an array of 32 disks and a cluster size of 8. If the $MTTR$ of a disk is 2 hours and its $MTTF$ is 300,000 hours, then the mean time to data loss for the array is about 23,000 years. Since the $MTTR$ of a disk is proportional to the load on the array during the online rebuild process, and since the lazy/eager recovery schemes impose a lower recovery overhead on the array, they have a higher $MTTDL$ than standard RAID level 5 or declustered parity arrays.

The $MTTDL$ for the IRAD architecture, on the other hand, depends on the rebuild algorithm used by the array. If parity information is used to rebuild failed disks, then the MTTDL is given by Equation 6.4. Since a disk failure imposes no additional load on the surviving disks, the IRAD architecture has a lower $MTTR$ for a failed disk, and hence, a higher $MTTDL$ as compared to RAID level 5 or declustered parity architectures. For IRAD arrays that rebuild failed disks from tertiary storage, it is not meaningful to compute $MTTDL$ since lost data can always be recovered from backup tapes. Hence, we define a new metric to compute the resilience of the array to failures. The mean time to shutdown ($MTTS$) for the IRAD architecture is defined as the average time before the array is taken offline for repairs. Since the architecture supports graceful degradation in the image quality in the presence of multiple failures, the array must be taken offline when i disks fail within any group of N disks and the resulting image quality is too poor to be acceptable. Let f denote the failure rate of a single disk (i.e., $f = 1/MTTF$), and let r denote the rate of repair of a disk from tertiary storage (i.e., $r = 1/MTTR$). Then the IRAD architecture that tolerates $i-1$ disk failures per cluster can be modeled as an $i+1$ state Markov chain as shown in Figure 6.10. The mean time to shutdown can be either computed analytically using the theory of Markov chains [85], or computed numerically using tools such as SHARPE [75]. In the simplest case, where the array can tolerate two failures per cluster (i.e., $i = 3$), $MTTS = MTTF^3/[D(C-1)(C-2)MTTR^2]$. Thus, when $D = 32$, $N = C = 8$, and $MTTR = 3$ hours, the $MTTS$ is over 250 million years.

To summarize, the lazy/eager recovery based arrays and the IRAD architecture have a storage space overhead and MTTDL that is comparable to conventional arrays. However, they have a lower failure recovery overhead and a higher buffer requirement as compared to conventional arrays. Thus, our approaches trade buffer space for lower recovery overhead. Whereas lazy/eager recovery scheme may be chosen for perfect recovery of images, the IRAD architecture may be chosen for its advantages such as tolerance to multiple disk failures, resilience to network losses, etc. The choice of a particular recovery scheme depends on the application requirements.

Table 6.3: Characteristics of MPEG traces

File	Encoding	Pattern	Length (frames)	Average bit rate (Mb/s)	Motion
Frasier	MPEG	$I(BBP)^3BB$	6000	1.498	Moderate
Ice Hockey	MPEG	$I(BBP)^4BB$	750	1.53	High
Simpsons	MPEG	$I(BBP)^2BB$	720	0.8	High
Animation	MPEG	$I(B^9P)^3B^9$	1200	0.7	Moderate
Space	MPEG	$IBBPBB$	550	0.61	Low

6.4 Experimental Evaluation

6.4.1 LRJ/LRM algorithms and the IRAD Architecture

To evaluate the efficacy of our loss-resilient algorithms, as well as the IRAD architecture, we have developed prototype codecs for LRJ and LRM. We carried out several experiments using these prototype codecs. In the LRJ algorithm, when the information contained in a sub-image is not available, the quality of the reconstructed image is directly dependent on the amount of original image data available for reconstruction. Hence, increasing the value of the degree of image partitioning, N, improves the quality of the reconstructed images. However, with increase in N, the efficiency of the compression algorithm deteriorates. Figure 6.11 illustrates the visual quality of the reconstructed image for various values of N.

To quantitatively capture the improvement in the quality of the reconstructed images with increase in N, we have also computed the *Peak Signal to Noise Ratio (PSNR)* for all the images. For an $M \times N$ image of resolution r bits/pixel, if $p(x, y)$ and $p'(x, y)$ denote the pixel values at location (x, y) in the original and the reconstructed images, respectively, then the PSNR value can be defined as:

$$PSNR = 10 * \log \left(\frac{(2^r - 1)^2}{\sigma^2} \right) \text{ dB}$$

where

$$\sigma = \sum_{x=1}^{M} \sum_{y=1}^{N} (p(x, y) - p'(x, y))^2$$

Figure 6.12(a) depict the variation in the PSNR value of the recovered image with increase in N for the LRJ algorithm. Figure 6.12(b), on the other hand, illustrates the degradation in compression efficiency (measured in terms of percentage increase in compressed image size) with increase in N. In practice, a server can choose an appropriate value of N depending upon the desired quality of the reconstructed image and the maximum tolerable degradation in compression efficiency. Our experiments indicate that $N = 8$ yields acceptable image quality, and results in an increase in compressed image size by about 6%.

Next, we conducted experiments to determine the efficacy of the LRM algorithm. The characteristics of the MPEG streams used in our experiments are shown in Table 6.3. Figure 6.13(a) depicts the picture quality (i.e., PSNR) for LRM streams obtained by varying the degree of image partitioning N, while Figure 6.13(b) shows the

Figure 6.11: Original and reconstructed image for $N = 4, 8, 12, 16$ with a single disk failure

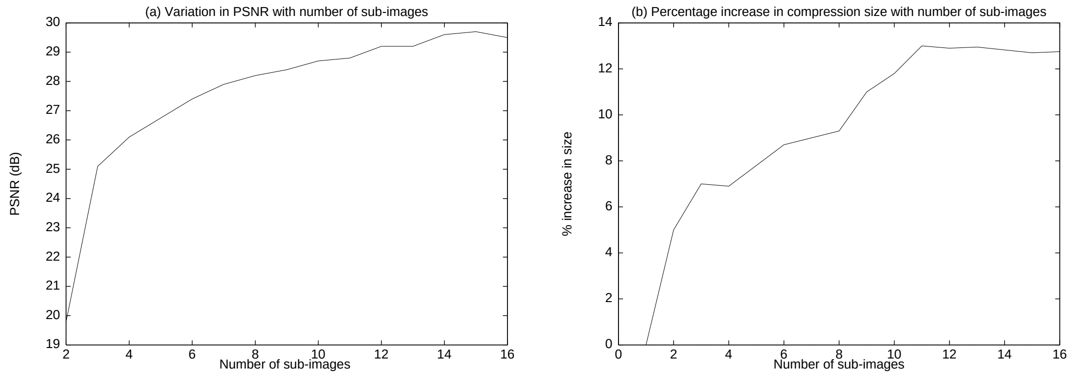


Figure 6.12: Variation of PSNR and compression efficiency with number of sub-images in LRJ

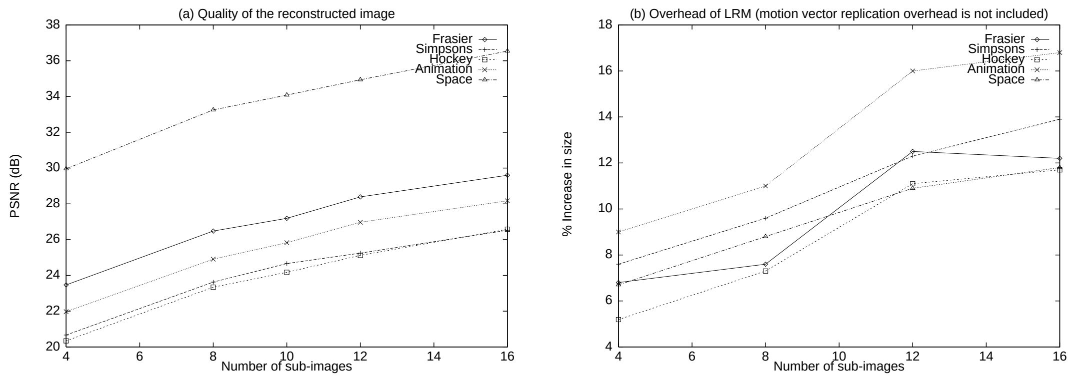


Figure 6.13: Variation of PSNR and compression efficiency with number of sub-images for LRM

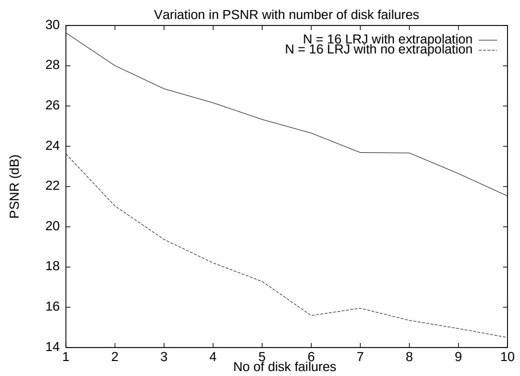


Figure 6.14: Variation of quality of reconstructed image for multiple disk failures

Frasier	5.3%
Simpsons	3.35 %
Hockey	4.8%
Animation	8.08%
Space	2.39%

Table 6.4: Overhead of Motion Vector Replication. The table shows the percentage increase in size of the original MPEG streams due to motion vector replication.

loss in compression efficiency due to image partitioning for these streams. Table 6.4 tabulates the overhead of maintaining an additional copy of the motion vectors. The overhead of replicating motion vector varied from 2% to 8%. The overhead of 8% was obtained for the animation sequence which had an abnormally large number of B frames as compared to I and P frames (see Table 6.3), and hence a larger number of motion vectors. However, for all other streams, the overhead was much lower with an average overhead of 4%. We observed reasonable recovery for $8 \leq N \leq 10$, with an 8% loss in compression efficiency. Thus, the total storage space overhead was around 12%.

Finally, to demonstrate that the IRAD architecture can tolerate multiple disk failures, we carried out several experiments. Figure 6.14 illustrates that the quality of the reconstructed image gradually deteriorates with increase in number of failed disks for the LRJ algorithm. It also demonstrates that the simple methods employed by the LRJ algorithm to extrapolate DC and AC coefficient values significantly improve the quality of the reconstructed image.

6.4.2 Parity-Based Failure Recovery

To evaluate the effectiveness of the lazy and the eager recovery schemes, we augmented our event-driven disk array simulator *diskSim*. We carried out extensive trace-driven simulations to evaluate these failure recovery schemes. The simulation environment consisted of a disk array with 32 disks. The characteristics of each disk is shown in Table 6.5. The SCAN disk scheduling algorithm was employed for retrieving data blocks from a disk during each round. Each VBR video file stored on the array is assumed to be encoded using the MPEG compression algorithm. We used the MPEG streams shown in Table 6.3 for our experiments and simulations. Data blocks of a file were assumed to be 64KB in size and striped across the disks in array. The placement strategy ensured that all data blocks in a parity group belong to the same video file. The playback rate of each stream was assumed to be 30 frames/sec.

We compared the lazy and eager recovery schemes to the standard recovery scheme in a RAID level 5 array. Figure 6.15(a) depicts the total number of blocks retrieved by the entire array during each round normalized by the number of disks in the array. Recall from Section 6.2, that the lazy and the eager schemes impose a recovery overhead of $1/(G-1)$ on a RAID level 5 array. As illustrated by the figure, for $G = 2$ or mirroring, all schemes show a 100% increase in load, which is consistent with the analytical result. For larger parity group sizes, the recovery overhead decreases with increase in G for the lazy and eager recovery schemes. On the other hand, for standard RAID level 5, the increase in load is smaller than 100% for small values of G, ($G > 2$). This is because the number of blocks accessed by each client in a round approximately equals the parity group size. Since data blocks of the parity group requested for playback need not be accessed again for failure recovery, the number of additional

Disk capacity	2 GBytes
Number of disks in the array	32
Bytes per sector	512 KB
Sector per track	99
Tracks per cylinder	21
Cylinders per disk	2627
Minimum seek time	1.7 ms
Maximum seek time	22.5 ms
Maximum rotational latency	11.1 ms

Table 6.5: Disk Parameters of Seagate-Elite3 disk

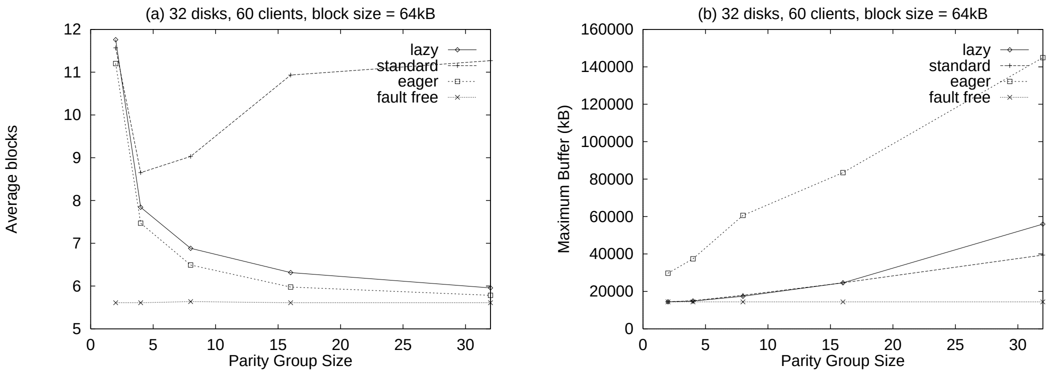


Figure 6.15: Disk recovery and buffer space overhead for lazy, eager and standard RAID-5

blocks that must be accessed to reconstruct the lost block is smaller than the worst case value of $G - 1$. However, as the parity group size increases, the number of additional blocks that must be accessed to reconstruct the lost block increases and hence, the recovery overhead approaches 100%.

Figure 6.15(b) shows the total buffer requirements of different recovery schemes. The eager recovery scheme has the highest buffer requirement with the buffer increasing linearly with the parity group size. For small values of the parity group size (i.e., when $G - 2 \leq k^{\min}$), the lazy recovery approach has the same buffer requirements as the RAID level 5 array, consistent with the analytical result. However, as the parity group size increases, $(G - 2)$ becomes greater than $k^{\min}$, and hence, the buffer requirements of the lazy scheme become larger than that for RAID level 5. Thus, our simulation results validate the analytical results derived in Section 6.3. They also demonstrate that the lazy/eager recovery schemes trade buffer space for lower recovery overhead and hence, higher array utilization.

6.5 Related Work

Recently, several research groups have adapted conventional failure recovery techniques for file servers storing continuous media data. An admission control algorithm that reserves contingency disk bandwidth to accommodate load increases in the event of a disk failure in a declustered-parity-based disk arrays was presented in [65]. Similarly, in the *Streaming RAID* server [84], a RAID level 3 array is adapted to exploit the periodic nature of video accesses for efficient data retrieval. By restricting the maximum number of users simultaneously accessing the array, the server ensures that the real time requirements of video streams are not violated even during a disk failure. Techniques that distribute parity information across a random permutation of disks, and thereby distribute the additional load due to a failure uniformly across all surviving disks were proposed in [82]. The Segmented Information Dispersal approach employs placement techniques that distribute the increase in load due to a disk failure across a large number of disks, thereby reducing the recovery overhead per disk [23]. The difference between these techniques and our approach is that the former treat continuous media data as an uninterpreted sequence of bits and do not exploit any of its characteristics. In contrast, techniques proposed in this chapter exploit the semantics of the data for efficient failure recovery.

6.6 Concluding Remarks

In this chapter, we argued that conventional failure recovery techniques can impose a higher recovery overhead than is necessary for continuous media applications. We then presented two failure recovery techniques that utilize the inherent characteristics of continuous media to ensure that the user-invoked on-the-fly failure recovery process does not impose any significant load on the disk array. Whereas the first approach utilizes the inherent redundancy in video files (rather than error-correcting codes) to recover from disk failures, the second exploits the sequential nature of playback of continuous media files to reduce the overhead of failure recovery. We demonstrated the efficacy of the former technique in the context of JPEG and MPEG compression algorithms, and showed that the latter techniques reduces the recovery overhead by a factor of $G-1$ as compared to conventional parity-based techniques. The former technique decouples the tasks of online reconstruction of requested data from that of perfect rebuild of failed disks—a fundamental departure from conventional recovery techniques which use a single mechanism, such as parity, for both tasks. The IRAD architecture that we presented is an inherently distributed, scalable, end-to-end solution to failure recovery and supports supports graceful degradation in the quality of the reconstructed images with increase in the number of disk failures.

Chapter 7

Symphony Implementation and Evaluation

Science has always prided itself on being empirical and believing only what could be verified.
—Bertrand Russell, Limitations of Scientific Method

A physically integrated file system must manage heterogeneity in application requirements and data characteristics. Symphony meets this requirement by employing a two layer architecture. The lower layer of Symphony (data type independent layer) implements a set of data type independent mechanisms that provide core file system functionality. The upper layer (data type specific layer) consists of a set of modules, one per data type, that use these mechanisms to implement data type specific policies. The layer also exports a file server interface containing methods for reading, writing, and manipulating files. Figure 7.1 depicts this architecture. The preceding chapters described the mechanisms and policies that we have developed for such a two layer architecture. In this chapter, we describe the implementation of these mechanisms and policies in Symphony and present the results of our experimental evaluation.

The rest of the chapter is organized as follows. Section 7.1 describes the data type independent layer of Symphony. Section 7.2 describes the data type specific layer. Section 7.3 presents an experimental evaluation of the Symphony prototype. Section 7.4 compares Symphony with other integrated file systems that have been developed, and finally, Section 7.5 summarizes our results.

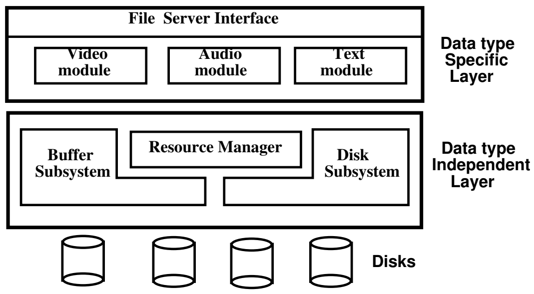


Figure 7.1: Architecture of Symphony

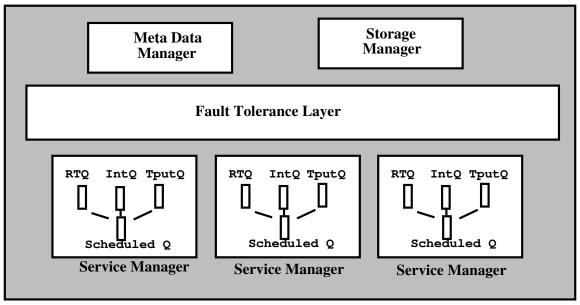


Figure 7.2: The Disk Subsystem of Symphony

7.1 The Data Type Independent Layer

The data type independent layer of Symphony employs mechanisms for disk scheduling, placement, caching, failure recovery, meta data management, and resource management. As shown in Figure 7.1, the layer consists of three components: the *disk subsystem*, the *buffer subsystem*, and the *resource manager*. In what follows, we describe each component in detail.

7.1.1 The Disk Subsystem

The disk subsystem of Symphony is responsible for efficiently multiplexing storage space and disk bandwidth among different data types. It consists of four components (see Figure 7.2): (1) a *service manager* that supports mechanisms for efficient scheduling of interactive, throughput-intensive and real-time requests; (2) a *storage manager* that supports mechanisms for allocation and deallocation of blocks of different sizes, as well as techniques for controlling their placement on a disk array; (3) a *fault tolerance layer* that enables multiple data type specific failure recovery techniques; and (4) a *meta data manager* that enables data type specific structure to be assigned to files. The key features and the algorithms used to implement these components are described below.

7.1.1.1 Service Manager

The main objective of the service manager is to support multiple service classes and meet the performance requirements of requests within each class. The service manager supports three service classes: interactive best-effort, throughput-intensive best-effort, and soft real-time; and two retrieval modes: client-pull and server-push. Interactive requests desire low average response times but do not require any performance guarantees. Throughput-intensive requests desire high aggregate throughput and do not care about the response times provided to individual requests. Real-time requests have deadlines associated with them that must be met by the service manager; real-time requests can be either periodic or aperiodic. Interactive, throughput-intensive and aperiodic real-time requests can arrive at arbitrary instants and are serviced using the client-pull mode. Periodic real-time requests, on the other hand, are serviced using the server-push mode. Such requests are issued by the data type specific layer at the beginning of each period and must be serviced by the end of the period (i.e., all requests have the end of the period as their deadline).

The service manager employs the Cello disk scheduling algorithm to meet the requirements of requests in each class. As explained in Chapter 4, Cello consists of a class-independent scheduler and a set of class-specific schedulers. The class-independent scheduler allows weights to be assigned to each class and allocates disk bandwidth to classes in proportion to their weights;[1] bandwidth unused by a class is reallocated to other classes with pending requests. The class-specific schedulers are responsible for determining a fine-grain interleaving of requests that aligns the service provided with application needs. As shown in Figure 7.2, Cello maintains four queues per disk: three pending queues, one for each service class, and a scheduled queue. Newly arriving requests are queued up in the appropriate pending queue. Pending requests are inserted into the scheduled queue by class-specific schedulers so as to: (i) reduce seek time and rotational latency overhead, and (ii) meet requirements of individual requests, such as request deadlines. Requests in the scheduled queue are then serviced by the disk in FIFO order.

7.1.1.2 The Storage Manager

The main objective of the storage manager is to enable the coexistence of multiple placement policies in the data type specific layer. To achieve this objective, the storage manager supports multiple block sizes and allows control over their placement on the disk array.

To allocate blocks of different sizes, the storage manager requires the minimum block size (also referred to as the *base* block size) and the maximum block size to be specified at file system creation time. These parameters define the smallest and the largest units of allocation supported by the storage manager. The storage manager can then allocate any block size within this range, provided that the requested block size is a multiple of the base block size. The storage manager constructs each requested block by allocating a sequence of contiguous base blocks on disk.

To allow control over the placement of blocks on the array, the storage manager allows *location hints* to be specified with each allocation request. A location hint consists of a (disk number, disk location) pair and denotes the preferred location for that block. The storage manager attempts to allocate a block conforming to the specified hint, but does not guarantee it. If it is unable to do so, the storage manager allocates a free block that is closest to the specified location. If the disk is full, or if the storage manager is unable to find a contiguous sequence of base blocks to construct the requested block, then the allocation request fails.

The ability to allocate blocks of different sizes and allow control over their placement has the following implications:

- By allowing a location hint to be specified with each allocation request, the storage manager exposes the details of the underlying storage medium (i.e., the presence of multiple disks) to the rest of the file system. This is a fundamental departure from conventional file systems which use mechanisms such as *logical volumes* to hide the presence of multiple disks from the file system. By providing an abstraction of a single large logical disk, a logical volume makes the file system oblivious of the presence of multiple disks [40]. This enables file systems built for single disks to operate without any modifications on a logical volume containing multiple disks. The disadvantage, however, is that the file system has no control over the placement of blocks on disks (since two adjacent logical blocks could be mapped by the volume manager to different locations, possibly on different disks). In contrast, by exposing the presence of multiple disks, the storage manager allows the data type specific layer precise control over the placement of blocks, albeit at the expense of having to explicitly manage multiple disks.

[1] The weights associated with a class are specified at file system startup time. In the future, we plan to extend the service manager to monitor the workload from each class and adapt these weights accordingly.

- The mechanisms provided by the storage manager enable any placement policy to be implemented in the data type specific layer. For instance, by appropriately generating location hints, a placement policy can stripe a file across all disks in the array, or only a subset of the disks. Similarly, location hints can be used to cluster blocks of a file on disks, thereby reducing seek and rotational latency overheads incurred in accessing these blocks. The placement policy can also tailor the block size on a per-file basis (depending on the characteristics of the data) and maximize file server throughput. However, allowing a large number of block sizes to coexist can lead to fragmentation. The storage manager attempts to minimize the effects of fragmentation by coalescing adjacent free blocks to construct larger free blocks [48]. However, such coalescing does not completely eliminate fragmentation effects, and hence, the flexibility provided by the storage manager must be used judiciously by placement policies in the data type specific layer. This can be achieved by restricting the block sizes used by these policies to a small set of values.

7.1.1.3 The Fault Tolerance Layer

The main objectives of the fault tolerance layer are to support data type specific reconstruction of blocks in the event of a disk failure, and to rebuild failed disks onto spare disks. To achieve these objectives, the fault-tolerance layer maintains parity information on the array. To enable data-type specific reconstruction, the fault-tolerance layer supports two mechanisms: (1) a *reliable* read, in which parity information is used to reconstruct blocks stored on the failed disk, and (2) an *unreliable* read, in which parity based reconstruction is disabled, thereby shifting the responsibility of failure recovery to the client [91]. Unlike read requests, parity computation can *not* be disabled while writing blocks, since parity information is required to rebuild failed disks onto spare disks.

The key challenge in designing the fault-tolerance layer is to reconcile the presence of parity blocks with data blocks of different sizes. The fault-tolerance layer hides the presence of parity blocks on the array by exporting a *set of logical disks*, each with a smaller capacity than the original disk. The storage manager then constructs a block by allocating a sequence of contiguous base blocks on a logical disk. To minimize seek and rotational latency overheads, we require that this sequence be stored contiguously on the physical disk as well. Since the fault-tolerance layer uniformly distributes parity blocks across disks in the array (analogous to RAID-5 [67]), the resulting interleaving of parity and data blocks can cause a sequence of contiguous blocks on a logical disk to be separated by intermediate parity blocks. To avoid this problem, the fault-tolerance layer uses a parity block size that is equal to the maximum block size that can be allocated by the storage manager (see Figure 7.3). Each data block within a parity group now contains a sequence of base blocks, all of which are contiguous on disk. By ensuring that each allocated block is contained within a data block of a parity group, the storage manager can ensure that the block is stored contiguously on disk.

7.1.1.4 The Meta Data Manager

The meta data manager is responsible for allocating and deallocating structures that store meta data information, and allows any data type specific structure to be assigned to files. Like in the UNIX file system [52], meta data structures are of a fixed size and are stored on a reserved portion of the disk. Each meta data structure contains information such as the file owner, file size, file creation time, access protection information, the block size used to store the file, the type of data stored in the file and a two level index. Level one of the index maps logical units (e.g., frames) to byte offsets, whereas level two maps byte offsets to disk block locations. This enables a file to be accessed as

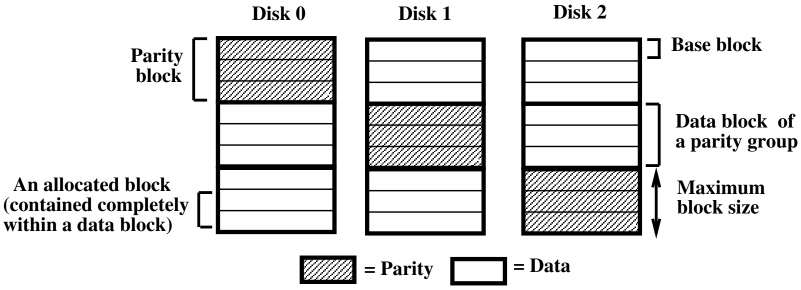


Figure 7.3: Parity placement in the fault tolerance layer

a sequence of logical units. Moreover, by using only the second level of the index, byte level access can also be provided to clients. Thus, by appropriately defining the logical unit of access, any data type specific structure can be assigned to a file.

Note that, the information contained in meta data structures is data type specific in nature. Hence, the meta data manager merely allocates and deallocates meta data structures; the actual meta data itself is created, interpreted, and maintained by the data type specific layer.

7.1.2 The Buffer Subsystem

The main objective of the buffer subsystem is to enable multiple data type specific caching policies to coexist. To achieve this objective, the buffer subsystem partitions the cache among various data types and allows each caching policy to independently manage its partition. To enhance utilization, the buffer subsystem allows the buffer space allocated to cache partitions to vary dynamically depending on the workload.

To implement such a policy, the buffer subsystem maintains two buffer pools: a pool of deallocated buffers, and a pool of cached buffers. The cache pool is further partitioned among various caching policies. Buffers within a cache partition are maintained by the corresponding caching policy in a list ordered by increasing access probabilities; the buffer that is least likely to be accessed is stored at the head of the list. To illustrate, the LRU policy maintains the least recently accessed buffer at the head of the list, while the MRU policy maintains the most recently accessed buffer at the head. For each buffer, the caching policy also computes a *time to reaccess (TTR)* metric, which is an estimate of the next time at which the buffer is likely to be accessed [83]. Since the TTR value is inversely proportional to the access probability, the buffer at the head of each list has the maximum TTR value, and TTR values decrease monotonically within a list.

On receiving a buffer allocation request, the buffer subsystem first checks if the requested block is cached, and if so, returns the requested block. In case of a cache miss, the buffer subsystem allocates a buffer from the pool of deallocated buffers and inserts this buffer into the appropriate cache partition. The caching policy that manages the partition determines the position at which the buffer must be inserted in the ordered list.

Whenever the pool of deallocated buffers falls below a low watermark, buffers are evicted from the cache and returned to the deallocated pool.[2] The buffer subsystem uses TTR values to determine which buffers are to be evicted

[2] An alternative is to evict cached buffers only on demand, thereby eliminating the need for the deallocated pool. However, this can slow down the buffer allocation routine, since the buffer to be evicted may be dirty and would require a disk write before the eviction. Maintaining a small pool of deallocated buffers enables fast buffer allocation without any significant reduction in the cache hit ratio.

from the cache. At each step, the buffer subsystem compares the TTR values of buffers at the head of each list and evicts the buffer with the largest TTR value. If the buffer is dirty, then the data is written out to disk before eviction. The process is repeated until the number of buffers in the deallocated pool exceeds a high watermark.

7.1.3 The Resource Manager

The key objective of the resource manager is to reserve system resources (i.e., disk bandwidth and buffer space) to provide performance guarantees to real-time applications. To achieve this objective, the resource manager uses: (1) a QoS negotiation protocol which allows clients to specify their resource requirements, and (2) admission control algorithms that determines if sufficient resources are available to meet the QoS requirement specified by the client.

The QoS negotiation process between the client and the resource manager uses a two phase protocol. In the first phase, the client specifies the desired QoS parameters to the resource manager. Typical QoS parameters specified are the amount of data accessed by a request, the deadline of a request and duration between successive requests. The resource manager then invokes the appropriate admission control algorithm to determine if it has sufficient disk bandwidth and buffer space to service the client. The admission control algorithm returns a set of QoS parameters, which indicate the resources that are available to service the client at the current time. Depending on the available resources, the QoS parameters returned by the admission control algorithm can be less than or equal to the requested QoS. The resource manager tentatively reserves these resources for the client and returns the results of the admission control algorithm to the client. In the second phase, depending on whether the QoS parameters are acceptable to the client, it either confirms or rejects these parameters. In the former case, the tentatively reserved resources are committed for the client. In the latter case, resources are freed and the negotiation process must be restarted, either with a reduced QoS requirement, or at a later time. If the resource manager does not receive either a confirmation or rejection from the client within a specified time interval, it releases the resources that were reserved for the client. This prevents malicious or crashed clients from holding up unused resources. Once QoS negotiation is complete, the client can begin reading or writing the file. The reserved resources are freed when the client closes the file.

Depending upon the nature of the guarantees provided, admission control algorithms can be classified as either deterministic or statistical. Deterministic admission control algorithms make worst case assumptions about resources required to service a client and provide deterministic (i.e., hard) guarantees to clients. In contrast, statistical admission control algorithms use probabilistic estimates about resource requirements and provide only probabilistic guarantees. The key tradeoff between deterministic and statistical admission control algorithms is that the latter leads to better utilization of resources than the former at the expense of weaker guarantees. Several admission control algorithms have been proposed for providing deterministic as well as statistical guarantees [71, 90, 89]; the resource manager employs variants of these algorithms for reserving resources.

7.2 The Data Type Specific Layer

The data type specific layer consists of a set of modules that use the mechanisms provided by the data type independent layer to implement policies optimized for a particular data type. The layer also exports a file server interface containing methods for reading, writing, and manipulating files (see Figure 7.1). Each module implements a data type specific version of these methods, thereby enabling applications to create and manipulate files of that data type. In what follows, we first describe data type specific modules for two data types, namely video and text and then describe the file server interface.

7.2.1 The Video Module

The video module implements policies for placement, retrieval, meta data management, and caching of video data. Before describing these policies, let us first examine the structure of a video file. Digitization of video yields a sequence of frames. Since the size of a frame is quite large (a typical frame is 300KB in size), digitized video data is usually compressed prior to storage. Compressed video data can be multi-resolution in nature, and hence, each video file can contain one or more sub-streams. For instance, an MPEG-1 encoded video file always contains a single sub-stream, while MPEG-2 encoded files can contain multiple sub-streams [41]. Depending on the desired resolution, only a subset of of these sub-streams need to be retrieved during video playback; all sub-streams must be retrieved for playback at the highest resolution.

The video module supports video files compressed using a variety of compression algorithms. This is possible since the video module does not make any compression-specific assumptions about the structure of video files. Each file is allowed to contain any number of sub-streams, and each frame is allowed be arbitrarily partitioned among these sub-streams. Hence, a sub-stream can contain all the data from a particular frame, a fraction of the data, or no data from that frame. Such a file structure is general and encompasses files produced by most commonly used compression algorithms. Assuming this structure for a video file, we now describe placement, retrieval, meta data management and caching policies for video data.

7.2.1.1 Placement Policy

Placement of video files on disk arrays is determined by the block size and the striping policy. The video module supports both fixed-size blocks (each of which contains a fixed number of bytes) and variable-size blocks (each of which contains a fixed number of frames). Fixed-size blocks are more suitable for environments with frequent writes and deletes, whereas variable-size blocks incur lower seek and rotational latency overheads and enable a file server to employ load balancing policies [92]. In either case, the specific block size to be used can be specified by the client at file creation time (a default value is used if the block size is unspecified). The video module then uses the interfaces exported by the storage manager to allocate blocks of the specified size. While the block size is known *a priori* for fixed-size blocks, it can change from one block to another for variable-size blocks. In the latter case, the video module determines the total size of the next f frames within a sub-stream (assuming that each variable-size block contains f frames), rounds it upwards to a multiple of the base block size, and requests a block of that size from the storage manager. Since the size of f frames may not be an exact multiple of the base block size, to prevent internal fragmentation, the video module stores some data from the next f frames in the unused space in the current variable-size block. Hence, accessing a variable-size block causes this extra data to be retrieved, which is then cached by the video module to service future read requests.

To effectively utilize the array bandwidth, the video module stripes each sub-stream across disks in the array. If sub-streams are stored on the array in terms of variable-size blocks, then the module stripes each sub-stream across all the disks in the array. On the other hand, when sub-streams are stored on the array in terms of fixed-size blocks, the striping policy depends on the array size. For small disk arrays, each sub-stream is striped across all disks in the array. However, such a policy degrades performance for large disk arrays. Hence, large arrays (consisting of few tens of disks or more) are partitioned into sub-arrays and each file is striped across a single sub-array [77].

Since the storage manager allows the disk number to be specified with the hint, such a striping policy can be easily implemented by generating appropriate location hints. The striping policy also optimizes the placement of multi-resolution video files on the array. This is achieved by storing blocks of different sub-streams that are likely to be accessed together adjacent to each other on disk [78]. Such contiguous placement significantly reduces seek and rotational latency overheads incurred during video playback. These multi-resolution optimizations can be easily implemented by generating appropriate location hints.

7.2.1.2 Retrieval Policy

The video module uses the interface provided by the service manager to support both periodic real-time and aperiodic real-time requests. Periodic real-time requests are supported in the server-push mode, while aperiodic real-time requests are serviced in the client-pull mode. Since periodic real-time requests are serviced by the disk scheduler in terms of periodic rounds, such requests are issued by the video module at the beginning of each round. For each periodic real-time client, the video module generates a list of blocks to be read or written and inserts them in the periodic real-time queue of the service manager at the beginning of a round. In contrast, for aperiodic real-time clients, the video module waits for an explicit request from the client before retrieving data. Such requests arrive at arbitrary instants and are inserted on arrival into the aperiodic real-time queue of the service manager.

7.2.1.3 Meta data Management

Symphony allows any data type specific structure to be assigned to files. Since the logical unit of access for video is a frame, the video module allows each file to be accessed as a sequence of frames. Each file can also be accessed as a sequence of bytes to support applications that require a byte stream interface. To allow efficient random access at the byte level and the frame level, the module maintains a two level index structure. The first level of the index, referred to as the frame map, maps frame offsets to byte offsets, while the second level, referred to as the byte map, maps byte offsets to disk block locations. Figure 7.4 illustrates the index structure. Whereas both levels of the index are used during frame-level access, only the byte map is used during byte-level access.

The two level structure permits efficient random access for fixed-size blocks. However, supporting random access for variable-size blocks is not straightforward, since variable block sizes complicate the mapping from byte offsets to block locations in the byte map. Recall that, each variable-size block consists of a sequence of base blocks which are of fixed size. Hence, by maintaining a mapping from byte offsets to block locations for *base blocks* (instead of variable-size blocks), it is possible to support efficient random access for variable-size blocks as well. The tradeoff though is the increased storage space requirement for the byte map.

7.2.1.4 Caching Policy

Since video accesses are sequential, caching policies such as LRU are ineffective for video files [13]. Recently, the Interval Caching policy was proposed for caching video blocks. The policy caches the interval between two clients accessing the same file, thereby serving requests of the trailing client from the cache. Given a fixed amount of buffer space, the policy maximizes the number of cached intervals (and hence, utilization) by caching intervals in increasing order of sizes [26].

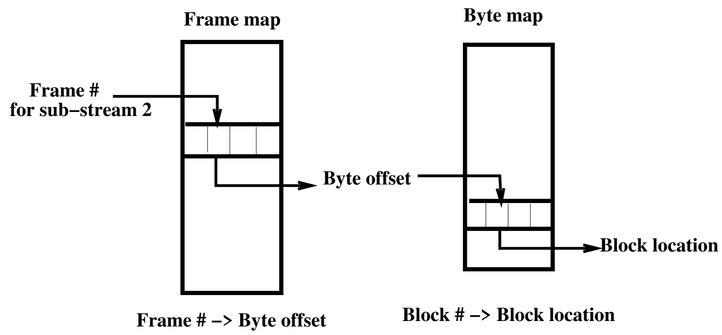


Figure 7.4: The index structure for the video inode. Assuming that a video file contains n sub-streams, both the frame map and the byte map contain a sequence of n-tuples. Each tuple in the frame map represents a frame, and the i^{th} field of a tuple denotes the location (i.e., byte offset) of that frame in sub-stream i. Each tuple in the byte map represents a block, and the i^{th} field of a tuple denotes the location of the block in sub-stream i.

The video module uses the Interval Caching policy to cache video blocks. Blocks of a file that are accessed by a single client are never cached (i.e., they are returned to the deallocated pool after use). When a second client starts accessing a file, then the video module begins caching blocks being accessed by the first client in its cache partition. The trailing client must access the initial portion of the file from disk (since those blocks weren't cached). All subsequent accesses, however, are serviced from the cache.

7.2.2 The Text Module

The policies implemented by the text module are very similar to those employed by conventional UNIX file systems [5, 52]. For instance, the text module supports only best-effort requests, all of which are serviced in the client-pull mode. The placement policy employed by the text module supports only fixed-size blocks, and stripes successive blocks of a file onto consecutive disks in a round-robin manner. The text module supports only a byte-level access to each file. To do so, it maintains a byte map for each text file, which is similar to the UNIX inode. Finally, as in UNIX, the text module uses an LRU policy to cache text blocks.

7.2.3 The File Server Interface

The data type specific layer also exports a set of methods that constitute the file server interface of Symphony. These methods are used by applications to create and manipulate files. Each module in the data type specific layer implements a data type specific version of these methods. The module may exclude certain methods if they are not relevant to the data type. For instance, since all text files are accessed using the client-pull mode, the text module does not support methods for the server-push retrieval mode. Table 7.1 lists the file server interface. The interface consists of four types of methods: methods for creating and deleting files, methods for client-pull-based reads and writes, methods for server-push-based accesses, and methods that reserve resources for real-time applications.

In addition to the above RPC interface, Symphony also implements the vnode interface of Solaris (the vnode interface provides a framework for supporting multiple file systems within Solaris [87]). Doing so enables Symphony to support all existing applications without any modifications. The interface also enables the Symphony server to be

Table 7.1: Symphony File Server Interface

fileHandle	=	ifsCreate(file name, data type, options)
fileHandle	=	ifsOpen(file name, options)
result	=	ifsDelete(file name)
result	=	ifsClose(fileHandle)
result	=	ifsFlush(fileHandle)
result	=	ifsRead(fileHandle, size, buffer, optional deadline)
result	=	ifsWrite(fileHandle, size, buffer, optional deadline)
result	=	ifsSeek(fileHandle, offset)
result	=	ifsPeriodicRead(fileHandle, recvPort[numSubStreams])
result	=	ifsPeriodicWrite(fileHandle, sendPort[numSubStreams])
result	=	ifsStop(fileHandle)
result	=	ifsQosNegotiate(fileHandle,qosIn,qosOut)
result	=	ifsQosConfirm(fileHandle,qos)
metaData	=	ifsgetMetaData(fileHandle,options)

mounted on client machines using NFS, thereby providing remote access to files. We have implemented the vnode interface as a loadable module in the Solaris kernel; the module communicates with the Symphony server (which resides in user space) using a pseudo-upcall mechanism[3] [6].

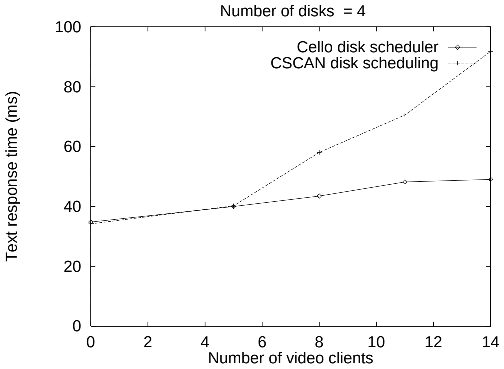
7.3 Experimental Evaluation of the Symphony Prototype

The prototype implementation of Symphony runs as a single multi-threaded process in user space and accesses disks as raw devices. We have used the prototype to evaluate the efficacy of policies and mechanisms implemented in Symphony. The testbed for our experiments consists of a cluster of Sun workstations connected by an ATM network. The Symphony prototype runs on a dual-processor Ultra Sparc (Model 2700) that has 128 MB of RAM and runs Solaris 2.5.1. The storage medium used for the server consists of four 2.1 GB Seagate Barracuda disks (Model ST12450W) connected to the Ultra Sparc via a fast wide SCSI interface. Symphony application programs run on a cluster of four Sparc-20 and Sparc-5 workstations, all of which run Solaris 2.5. All machines are connected to a 155 Mb/s Fore ATM switch (Model ASX-200) using ATM adapter cards. In what follows, we describe our experiments and analyze our results.

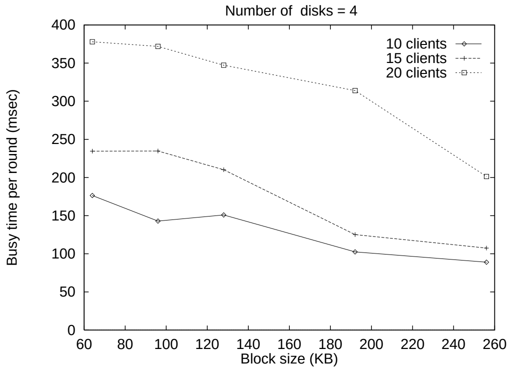
7.3.1 Performance of Text and Video Clients

Recall from Chapter 4 that, the Cello disk scheduler delays the servicing of real-time requests until their deadlines and uses the available slack to service interactive requests. By giving priority to interactive text requests whenever sufficient slack is available, Cello provides low response times to these requests. To demonstrate this behavior, we compiled two versions of the prototype, one which used Cello and the other which used CSCAN. In both cases, we populated the server with a large number of text and video files. Each text file was 64KB in size and was striped using

[3]Since Solaris does not support upcalls, we have devised a pseudo-upcall mechanism that enables the kernel to communicate with a user process. Our mechanism creates a number of threads in the user process, each of which issues a system call to enter kernel address space and then blocks. A newly arriving request in the kernel is handed to one these threads; the system call then returns with this request. After servicing the request, the results are transmitted back to the kernel using another system call.



(a)



(b)

Figure 7.5: Performance of text and video clients

a block size of 4KB. Each video file was 6.5MB in size and contained 1000 MPEG-1 compressed frames, striped using a block size of 64KB.[4] The playback rate for the each video file was 30 frames/s and the average bit rate was 1.5 Mb/s. We assigned weights of $w_1 = 0.6, w_2 = 0.05$, and $w_3 = 0.35$ to the real-time, throughput-intensive, and interactive classes, respectively. The duration of a round was set to 1 second. For both versions of the prototype, we varied the number of video clients and measured the response time seen by text requests. Each video client in our experiments was a modified version of `mpeg_play` and retrieved a randomly selected file in the periodic real-time mode. Each text client read a randomly selected text file in sequential order using 8KB requests. Figure 7.5(a) plots the average response time for a 8KB request observed in the two cases. The figure shows that the Cello provides better response times to text requests than CSCAN. This is because, CSCAN services requests in the order of their disk cylinder numbers. Since it interleaves text and video requests based on this ordering, the response times seen by text requests increases with increase in number of video requests. In contrast, Cello always gives priority to text requests regardless of the number of video requests (provided sufficient slack is available). Moreover, unused allocation of the real-time class is reassigned to the interactive class. This results in better response times to text requests, and the response time degrades only slightly with increase in number of video clients.

To demonstrate that the improvement in the response time for text requests did not come at the expense of deadline violations for video requests, we repeated the above experiment by varying the block size used for each video file, and measured the service time of disks (i.e., the duration for which a disk was busy in each round). Figure 7.5(b) depicts the service time of a disk for different number of video clients and different block sizes. It shows that the service time of the disk is within 600 ms, which is the duration of each round allocated to real-time requests (since $w_1 = 0.6$ and the round duration is 1s). Hence, the disk scheduler is able to meet the real-time requirement of video clients. Together, Figures 7.5(a) and (b) show that, even at a moderate disk utilization level of 25%, Symphony yields a factor of 1.9 improvement in response time of text requests over conventional disk scheduling algorithms, while meeting the deadlines of all real-time video requests,

Recall that, Symphony allows a client to specify the block size at file creation time. The block size used to stripe a file can have a significant impact on the server performance. Choosing a large block size reduces seek and rotational latency overheads and increases disk throughput (see Figure 7.6). However, as demonstrated in Chapter 5, choosing a large block size reduces the total number of blocks accessed from the array and results in a sparsely loaded array.

[4]The data for the video files was obtained by digitizing several television sitcoms, newscasts and sports events.

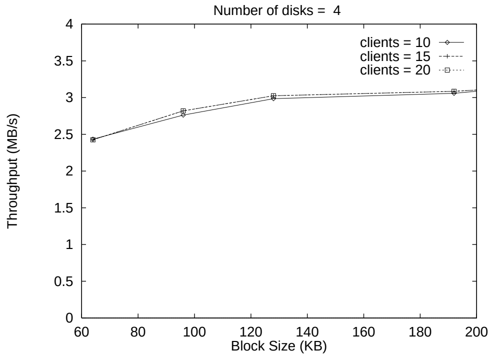


Figure 7.6: Impact of block size on server throughput

This can cause load imbalances on the array and reduce the number of clients supported by the server. A block size that balances these tradeoffs yields optimal performance. Analytical models presented in Chapter 5 can be used to determine such a block size.

7.3.2 Performance in the presence of disk failure

The fault-tolerance layer allows multiple fault tolerance policies to coexist within the file system. Specifically, it allows clients to enable or disable parity-based reconstruction (using reliable or unreliable reads) for recovering data. Since entire parity group must be retrieved to reconstruct a block requested from the failed disk, in the worst case, parity based reconstruction imposes a large (100%) overhead on the server [19]; no such overhead is imposed when parity-based reconstruction is disabled. To demonstrate this fact, we configured the prototype to assume that one of the disks in the array had failed. We varied the number of video clients and measured the load on the server (in terms of the service time of a disk) with parity-based reconstruction enabled. Next, we repeated the experiment with parity-based reconstruction disabled. Figure 7.7 plots the load on the server for the two scenarios. With parity-based reconstruction disabled, the server does not retrieve any additional data as compared to the fault-free state; hence, the load on the server remained unchanged. With parity-based reconstruction enabled, the service time of a disk was approximately twice of that in the fault-free state. Thus, disabling parity-based reconstruction reduces the failure recovery overhead for video applications from a factor of two to zero. The tradeoff though is that this option requires sophisticated clients that can exploit redundancies in video data to approximately reconstruct lost data. Although approximate reconstruction causes a degradation in image quality, as shown in Chapter 6, the degraded quality is within human perceptual tolerances.

7.4 Related Work

Several recent and ongoing research efforts have focussed on designing integrated file systems. The Fellini storage manager [55] and CMFS [2] are file systems that can handle both real-time continuous media data and best-effort textual data. Both are single disk file systems and do not employ multi-disk optimizations such as striping. Similarly, MMFS is a single disk file system that adds continuous media support to a FreeBSD-based file system [63]. The Tiger Shark file system from IBM and XFS from SGI are results of commercial efforts to build integrated file

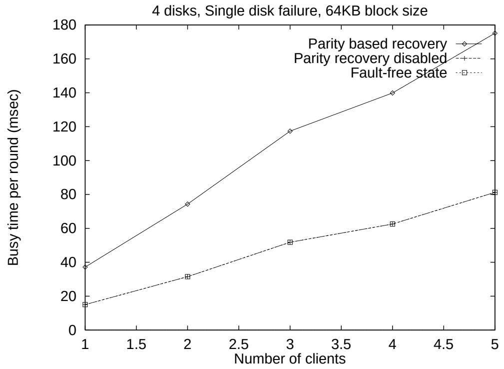


Figure 7.7: Reliable reads versus unreliable reads. The figure shows that disabling parity-based reconstruction significantly lowers the load on the server.

systems [36, 40]. These file systems come closest to Symphony in terms of the features offered. For instance, these file systems support variable-size blocks (referred to as *extents*), guaranteed rate I/O, etc. However, they do not employ features such as a disk scheduler that supports diverse applications, data type specific placement, failure recovery, and caching policies. Moreover, they statically partition the storage space available on the disk array using logical volumes. Logical volumes can either span a mutually exclusive set of disks, or share the same set of disks by statically partitioning the space on each disk. In the former case, both storage space and disk bandwidth get statically partitioned among logical volumes, while in the latter case only the storage space is statically partitioned and the disk bandwidth is dynamically shared by logical volumes. In contrast, Symphony is a physically integrated file system, in which all resources are dynamically shared among applications.

The logical disk abstraction [46] provides an interface that allows multiple *file systems* to coexist on a single storage device. Logical disks provide functionalities similar to those provided by the data type independent layer of Symphony, such as multiple block sizes, location hints, etc. However, a key difference between logical disks and Symphony is that the former does not differentiate between request types, and consequently provides only a best-effort service model. In contrast, Symphony employs multiple service classes that enable it to efficiently support real-time requests as well as best-effort requests.

Several efforts have focussed on designing extensible and adaptive systems. Exokernel [30] and SPIN [8] are two efforts towards building extensible operating systems. These systems focus on securely multiplexing resources among untrusted best-effort applications, whereas Symphony focuses on multiplexing resources so as to provide different qualities of service to applications. Stackable file systems enable file system extensions by layering one file system abstraction on top on another [47]. Finally, the Odyssey system employs application-aware adaptation (i.e., a collaborative partnership between the operating system and applications) to efficiently support diverse applications in a mobile computing environment [64].

7.5 Concluding Remarks

In this chapter, we discussed the implementation and evaluation of the Symphony prototype. The architecture of Symphony consists of two layers. The lower layer of Symphony implements a set of data type independent mechanisms that implement core file system functionality. Some of the novel features of this layer include: (1)

the Cello disk scheduler for supporting multiple scheduling policies, (2) a storage manager for supporting multiple placement policies, (3) a fault-tolerance layer that enables data type specific failure recovery, (4) a meta data manager that enables data type specific structure to be assigned to files while continuing to support the traditional byte stream interface, (5) a buffer manager that supports multiple caching polices, and (6) a resource manager that reserves resources to provide performance guarantees to real-time applications. The upper layer contains a set of modules that use these mechanisms to implement data type specific policies for placement, retrieval, failure recovery, caching and meta data management.

Our experiments with the Symphony prototype demonstrated the efficacy of dynamic allocation of resources and supporting multiple data type specific policies. Our results showed that Symphony yields a factor of 1.9 improvement in text response time over conventional disk scheduling algorithms even in the presence of moderate video loads, while continuing to meet the real-time requirements of video clients. We also showed that (i) tailoring the block size and placement policy to application requirements improves server throughput, and (ii) supporting data type specific failure recovery policies enables Symphony to reduce the recovery overhead for continuous media applications from a factor of two to zero.

Chapter 8

Conclusions

So Long, and Thanks for All the Fish.

—Douglas Adams, The Ultimate Hitchhiker's Guide

The emergence of applications with diverse performance requirements and data with heterogeneous characteristics have made existing file systems designed for a single application class inadequate. In this dissertation, we developed Symphony—an integrated multimedia file system that overcomes this limitation. In what follows, we first summarize the contributions of this dissertation and then explore avenues for future work.

8.1 Summary of Contributions

In this dissertation, we made five primary contributions. First, we proposed two different methodologies for designing integrated file systems and evaluated their tradeoffs. Second, we designed mechanisms that enable the coexistence of diverse data type specific policies in the file system. Third, we designed data type specific policies that exploit the characteristics of the data to optimize file system performance. Fourth, we developed a novel two layer architecture for Symphony that separates data type independent mechanisms from data type specific policies, and thereby facilitates easy extensions to the file system. Fifth, we instantiated our policies and mechanisms in a prototype implementation of Symphony and experimentally demonstrated the efficacy of our techniques for managing diverse applications and heterogeneous data. In what follows, we describe our contributions in detail.

We first proposed and evaluated two different methodologies for designing integrated file systems, namely logically integrated file systems and physically integrated file systems. We argued that use of a single physically integrated server for all applications is desirable in many environments over logically integrated file systems that employ separate servers for each application class. For such environments, we demonstrated that dynamic sharing of resources inherent in the former approach yields a manifold performance improvement over employing separate servers, at the possible expense of increased file system complexity. Based on these results, we chose the physically integrated file system architecture for designing Symphony.

We then examined the requirements imposed on a physically integrated file system and argued that managing heterogeneity in application requirements and data characteristics is key to such file systems. To do so, unlike existing file systems that employ a single technique for all applications, an integrated file system must enable the coexistence of multiple application-specific and data-type-specific techniques. Specifically, an integrated file system must: (i) export multiple classes of service to applications, (ii) support multiple data type specific policies for placement, failure recovery, caching, etc., and employ mechanisms that enable their coexistence, and (iii) employ an extensible architecture.

To meet these requirements, we first developed mechanisms for dynamically allocating file systems resources. The design of such mechanisms poses several challenges: (i) the mechanisms must be powerful enough to enable the coexistence of diverse policies, and (ii) the mechanisms must prevent interference between policies with conflicting requirements, while providing all the benefits of dynamic resource allocation. We developed mechanisms for disk scheduling, placement, caching, failure recovery and meta data management that achieved these objectives. The Cello disk scheduling framework that we developed for managing disk requests with different requirements achieved these objectives by: (i) supporting multiple application classes and aligning the service provided with application needs, (ii) protecting application classes from each other, (iii) being work-conserving and adapting to changes in work-load, (iv) minimizing seek and rotational latency overhead, and (v) being computationally efficient. Our experimental evaluation of Cello showed that, at a disk utilization of 60%, Cello yields a factor of 2.5 improvement in response time over a conventional disk scheduling algorithm such as SCAN when scheduling a mixture of text and video clients.

We then developed several data type specific policies for placement, failure recovery and meta data management. Since policies for managing textual data are well known, we focused on designing policies for continuous media. For placement, we developed analytical models that predict the optimal stripe unit size and degree of striping for disk arrays storing continuous media. Our models are the first in the literature to accurately characterize the performance of the disk arrays storing variable bit rate continuous media. For disk failure recovery, we proposed two novel techniques that utilize the characteristics of continuous media for efficient recovery. Whereas the first technique exploits the sequentiality of continuous media playback to reduce the recovery overhead in conventional disk arrays, the second technique exploits the inherent redundancy in video files (rather than error correcting codes) to *approximately* reconstruct data stored on failed disks. We showed that the former technique reduces the failure recovery overhead by a factor of $G - 1$ as compared to conventional techniques, where G is the parity group size. The latter technique decouples the task of *online reconstruction* of requested data from that of *rebuild* of failed disks—a fundamental departure from conventional recovery techniques, which employ a single mechanism, such as parity, for both tasks. Furthermore, the technique enhances the scalability of integrated file systems by: (1) integrating online reconstruction with the decompression of video files at client sites, and thereby reducing the recovery overhead at the server to zero; and (2) supporting graceful degradation in the quality of recovered images with increase in the number of disk failures.

We proposed a novel two layer architecture for Symphony, consisting of a data type independent layer and data type specific layer. Whereas the data type independent layer consists of a set of mechanisms that implement core file system functionality, the data type specific layer consists of a set of modules that use these mechanisms to implement data type specific policies. The two layers of Symphony cleanly separate data type independent mechanisms from data type specific policies, and thereby facilitate easy extensions to the file systems. We implemented the mechanisms and policies that we developed in the data type independent layer and the data type specific layers of Symphony, respectively. The data type independent layer of our prototype consists of a number of novel features including: (i) the Cello disk scheduling framework for supporting multiple service classes, (ii) a storage manager for supporting multiple placement policies, (iii) a fault-tolerance layer that enables data type specific failure recovery, (iv) a meta data manager that enables data type specific structure to be assigned to files while supporting the traditional byte stream interface, (v) a buffer manager that supports multiple caching policies, and (vi) a resource manager that reserves resources to provide performance guarantees to real-time applications. The data type specific layer contains modules for text, video and audio that use these mechanisms to implement data type specific policies. The video module, for instance, implements policies for placement, retrieval, failure recovery, meta data management, and caching that are tailored for multi-resolution video files and supports both server-push and client-pull modes for accessing files.

Our experimental evaluation of the Symphony prototype demonstrated its effectiveness in supporting diverse application classes and managing heterogeneous data. Our results showed that: (i) even at moderate utilization levels, a four disk Symphony prototype yields a factor of 1.9 improvement in response time of text requests over conventional disk scheduling techniques, while meeting the deadlines of all real-time video requests, (ii) tailoring the placement policy to application needs enables Symphony to optimize server throughput and (iii) supporting data type specific failure recovery policies enables Symphony to reduce the recovery overhead from a factor of two to zero for continuous media applications.

8.2 Directions for Future Research

In this dissertation, we addressed several issues regarding the architecture, mechanisms and policies employed by an integrated file system; however several questions remain unanswered and present avenues for future research.

- *File system interface:* An important issue not addressed by this dissertation is that of the interface exported by an integrated file system. We propose to investigate if the interface exported by an integrated file system needs to be different from that exported by existing file systems. We plan to examine if it is possible to change the underlying mechanisms and policies without changing the file system interface. If not, we plan to design a minimal set of enhancements to the existing file system interface that are both necessary and sufficient for supporting next generation applications.

- *Coordinated Resource Scheduling:* The performance guarantees provided by an integrated file system, especially to real-time applications, are contingent upon predictable allocation of other resources managed by the operating system and the network (e.g., processor bandwidth, network bandwidth). In fact, providing such guarantees requires coordinated scheduling of resources and appropriate accounting of resource usage. We plan to investigate issues in coordinated resource management so as to improve file server performance.

- *Scale:* The current design of Symphony employs a centralized file server and is targeted for LAN environments with tens or hundreds of users. However, integrated file systems will be employed in environments, such as the World Wide Web, to deliver information to thousands or millions of users over wide area networks. To scale to a large number of users, integrated file systems will be required to employ a distributed architecture consisting of multiple nodes. The design of such multi-node integrated multimedia file systems is an open problem.

Bibliography

[1] R K. Abbott and H. Garcia-Molina. Scheduling I/O Requests with Deadlines: A Performance Evaluation. In *Proceedings of RTSS*, pages 113–124, December 1990.

[2] D. Anderson, Y. Osawa, and R. Govindan. A File System for Continuous Media. *ACM Transactions on Computer Systems*, 10(4):311–337, November 1992.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[4] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of SIGMETRICS 96*, pages 126–137, May 1996.

[5] M. J. Bach. *The Design of the Unix Operating System*. Prentice Hall, 1986.

[6] Balaji Balasubramanyan. System Integration and Performance Evaluation of Symphony: A Multimedia File System. Master's thesis, Univ. of Texas at Austin, May 1998.

[7] P. Barham. A Fresh Approach to File System Quality of Service. In *Proceedings of NOSSDAV'97, St. Louis, Missouri*, pages 119–128, May 1997.

[8] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 267–284, December 1995.

[9] S Berson, L Golubchik, and R R. Muntz. Fault Tolerant Design of Multimedia Servers. In *Proceedings of SIGMOD Conference*, pages 364–375, 1995.

[10] D. Bitton and J. Gray. Disk Shadowing. In *Proceedings of the 14th Conference on Very Large Databases*, pages 331–338, 1988.

[11] M. Buddhikot, X. Chen, D. Wu, and G. Parulkar. Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS'98), Austin, TX*, pages 326–337, July 1998.

[12] B. Callaghan. WebNFS: The File System for the Internet. Technical report, Sun Microsystems Whitepaper, April 1997.

[13] P. Cao. *Application Controlled File Caching and Prefetching*. PhD thesis, Princeton University, 1996.

[14] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP Parallel RAID Architecture. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 52–63, May 1993.

[15] M J. Carey, R. Jauhari, and M. Linvy. Priority in DBMS Resource Scheduling. In *Proceedings of the 15th VLDB Conference*, 1989.

[16] M. S. Chen, H. I. Hsiao, C. S. Li, and P. S. Yu. Using Rotational Mirrored Declustering for Replica Placement in a Disk-array-based Video Server. In *Proceedings of the Third ACM Conference on Multimedia, San Francisco, California*, pages 121–130, November 1995.

[17] P. Chen and D. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of ACM SIGARCH Conference on Computer Architecture, Seattle, WA*, pages 322–331, May 1990.

[18] P. M. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.

[19] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, pages 145–185, June 1994.

[20] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *Journal of Real-Time Systems*, 3:307–336, 1991.

[21] E G. Coffman and M. Hofri. On the Expected Performance of Scanning Disks. *SIAM Journal of Computing*, 10(1):60–70, February 1982.

[22] E G. Coffman, L A. Klimko, and B. Ryan. Analysis of Scanning Policies for Reducing Disk Seek Times. *SIAM Journal of Computing*, 1(3):269–279, September 1972.

[23] A. Cohen and W. A. Burkhard. Segmented Information Dispersal. Technical Report CS95-444, Department of Computer Science, University of California, San Diego, 1995.

[24] G. Copeland and T. Keller. A Comparison of High-Availability Media Recovery Techniques. In *Proceedings of the ACM Conference on Management of Data*, pages 98–109, 1989.

[25] A. Dan and D. Sitaram. An Online Video Placement Policy based on Bandwidth to Space Ratio (BSR). In *Proceedings of ACM SIGMOD'95, San Jose, CA*, pages 376–385, May 1995.

[26] A. Dan and D. Sitaram. A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads. In *Proceedings of Multimedia Computing and Networking (MMCN) Conference*, pages 344–351, 1996.

[27] J. M. Danskin, G. M. Davies, and X. Song. Fast Lossy Internet Image Transmission. In *Proceedings of the Third ACM Conference on Multimedia, San Francisco, California*, pages 321–332, November 1995.

[28] P J. Denning. Effects of Scheduling on File Memory Operations. In *Proceedings of AFIPS SJCC*, pages 9–21, 1967.

[29] Digidesign Corporation, http://www.digidesign.com. *Pro Tools III*, 1998.

[30] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 251–266, December 1995.

[31] R. Flynn and W. Tetzlaff. Disk Striping and Block Replication Algorithms for Video File Servers. In *Proceedings of the International Conference on Multimedia Computing Systems (ICMCS)*, pages 590–597, 1996.

[32] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):47–58, April 1991.

[33] R. Geist and S. Daniel. A Continuum of Disk Scheduling Algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, February 1987.

[34] G Gibson and D Patterson. Designing Disk Arrays For High Data Reliability. *Journal of Parallel and Distributed Computing*, pages 4–27, January 1993.

[35] J. Gray, B. Horst, and M. Walker. Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 148–160, 1990.

[36] R. Haskin and F. Schmuck. The Tiger Shark File System. *Proceedings of COMPCON*, Spring 1996.

[37] M. Hofri. Disk Scheduling: FCFS vs. SSTF Revisited. *Communications of the ACM*, 23(11):645–653, November 1980.

[38] M. Holland and G. Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 23–35, October 1992.

[39] M. Holland, G. Gibson, and D. Siewiorek. Fast, On-line Recovery in Redundant Disk Arrays. In *Proceedings of the 23rd International Symposium on Fault Tolerant Computing*, pages 422–431, 1993.

[40] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit File System with Guaranteed Rate I/O. Technical report, Silicon Graphics, Inc, Available online as http://www.sgi.com/Technology/xfs-whitepaper.html.

[41] International Organisation for Standardisation. *Information Technology - Generic Coding of Moving Pictures and Associated Audio Systems: Systems, Video and Audio, International Standard (MPEG2), ISO/IEC 13818,*, November 1994.

[42] International Organization for Standardization (ISO). *Information technology – Coding of moving pictures and associated audio for digital storage media upto 1,5 Mbits/s, International Standard IS 11172 (MPEG)*, 1992.

[43] D M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical report, Hewlett Packard Labs, February 1991.

[44] P W. Jardetzky. *Network File Server Design for Continuous Media*. PhD thesis, University of Cambridge, August 1992.

[45] M.B. Jones, P. Leach, R. Draves, and J. Barrera. Support for User-Centric Modular Real-Time Resource Management in Rialto Operating System. In *Proceedings of NOSSDAV'95, Durham, New Hampshire*, April 1995.

[46] W. Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, 1993.

[47] Y. Khalidi and M. Nelson. Extensible File Systems in Spring. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, December 1993.

[48] D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison Wesley, 1973.

[49] P J. Leach and D C. Naik. A Common Internet File System Protocol. Interet draft, Internet Engineering Task Force (IETF), December 1997.

[50] E.K. Lee and R. Katz. Performance Consequences of Parity Placement in Disk Arrays. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–199, 1991.

[51] E.K. Lee and R.H. Katz. An Analytic Performance Model for Disk Arrays. In *Proceedings of the 1993 ACM SIGMETRICS*, pages 98–109, May 1993.

[52] S. J. Leffler, M. K. McKusick, M J. Karels, and J. S. Quartermann. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison Wesley, 1989.

[53] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of Real Time Systems Symposium*, pages 110–123, December 1992.

[54] C L. Liu and J W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 30:47–61, 1973.

[55] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. The Fellini Multimedia Storage Server. *Multimedia Information Storage and Management , Editor S. M. Chung, Kluwer Academic Publishers*, 1996.

[56] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[57] J. Menon and J. Cortney. The Architecture of a Fault-Tolerant Cached RAID Controller. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 76–86, May 1993.

[58] A. Merchant and P. S. Yu. Design and Modeling of Clustered RAID. In *Proceedings of the International Symposium on Fault Tolerant Computing*, pages 140–149, 1992.

[59] A. Molano, K. Juvva, and R. Rajkumar. Real-time File Systems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proceedings of IEEE Real-time Systems Symposium*, December 1997.

[60] Antoine Mourad. Reliable Disk Striping in Video-On-Demand Servers. Technical report, AT&T Bell Labs, 1995.

[61] R R. Muntz and J C.S. Lui. Performance Analysis of Disk Arrays Under Failure. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 162–173, 1990.

[62] G. Nerjes, P. Muth, M. Paterakis, Y. Romboyannakis, P. Triantafillou, and G. Weikum. Scheduling Strategies for Mixed Workloads in Multimedia Information Servers. In *Proceedings of the 8th International Workshop on Research Issues in Data Engineering (RIDE'98), Orlando, Florida*, February 1998.

[63] T. Niranjan, T. Chiueh, and G. Schloss. Implementation and Evaluation of a Multimedia File System. In *Proceedings of ICMCS'97, Ottawa, Canada*, 1997.

[64] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles, St. Malo, France*, October 1997.

[65] B. Ozden, R. Rastogi, P. J. Shenoy, and A. Silberschatz. Fault-tolerant Architectures for Continuous Media Servers. In *Proceedings of the ACM SIGMOD, Montreal, Canada*, June 1996.

[66] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw Hill, 1991.

[67] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Array of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD'88*, pages 109–116, June 1988.

[68] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[69] S. Pinker. *How the Mind Works*. W. W. Norton & Co., 1997.

[70] E. J. Posnak, S. P. Gallindo, A. P. Stephens, and H. M. Vin. Techniques for Resilient Transmission of JPEG Video Streams. In *Proceedings of Multimedia Computing and Networking, San Jose, CA*, pages 243–252, February 1995.

[71] P. Venkat Rangan and H.M. Vin. Designing File Systems for Digital Video and Audio. In *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP'91), Operating Systems Review, Vol. 25, No. 5*, pages 81–94, October 1991.

[72] A.L. Narasimha Reddy and J. Wyllie. Disk Scheduling in Multimedia I/O System. In *Proceedings of ACM Multimedia'93, Anaheim, CA*, pages 225–234, August 1993.

[73] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[74] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[75] R. Sahner and K. S. Trivedi. SHARPE: Symbolic Hierarchical Automated Reliability/Performance Evaluator, Introdution and Guide for Users. Technical report, Department of Computer Science, Duke University, September 1986.

[76] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the 1990 Winter USENIX Conference, Washington, D.C.*, pages 313–323, Jan 1990.

[77] P J. Shenoy and H M. Vin. Efficient Striping Techniques for Multimedia File Servers. In *Proceedings of NOSSDAV '97, St. Loius, MO, Extended version available as Technical Report TR96-27, Dept. of Computer Sciences, Univ. of Texas at Austin*, pages 25–36, May 1997.

[78] P J. Shenoy and H M. Vin. Efficient Support for Interactive Operations in Multi-resolution Video Servers. *ACM Multimedia Systems Journal (to appear)*, 1998.

[79] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts, 3rd e.d.* Addison-Wesley, 1991.

[80] S.S.Rao, H.M.Vin, and A. Tarafdar. Comparative Evaluation of Server-push and Client-pull Architectures for Multimedia Servers. In *Proceedings of NOSSDAV'96*, pages 45–48, April 1996.

[81] T. Teorey and T. B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM*, 15(3):177–184, March 1972.

[82] R Tewari, D M. Dias, R Mukherjee, and H M. Vin. High Availability in Clustered Multimedia Servers. Technical Report RC 20108, IBM, 1995.

[83] R. Tewari, H M. Vin, A. Dan, and D. Sitaram. Caching in Bandwidth and Space Constrained Hierarchical Hyper-Media Servers. Technical Report TR96-30, Department of Computer Sciences, Univ. of Texas at Austin, December 1996.

[84] F A. Tobagi, J Pang, R Baird, and M Gang. Streaming RAID – A Disk Array Management System For Video Files. In *Proceedings of ACM Multimedia '93, Anaheim, CA*, pages 393–400, 1993.

[85] K. S. Trivedi. *Probability & Statistics With Reliability, Queuing, And Computer Science Applications*. Prentice-Hall, Inc., 1982.

[86] C.J. Turner and L.L. Peterson. Image Transfer: An End to End Design. In *Proceedings of ACM SIGCOMM'92, Baltimore*, pages 258–268, August 1992.

[87] U Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.

[88] M. Vernick, C. Venkatramini, and T. Chiueh. Adventures in Building the Stony Brook Video Server. In *Proceedings of ACM Multimedia'96*, 1996.

[89] H. M. Vin, A. Goyal, and P. Goyal. Algorithms for Designing Large-Scale Multimedia Servers. *Computer Communications*, 18(3):192–203, March 1995.

[90] H. M. Vin, P. Goyal, A. Goyal, and A. Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In *Proceedings of the ACM Multimedia'94, San Francisco*, pages 33–40, October 1994.

[91] H. M. Vin, P. J. Shenoy, and S. Rao. Efficient Failure Recovery in Multi-Disk Multimedia Servers. In *Proceedings of the 25th International Symposium on Fault Tolerant Computing Systems, Pasadena, CA*, pages 12–21, June 1995.

[92] H.M. Vin, S.S. Rao, and P. Goyal. Optimizing the Placement of Multimedia Objects on Disk Arrays. In *Proceedings of the Second IEEE International Conference on Multimedia Computing and Systems, Washington, D.C.*, pages 158–165, May 1995.

[93] R. Wijayaratne and A. L. N. Reddy. Providing QoS Guarantees for Disk I/O. Technical Report TAMU-ECE97-02, Department of Electrical Engineering, Texas A&M University, 1997.

[94] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, Copper Mountain Resort, Colorado*, pages 96–108, Decmember 1995.

[95] J. Wolf, P. S. Yu, and H. Shachnai. DASD Dancing- A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems. In *Proceedings of ACM SIGMETRICS'95*, pages 157–166, 1995.

[96] B L. Worthington, G R. Ganger, and Y N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of ACM SIGMETRICS'94*, pages 241–251, May 1994.