Improving File System Performance With Adpative Methods

by

Jeanna Neefe Matthews

B.S. (Ohio State University) 1994

M.S. (University of California at Berkeley) 1997

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Commitee in charge:

Professor Thomas E. Anderson, Co-Chair

Professor Joseph. M. Hellerstein, Co-Chair

Professor David A. Patterson

Professor Alice M. Agogino

Fall 1999

The dissertation of Jeanna Neefe Mattthews is approved:

_____

Co-Chair Date

_____

Co-Chair Date

_____

Date

_____

Date

University of California, Berkeley

Fall 1999

Improving File System Performance With Adaptive Methods

by

Jeanna Neefe Matthews

Abstract

Improving File System Performance With Adaptive Methods

by

Jeanna Neefe Matthews

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Thomas E. Anderson, Co-Chair

Professor Joseph M. Hellerstein, Co-Chair

My thesis is that the systematic application of simple adaptive methods to file system design can produce systems that are significantly more robust to changing hardware and diverse workloads than existing systems. I present modifications to the log-structured file system that allow it to provide robust write performance in a wide range of environments. I also present a dynamic reorganization algorithm that makes disk layout responsive to read patterns. I evaluate these adaptive algorithms with trace driven simulation on a combination of synthetic and measured traces. I find that simple adaptive algorithms can dramatically improve worst case performance and can allow average case performance to scale with improvements in disk technology.

Table Of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the members of dissertation committee: Tom Anderson, Joe Hellerstein, Dave Patterson and Alice Agogino.

Tom is a researcher with a clear commitment to excellence. I have especially appreciated his thorough revisions and insightful comments on my written work. He is also a superb writer and communicator. I just hope that a little bit of it has rubbed off on me over the years!

I would like to thank Joe Hellerstein for being my local advisor in Tom's absence. He routinely went above and beyond the call of duty in that role, despite the demands of his own research and his own students. My last years at Berkeley would have been immensely more difficult it if it hadn't been for his willingness to get involved.

I am grateful to Dave Patterson for his focus on the overall development of graduate students. I consistently think back to the "How To Have a Bad Academic Career" talk he gave in his computer architecture course my first year. I marvel both at the depth and wisdom of his advice and at the time he invested in communicating it to young graduate students in such a compelling manner. It is just one example of the mentorship which is so obviously part of his life. It has been a privilege to work with a true master of his craft.

Alice Agogino is a wonderful role model. It is inspiring to see such a successful, female faculty member in action. I am grateful to have taken a class with her and to have had her on my committee.

I would especially like to thank my office mates throughout my time at Berkeley. I cannot imagine my time at Berkeley without them.

In my first semester, I was assigned to a bull pen office for first year graduate students on the third floor of Soda with Armando Fox, Angie Schuett, Eric Anderson and others. We did projects together, went dancing together, instituted a weekly Disney movie night and overall settled into our new lives together. In retrospect, I realize what a gift it was to bond so immediately with such a wonderful group of people. They helped make Berkeley feel like home from the very start.

When I joined the NOW project, I moved into 475 Soda with the xFS file system crew: Mike Dahlin, Randy Wang and Drew Roselli. We laughed; we fought; we danced the xFS victory dance; we stayed up all night thinking "good thoughts" while our benchmarks ran. I am truly thankful for that camaraderie and for always having a place I could ask even the silliest question. Mike Dahlin was the senior graduate student in our group and a great mentor to me. He inspired me and others to work on his vision because he was so able and willing to teach as he went. Randy Wang is amazing in his ability to make things really work. I firmly believed that with a twirl of his pen, a little classical music and one good night of hacking, he could make anything work. Drew Roselli has been my partner in crime and my compatriot for the last several years. Together, we have weathered administrators who didn't want to let us turn in our tech reports, renew our copy cards, etc. without an advisor, various difficult NTU students, and mountains of general angst. Being in it together made all the difference in the world! Drew is a truly talented individual who isn't afraid to follow her own inspiration. Adam Costello joined our office joined our office as the xFS project was winding down. He is a truly kind person who has many, many times gone out of his way to help me or others. By example, he has taught me so many things, like the value of doing things right the first time and of exploring a tangent.

I would also like to thank the other members of the NOW project —for bus rides, for sled rides, for long lunches, for help cheerfully given and for honest yet kind critiques.

I would like to thank my friends from outside the computer science department for their support, for their prayers and for moments of sanity and perspective. I would especially like to thank Andrew and Susan Dickens and their family, Taylor and Youla Overby and their family, Cyndi Diaz , Pam Norton, and Matthew Artzen.

I would certainly like to thank my family. My mother, JoAnn Krach, and my grandmother, Wilda Motts, especially, have always believed in me, given me unconditional love and acceptance and encouraged me to push just a little more. My husband's parents, Robert and Barbara Matthews, welcomed me into their lives and gave me a whole second family to belong too.

Finally, I cannot find words to express my overwhelming thanks to my husband Lenny for his love, his wisdom and his absolute guts and determination. I am truly blessed to have such a talented and dynamic partner with whom to share my life. Lenny, you make this life a joy and an adventure every day! I would also like to thank my children Robert and Abigail for their patience and their sweetness. They truly are the delight of my heart.

Above all, I would like to thank God for being my sufficiency in all things — especially graduate school.

1 Introduction

1.1. Motivation

Operating systems sit on the boundary between applications and the raw hardware (Figure 1-1). Their mission is to marshal the best combination of hardware resources to efficiently and fairly satisfy the needs of all the applications using the system. They enable diverse applications to co-exist in the same system by protecting applications from each other and by providing communication and resource sharing between applications. They provide a simplified abstraction of the raw hardware, yet they are expected to deliver to the end user the full performance potential of the hardware. Also, it is important for operating systems to be highly reliable because the reliability of the applications on which users depend can be no better than that of the underlying system.

Operating system design is especially challenging because applications place diverse demands on the system. For example, applications may have different data integrity constraints or may perform best with different cache replacement strategies. Applications as varied as scientific simulations, software development and multi-media presentations may be running together on a single computer system.

To make matters worse, operating systems implementations often live through many generations of hardware technology. Operating system software changes slowly because any modification can introduce unexpected errors, requiring costly testing procedures to reverify correctness to meet user demands for reliability. On the other hand, hardware changes rapidly. Processors, mem-

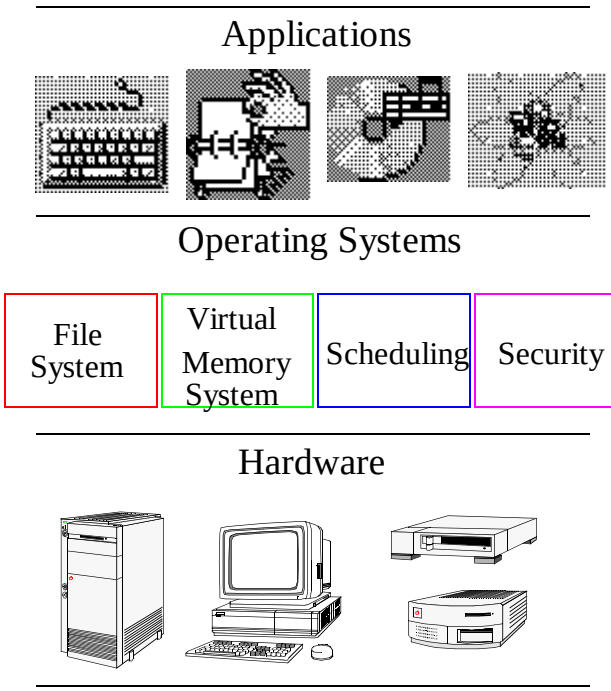

FIGURE 1-1. **Operating systems sit on the boundary between applications and the raw hardware.**

ory and disks are all improving exponentially, but a different rates. Each year, the number of cycles needed for an average memory access changes, the relative price of memory and disk changes, etc. To be efficient, operating systems must shift work away from the pieces of hardware that are limiting the system. With each hardware resource changing at a different exponential rate, system bottlenecks change over time.

For the most part, operating systems designers have dealt with this challenging situation by optimizing for a fixed point on the moving target of hardware and workload characteristics. The typical model is as follows. Determine a “common case” workload to use as a benchmark. Then, tune the system for this benchmark and for the characteristics of the hardware available at the time the system is designed. In this model, workloads with different requirements than the “common

case" benchmark may not be well served. In addition, as hardware improves or workload changes, a static operating system may become untuned, even in the common case.

Operating system designers have sometimes dealt with the problem of diverse environments by developing interfaces by which users, system administrators or application writers may alter operating system behavior to suit their needs. The simplest example is the exposing of parameters that can be set when the operating system is installed or before it begins executing. Exposing such parameters is an improvement over statically optimized systems because they allow the system to be adjusted based on the workload and hardware factors that do not change as the system is running.

However, the multiple levels of abstraction in modern computer systems hide much activity from the end user and even an experienced user or expert system administrator may have difficulty choosing optimal parameter settings for their environment. For a concrete example of the difficulties of setting parameters, consider SAM, a system administrator tool for HP-UX with a GUI interface designed to simplify routine system administration tasks. Under kernel configuration and configurable parameters, SAM allows about 100 parameters to be configured. Examples include the scheduling interval, the number of priority levels for the scheduler, the percentage of available memory to allocate to the buffer cache, maximum write delay time, read-ahead window and the size of a second-chance paging list. The current value of each parameter is listed and often a short description of its purpose. Some like *nproc* ("Maximum Number of Processes") are fairly intuitive. However, others like *no_lvm_disks* ("Boolean; Set only if systems has no LVM disks") are less clear. Others like *acctresume* ("Threshold to Resume Accounting") do not indicate any relevant units. Still others like *vme_io_estimate* contain no description at all.

Designing extensible systems is another approach to dealing with the problems of heterogeneous environments. Extensible systems provide user-level access to interfaces through which

application writers can implement their own versions of system services. This provides even more flexibility to alter system behavior. It allows not only the parameters of a single implementation to be customized; it allows the entire implementation to be customized.

Extensible systems are especially attractive for those application writers who understand the needs of their applications extremely well and whose applications are not currently well served by the generic services provided by the operating system. Without extensible systems, these application writers might be tempted to write an entire operating system customized to their application. Extensible systems offer them the opportunity to implement only those services to which their applications are extremely sensitive. However, implementing an operating system extension is even more complicated than setting a multitude of exposed parameters. Thus, the benefits of extensible systems are limited by the number and applicability of customized implementations.

Furthermore, extensible systems have difficulty providing communication between extensions and in enforcing protection and performance isolation between them. Allowing each application to choose its own version of system services has similar problems to allowing each citizen of a society to choose a government customized to their needs. Finally, even an extension customized for a specific application must deal with a rapidly changing hardware base and most likely even diverse workload patterns from different users of the same application.

Ideally, these problems could be avoided if operating systems could be designed to adapt themselves to their changing environment. However, changing in response to every variation in environment would be prohibitively expensive and complex. It would require: 1) monitoring every detail of a constantly changing environment, 2) implementing every possible alternate system behavior and 3) modeling the trade-offs between all these alternatives.

This suggests some interesting questions. Could designers take just a few of the configurable parameters they have exposed and find a way to set them automatically? Are there any situations in which the system could automatically identify the best extensible module for a given workload? If measuring every detail of the system is too time-consuming, would it be possible to measure a few key parameters with significant impact on system performance or robustness? If it is too difficult to anticipate every possible change in technology, could the system at least be prepared to scale with foreseeable technology trends? If it is too complicated to model the system in enough detail to achieve optimal performance in every situation, could a simple model at least avoid dramatically lower worst case performance? If it would be prohibitively expensive to implement every possible alternative behavior, could just a few options satisfy a significantly larger fraction of environments than just one static policy? This thesis argues that the answer to these questions is yes — that the systematic application of simple adaptive methods can produce systems of reasonable complexity that are significantly more robust than existing systems to changing hardware and diverse workload characteristics.

1.2. Adaptive Methods

An adaptive method is an algorithm that alters system behavior by:

1. Measuring the characteristics of available hardware resources.
2. Measuring the characteristics of the workloads applications place on the system.
3. Using analytical models to balance the trade-offs between possible system actions.

Least-recently-used caching is an example of a very simple adaptive method which takes into account the access stream and the fact that the cache can be accessed more quickly than its backing store. Database query optimizers are a more complex example. They consider many parameters including the number of tables being accessed, the number and type of indices on these tables, any

join criteria between the tables and the way in which the tables are stored [Selinger et al., 1979][Mackert and Lohman, 1986].

Adaptive methods have been successfully applied to many areas of computer science. To name just a few examples: In networks, TCP includes algorithms to adapt retransmission timers and window sizes to avoid congestion in the network [Jacobson and Karels, 1988]. In architecture, reconfigurable computing tailors and dedicates functional units to take advantage of application dependent dataflow [Villasenor and Mangione-Smith, 1997]. In distributed systems, implicit coscheduling uses locally observed round trip time and message arrivals to adaptively schedule the pieces of a parallel program at the same time on separate machines [Arpaci-Duseeau et al., 1998]. In storage systems, AutoRAID moves data into the faster mirrored region from the RAID-5 region as it is accessed more frequently and also adjusts the amount of space in the two regions based on how full the storage system is [Wilkes et al., 1995]. In compilers, runtime systems have been developed to re-optimize code based on profiling of the instruction stream or based on hardware characteristics like the cache size or pipeline characteristics [Compaq Digital, 1999][Java, 1999]. In control theory, adaptive methods are formalized as the optimization of functions of dynamically changing variables in order to prove when a solution is optimal or robust [Feiertag and Organick, 1971][Clark, 1992].

1.3. Adaptive Methods In File System Design

In this thesis, I focus on developing simple and effective adaptive methods for file systems. The file system is the part of the operating system that is responsible for the correct and efficient retrieval of data from stable storage. File systems are a particularly interesting case study for the application of adaptive methods for several reasons. First, file system performance is critical to the overall performance of the system due to the widely recognized “I/O gap” which has occurred as

improvements in processor and memory technology have far outpaced improvements in disk technology. Second, extensible file system designs have difficulty dynamically sharing the physical storage media between multiple implementations. Third, the file system has high reliability requirements because the rest of the system depends on the file system to correctly retrieve executables and data from stable storage; therefore, the trade-offs between complexity and performance for adaptive file systems methods are particularly interesting.

File system designers, like all operating system designers, must cope with a wide variety of usage patterns. In some environments, files tend to be small and to be read and written sequentially in their entirety [Baker et al., 1991]; other environments are dominated by random access to large files [TPC-C, 1990]. In some environments, there are frequent idle periods in which the system can perform optimization activities without disturbing users [Blackwell et al., 1995]; in other environments, the system is constantly responding to application requests [Network Appliance, 1997a]. In some environments, there is a high degree of locality in the set of data that is actively accessed [Baker et al., 1991]; other environments are dominated by streaming access to large files [Bolosky et al., 1996].

File system designers are also faced with the rapid evolution of modern storage systems. A single file system may be used to store data on a single disk or on a complex storage array with many disks. Even the characteristics of individual disks have been rapidly changing. Rapid improvements in aerial densities enabled by precise optical positioning of disk heads has led to dramatic improvements in disk capacity and bandwidth; improvements in disk latency continue to be stalled by mechanical access delays.

File systems must also deal with the fundamentally different performance characteristics of read and write access. For the durability of writes across system-crashes, it is not crucial where on disk new data is placed. Many systems have used this fact to improve write performance by batch-

ing writes [Rosenblum and Ousterhout, 1992], reordering writes [Seltzer et al., 1990] or placing writes into the nearest free disk location [Chao et al., 1992][English and Stepanov, 1992][Hitz et al., 1995][Wang et al., 1999]. These same techniques are not well suited to improving read performance; reads must be directed to the position of the last write — not any convenient location. In fact, these techniques which are effective at improving write performance can actually decrease read performance by scattering related data as it is written.

Nowhere is the need for adaptive methods more apparent than in the area of file system layout policies. File system layout policies must provide stable and robust performance across a wide range of workloads, must balance the differing needs of read and write access patterns and must provide high performance access to rapidly evolving storage systems. However, file system design has been dominated by a variety of static layout policies all of which fail in at least one of these criteria. The Berkeley Fast File System [McKusick et al., 1984] does not expose the high performance possible from modern storage systems. The Log-Structured File System [Rosenblum and Ousterhout, 1992] does not provide stable performance across a wide range of workloads [Seltzer et al., 1995]. Write-Anywhere systems [Chao et al., 1992][English and Stepanov, 1992][Hitz et al., 1995][Wang et al., 1999] do not balance the differing needs of read and write access patterns. To date, file system designers have not combined the best of these various systems. Instead, the discussion of layout policies has been dominated by arguments over what “common case” workload should be used to evaluate them or what hardware resources are reasonable to assume in every system.

1.4. Contributions of this Dissertation

My thesis is that the systematic application of simple adaptive methods to file system design can produce systems that are significantly more robust to changing hardware and diverse work-

loads than existing systems. This dissertation contains two main categories of optimizations — those targeted at improving write performance and those targeted at improving read performance.

I contribute to providing high write performance by designing adaptive methods that allow a log-structured file system (LFS) to provide robust write performance in a wide range of environments. I begin with LFS because its common case write performance is excellent and in fact, it is considered a write-optimized file system. In addition, it has many other benefits including ease of recovery and the ability to scale with technology trends that favor large transfers to disk. However, in certain cases, LFS exhibits dramatically lower performance due to the overhead of its *cleaner* or garbage collection process. This problem with LFS, pointed out in [Seltzer et al., 1993] and [Seltzer et al., 1995], led many to discount LFS as an interesting but impractical design. In this dissertation, I demonstrate that by incorporating adaptive methods into the system, LFS can retain its excellent common case performance and perform as well as other alternatives in the situations for which it formerly had poor performance.

I make the following contributions to improving LFS write performance:

1. Large transfer sizes make the most efficient use of disk bandwidth, but I show that they have a contervailing detrimental effect on garbage collection overhead. I show how to choose the LFS segment size by trading transfer efficiency against cleaning efficiency.
2. I present an adaptive garbage collection algorithm that combines traditional LFS cleaning with an alternate garbage collection mechanism called hole-plugging [Wilkes et al., 1995]. This algorithm dynamically adapts to changes in disk utilization and workload to avoid the traditional LFS performance cliff at high disk utilizations, while still preserving the advantage LFS has at lower disk utilizations.
3. I show how to further reduce cleaning overhead by taking advantage of cached segments when cleaning.

Together, these enhancements make LFS more stable over a wider range of workloads by eliminating its dramatic worst case performance. Overall, these modifications improve LFS write performance by up to a factor of four and reduce cleaning overhead by up to a factor of six.

I also contribute to providing robust read performance by designing adaptive methods that identify and cluster together groups of related data. These algorithms are relevant to LFS as well as other file system architectures. LFS can benefit from adaptive data reorganization when read patterns do not match write patterns or when cleaning disturbs the temporal locality of the original log. Write-anywhere systems can benefit from adaptive data reorganization because they scatter data to the most convenient location as it is written. Even read-optimized systems which attempt to group whole files and directories together can benefit from adaptive data reorganization when read patterns do not follow the directory hierarchy or when the disk becomes fragmented and they are forced to allocate files in many small discontiguous fragments.

I make the following contributions to improving file system read performance:

1. I demonstrate how existing layout policies which are not adaptive fail to provide robust read performance when read patterns do not match the default layout or when the default layout is disturbed by fragmentation or garbage collection.
2. I present a dynamic reorganization algorithm that uses past read access patterns to predict future read patterns and to group data together on disk accordingly.
3. Within the context of LFS, I show how dynamic reorganization can be used to augment a write-optimized system by suggesting targeted improvements to existing data layout.

Together, these optimizations allow disk layout to change over time to support efficient read access and enable the system to balance the competing demands of read and write access.

1.5. Organization of the Dissertation

The body of this thesis consists of 6 chapters.

Chapters 2 through 4 examine each layer of Figure 1-1 on page 2 as they relate specifically to file systems. In Chapter 2, I discuss trends in hardware technology. In Chapter 3, I give a brief history of file system design and describe existing file system designs. In Chapter 4, I discuss variations in workload characteristics and how they affect the relative merits of the various file system architectures. Together, Chapters 2 through 4 motivate the need for adaptive methods in file system design.

Chapters 5 and 6 present various adaptive file system methods and show their benefit to existing file system designs. In Chapter 5, I show how to improve the write performance of log-structured file systems by the application of adaptive methods to reduce the cleaning overhead. In Chapter 6, I show how to improve read performance and how to balance read performance and write performance with a dynamic reorganization algorithm.

Finally, Chapter 7 summarizes the key conclusions of this dissertation, points out some of the lessons I have learned while doing this work, and discusses areas that would benefit from further investigation.

2 Technology Trends

In this chapter, I discuss trends in hardware technology that are especially relevant to file system performance and to which it is most important to adapt.

2.1. Overview

"I/O certainly has been lagging in the last decade."— Seymour Cray (1976)

"I/O's revenge is at hand" — Hennessy and Patterson (1996)

Storage systems have increasingly become the bottleneck in modern computer systems because improvements in mechanical disk access latency, such as seek and rotational delay, have failed to keep pace with improvements in integrated circuit technology. While processor speed has been doubling approximately every 18 months, the rate at which the disk can position the head has been doubling only every ten years! This situation is commonly referred to as the I/O bottleneck or the "I/O gap".

Disk bandwidth, however, has been improving much faster than disk latency due to rapidly increasing aerial densities. This indicates an opportunity to narrow the I/O gap with techniques that scale with disk bandwidth rather than latency. This has placed a premium on finding ways to avoid latency by placing data in nearby disk locations or to amortize latency with larger transfer sizes. Although main memory latency has not improved dramatically, it is still much faster than

disk in absolute terms. However, disk technologies are improving at a faster rate than memory technologies in access times, bandwidth and cost/capacity.

Table 2-1 summarizes recent rates of improvement for processors, memory and disk. In the sections that follow, I document and discuss these trends and their implications for file system design. Documenting technology trends requires access to historical cost and performance data with accurate dates. Such data can sometimes be difficult to find. Many hardware vendors only provide prices and performance data for currently shipping products. Many provide specification sheets for older products, but with no corresponding date of introduction. Examining the advertisements in past issues of popular computer magazines like *Byte* or *PC Magazine* can be good source for some types of historical data, especially price data. However, advertisements do not always provide detailed performance information. Independent benchmarking organizations exist for some products and can provide excellent historical information by way of dated publications comparing available products. Despite these difficulties, documenting historical trends is important to prepare for the cumulative effects of exponential improvements over time.

TABLE 2-1. **Summary of Recent Rates of Improvement for Processors, Disks and Memory** This summaries recent technology trends. The trends are discussed and illustrated in greater detail in the sections which follow. The rate for processor performance improvement was computed from performance ratings reported in periodic SPEC newsletters (Section 2.3). The rates for improvements in disk and memory characteristics are as reported in Professor David Patterson's keynote address at SIGMOD98, "Hardware Technology Trends and Database Opportunities".

	Rate of Improvement	**Performance Doubles In**	**1998 Baseline**	**Discussed in**
Processor Performance	57% per year	1.5 years	10-15 SPECINT95	Section 2.3
Disk Access Latency	8% per year	9.0 years	9 milliseconds	Section 2.2
Disk Bandwidth	40% per year	2.1 years	15-21 MB/sec	Section 2.2
Disk Cost/Capacity	60% per year	1.5 years	$0.09/MB	Section 2.2
Memory Access Latency	7% per year	10 years	10 nanoseconds	Section 2.4
Memory Bandwidth	20% per year	3.8 years	800 MB/sec	Section 2.4
Memory Cost/Capacity	25% per year	3.1 years	$1.19/MB	Section 2.4

2.2. Trends in Storage Technology

Magnetic disks are the dominant technology used for stable on-line storage of data. In Section 2.2.1, I briefly review the internals of a disk drive. In Sections 2.2.2 through 2.2.5, I discuss trends in magnetic disk technology and some relevant facts about the disk drive industry.

2.2.1. Internal Structure of a Disk Drive

Figure 2-1 illustrates the internal structure of a disk drive. A disk drive consists of a number of platters anchored to a central spindle. Normally, the platters are covered on both sides with a thin coating of magnetic media which serves are the recording media. There is one disk arm which contains a read/write head for each recording surface. All the read/write heads share a common data channel. Therefore, only one head may be active at any one time.

Each platter is divided into concentric circles called tracks. A cylinder is the set of tracks, one per platter, that are at the same offset from the spindle. Each track is further divided into sequential units called sectors. A sector is of fixed size (often 512 bytes) and is the smallest addressable unit of data within the disk drive. Some drives guarantee the atomicity of single sector writes (i.e. the entire sector will be completely changed to the newly written value or none of it will). In drives that do not guarantee the atomicity of sector writes, a power failure can corrupt the current sector being written. In either case, writes involving more than one sector are not guaranteed to be atomic.

Most modern disks divide the disk into groups of adjacent cylinders or zones in order to take advantage of the fact that the outer tracks are larger and able to store more data than the inner tracks. Within each zone, there are a fixed number of sectors per track; outer zones have more sectors per track and therefore higher bandwidth. Multiple zones provides higher capacities and higher bandwidths for some areas of the disk. However, zoned disks are more costly and complex

because the drive's microprocessor must interpret/produce a different frequency of magnetic signal for each zone [Schwaderer and Wilson, 1996].

In order to access a specific disk address, the disk arm first moves or seeks to the proper cylinder. The time required to perform this action is called seek time. Seek time is an non-linear function of the number of cylinders that must be traversed. Long seeks involve first speeding up the disk arm to cover some distance and then slowing it back down as the desired cylinder approaches. A one track seek is relatively inexpensive because the tracks are so close together that moving the head over one track can be considered a special case of the minute adjustments, compared to the physical size of the disk head, required to keep the head positioned when reading or writing. Average seek times may be lower for reads because reads can probabilistically sense the data even before the head has completely settled above the correct track.

Once the disk arm is located over the proper cylinder, the disk platters must rotate until the desired disk location is directly under the read/write head. The time required for this is called rotational latency. The sum of seek and rotational delay along with overhead added by the disk controller is sometimes referred to collectively as head settling time or access latency. Finally, transfer time is the time required for the desired data to pass under the head.

Disk drives contain more storage space than is exposed to the file system. First, each sector contains a header and error correction codes by which positioning and recording errors can be detected and repaired. Second, drives are formatted with space sectors which can be used to replace normal data sectors in which frequent recording errors have been detected.

2.2.2. Improvements In Aerial Density

Magnetic disk technology has seen dramatic improvements in the density of data stored on the recording surfaces. The first disk drive produced was the RAMAC in 1956. It was the size of two

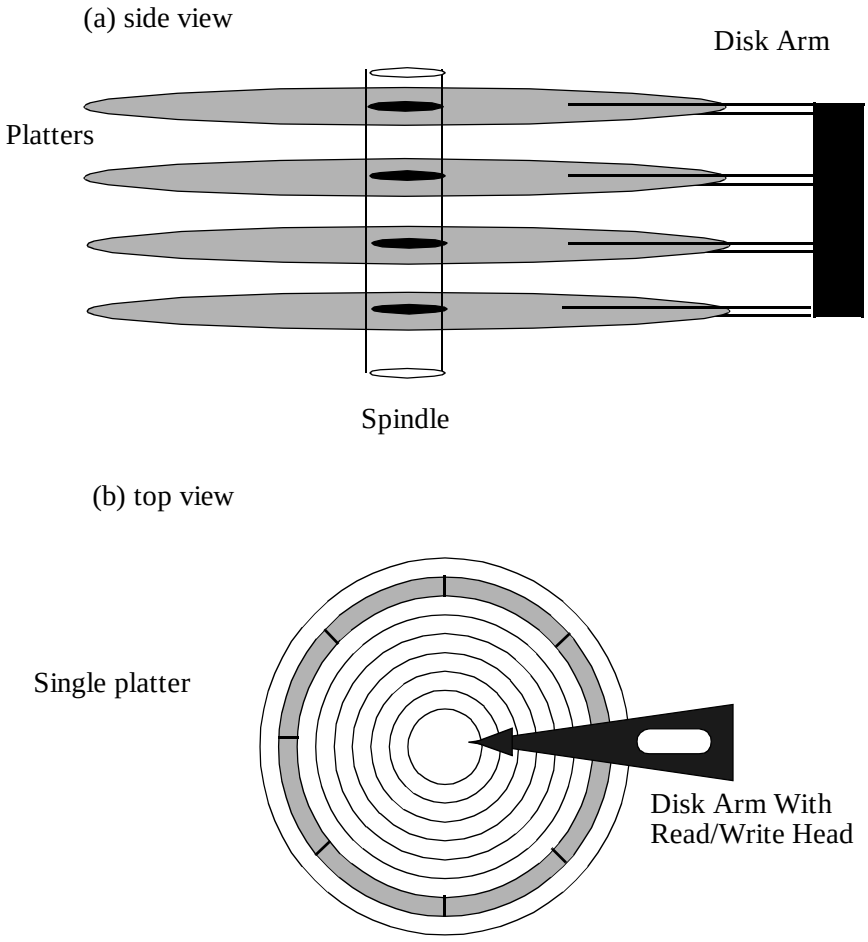

FIGURE 2-1. **Internal Structure of a Disk Drive** This figure gives a high-level view of the internal structure of a disk drive. The top illustration shows a side view of four platters attached to a central spindle and a disk arm with read/write heads for each recording surface. The bottom illustration shows the top view of a platter and the disk arm with the read/write head. The shaded area of the platter represents a single track, divided into eight sectors. In reality, modern disks can have tens of thousands of tracks with hundreds of sectors per track.

refrigerators and contained 50 platters, each 24 inches in diameter. That is a far cry from modern drives with two 2.5 inch platters — especially considering that the modern drives hold over 200 times as much data! Over the entire period 1956 to present, aerial densities have improved at an average rate of 33% per year[1]. More recently, aerial density has been improving at about 60% per year [Grochowski, 1998] due to developments in magnetoresitive (MR) heads and better magnetic media [Schwaderer and Wilson, 1996]. These improvements in data density have been used to fuel both larger capacity and smaller form factor drives.

Indications are that current increases in data density are being channeled into higher capacity, rather than smaller form factor, drives. Tables 2-2 and 2-3 present data from the 1999 DISK/TREND report [DISK/TREND, 1999]. Table 2-2 indicates that disks of larger and larger capacity will dominate the market; while Table 2-3 indicates that the form factors will remain fairly constant. Five and a quarter and three and half inch drives are typically used in desktop computers with three and half inch drives taking the lead. Two and half inch drives have become the standard for notebook computers. Smaller drives have been demonstrated, but applications with reasonable volume have not been demonstrated [Schwaderer and Wilson, 1996]. Large capacity disks present a problem for file system architectures whose crash recovery time scales with total storage capacity.

1. This was calculated from data provided by Quantum Corporation. They list the MB/in^2 for disks representing firsts in disk drive technology from 1956 through 1995 [Quantum, 1999a].

TABLE 2-2. **Percentage of Disk Drive Revenues By Drive Capacity** The data in this table is from the on-line portion of the 1999 DISK/TREND Report. The table lists the percentage of each year's total revenue represented by sales of disks of various capacities. Total revenues in 1998 were approximately $30 billion and are projected to be approximately $36 billion and $50 billion in 2000 and 2002 respectively.

Percentage of sales revenues of fixed disk drives	**1998 Revenues**	**2000 Revenues (Forecast)**	**2002 Revenue (Forecast)**
Less than 2 GB	1.20 %	0.16 %	0.19 %
2-3GB	13.00 %	0.94 %	0.00 %
3-5 GB	41.00 %	4.30 %	0.30 %
5-10 GB	31.00 %	21.00 %	1.50 %
10-20 GB	13.00 %	39.00 %	5.80 %
20-40 GB	0.21 %	28.00 %	20.00 %
40-80 GB	0.53 %	5.70 %	37.00 %
More than 80 GB	0.00 %	0.47 %	35.00 %

TABLE 2-3. **Percentage of Disk Drive Shipments By Diameter** The data in this table is from the on-line portion of the 1999 DISK/TREND Report. This table lists the percentage of each year's total worldwide unit shipments for disks of various diameters. Total worldwide unit shipments in 1998 were approximately 145 million units and are projected to be approximately 194 million units and 253 million units in 2000 and 2002 respectively.

Percentage of Worldwide Unit Shipments	**1998 Shipments**	**2000 Shipments (Forecast)**	**2002 Shipments (Forecast)**
5.25 inch	2.80 %	0.56 %	0.00 %
3.5 inch	85.00 %	87.00 %	87.00 %
2.5 inch	12.00 %	12.00 %	13.00 %
1.8 inch or less	0.08 %	0.13 %	0.30 %

2.2.3. Improvements In Access Time

Unfortunately, historical disk performance characteristics with accurate dates are difficult to find. Some disk manufactures supply specifications for drives no longer being sold, but do not indicate the year a drive was first offered. Quantum Corporation provides an interesting on-line list of "Firsts in Disk Drive Technology", but the parameters listed for each drive do not include access latency and the last drive listed was introduced in 1995 [Quantum, 1999a]. Advertisements

in popular computer magazines list the interface and the capacity of each drive, but not detailed performance information.

Patterson recently summarized recent hardware trends in a key note address at the 1998 ACM SIGMOD conference [Patterson, 1998]. Table 2-4 lists disk performance characteristics given in that presentation. In addition, in Table 2-5, I list the performance characteristics for four disk drives produced in the last decade. In the first three columns, I use the dates and performance parameters reported by Garth Gibson [Gibson, 1992] and Mike Dahlin [Dahlin, 1996] in similar discussions of technology trends and add a more recent data point for the Seagate Cheetah [Seagate, 1998]. I compute the average rate of improvement for each parameter listed. The rates at which seek and rotational delay are improving closely match the trends reported in [Patterson, 1998]. Recently, however, disk bandwidth has been improving more rapidly than the historical average due to recent improvements in disk head technology [Grochowski, 1998].

TABLE 2-4. **Current Disk Performance Characteristics** This data is taken data presented in Professor David Patterson's keynote address at SIGMOD98, "Hardware Technology Trends and Database Opportunities". Access latency refers to the time for an average seek plus one-half rotation.

	Improvement Per Year	**1998 Baseline**
Disk Access latency	8%	9 milliseconds
Disk Bandwidth	40%	15-21 MB/sec
Disk Cost/Capacity	60%	$0.09/MB

TABLE 2-5. **Trends In Hard Disk Performance Characteristics.** This table compares a series of disk drives from 1987 through 1998. Performance data for the first three drives are taken from similar discussions of technology trends found in [Gibson, 1992] and [Dahlin, 1996]. Performance data for the 1998 drive is taken from the product specification sheet available from the manufacturer [Seagate, 1998]. For each drive, average seek time, rotational speed, average rotational latency and maximum bandwidth are reported. In addition, the average time for an 8KB and a 1MB transfer are computed. They include the time for one average seek (read), one half rotation and the transfer size divided by the maximum bandwidth. Finally, the half-power point for each drive is computed by multiplying the average access time (one average seek (read) plus one half rotation) times the maximum bandwidth.

	1987	**1990**	**1994**	**1998**	**Improvement Rates (%/year)**
Model	Fujitsu M2361A	Seagate Elite 1 ST41600N	Seagate Barracuda 4 ST15150N	Seagate Cheetah 18 ST118202LW	
Average Seek Time (ms)	16.7	11.5	8.0/9.0 (read/write)	6.0/6.8 (read/write)	9 %
Rotational Speed (RPM) Average Latency (ms)	3600 8.3	5400 5.56	7200 4.2	10,025 2.99	9 %
Maximum. Bandwidth (MB/sec)	2.5	5	9	29	25 %
8 KB Transfer (ms)	28.3	18.9	13.1	9.3	10 %
1 MB Transfer (ms)	425	244	123	43.5	19 %
Half-power point (KB)	62.5	85.3	110	261	14 %

Seek and rotational latency has been improving much more slowly than data density because they are limited by mechanical delays. Rotational latency is determined by how fast the platters can spin and by the maximum rate of the data channel in the disk arm. Spindle speed has increased by about 9% per year from 3600 RPM in 1987 to over 10,000 RPM in 1999. Seek time is determined by the time it takes the disk arm to move across the platters. This time has decreased with the decreases in platter diameter and with improvements in disk arm actuator technology, but improvements have been limited to approximately 9% per year. If platter diameter remains fairly constant as predicted by Table 2-3, these improvements may slow even further.

Disk bandwidth is partially determined by the data density; at greater data densities, the amount of data passing by the head becomes greater even at the same rotational speed. Disk band-

width also improves with increases in rotational speed. As a result, as indicated in Table 2-5, disk bandwidth has been improving at a greater rate than seek and rotational delay.

Table 2-5 also indicates trends in total transfer time for both a small 8KB access and a large 1MB access. The small transfer is almost completely dominated by seek and rotational delay; while the large transfer scales with disk bandwidth. This illustrates an opportunity to narrow the I/O gap with large transfers if the file system is able to successfully group large groups of data together for sequential transfer.

One way to discuss the balance between access latency and bandwidth is the *half-power point*, or the transfer size required to achieve half of the raw bandwidth potential of a disk. The half-power point is the product of seek and rotational latency and bandwidth. It represents the point at which half of the total access time is spent in positioning the disk head and half of the time is spent in transferring data to or from the media. As disk bandwidth increases more rapidly than seek and rotational delay, this half-power point is steadily increasing. Therefore, each generation of disk technology requires larger transfers in order to achieve the same percentage of the raw transfer rate available from disk. Based on the four sample drives in Table 2-5, the half-power point has been increasing by approximately 14% per year over the past ten years. With disk bandwidth improving at a rate of 40% per year and seek and rotational delay at 8% per year, this half-power point is currently increasing at a rate of almost 30% per year.

$$HalfPower2 = Latency2 \times Bandwidth2 = (1 - 0.08) \cdot Latency1 \times (1 + 0.40) \cdot Bandwidth1$$

$$= 1.29 \times Latency1 \times Bandwidth1 = 1.29 \cdot HalfPower1$$

Because disk bandwidth is improving more rapidly than disk access latency, using single disks efficiently requires amortizing access latency over larger transfers or avoiding access latency by accessing nearby disk locations.

2.2.4. Disk Arrays

According to the 1998 and 1999 DISK/TREND reports, disk arrays account for 30% of the total fixed storage market.[1] Further, this percentage is forecasted to stay fairly constant through 2001. Disk arrays in single user systems are rare. Most disk arrays are placed in midrange to mainframe environments. A growing open system server market has led to a variety of network attached disk array systems.

The configuration of disk arrays is normally described by RAID (Redundant Arrays of Inexpensive Disks) levels 0 through 5 [Gibson, 1992]. The most commonly used levels are 0, 1 and 5. RAID-0 actually contains no redundancy and is simply an array of independent disks that stripe the data. In RAID-1, each disk is fully mirrored on another disk, reducing the useful storage capacity in half in exchange for much higher data availability. In RAID-5, the total storage capacity of the system is logically divided into stripes of data that stretch across the N disks in the system. For each stripe, each disk contains a data block. Of the N blocks in each stripe, N-1 contain actual data, while one contains the parity block. A parity block is the logical bitwise OR of the N-1 data blocks. It is a piece of redundant information that allows any one missing block in the stripe to be reconstructed from the information contained in the other N-1. Therefore, RAID-5 allocates 1/Nth

1. The on-line portion of the 1998 DISK/TREND Report lists the total disk array market at $11 billion in 1997 and forecasts $13 billion for 1998. The on-line portion of the 1999 DISK/TREND Report lists the single rigid disk drive market at $30 billion in 1998. Unfortunately, both full reports would cost over $10,000. I used the forecast for the 1998 disk array market and the reported 1998 single rigid disk drive market to compute the 30% average. The projections for 2001 from the two reports are $16 billion for disk arrays and $43 billion in single rigid disk drives, giving a projected 27% of the market to disk arrays.

of the total storage capacity to increase data availability, but increases the cost/capacity of the system at the same time.

In addition to increased reliability, disk arrays can offer the potential to utilize the aggregate bandwidth of many disks. A single transfer of data that spans many disks or multiple transfers that access different disks will achieve higher throughput. However, disk arrays typically have higher access latency due to added controller delay and network delay in the case of network-attached systems. In addition, parity schemes suffer an additional penalty for single block write transfers because the parity block must be read and updated in addition the actual data transfer.

2.2.5. Storage System Interfaces

Despite their internal complexities, disks present a relatively simple interface to the file system: total capacity of a storage device as an array of fixed-size sectors. Techniques exist for looking beyond this simple interface and extracting the relevant performance characteristics of disk drives [Worthington et al., 1995][Talagala et al., 1999]. [Worthington et al., 1995] present several tools for on-line extraction of SCSI disk parameters. [Talagala et al., 1999] presents a series of three benchmarks which can be used to extract a multitude of disk performance parameters from any disk. The parameters identified dynamically by such tools include the number of heads, head switch time, cylinder switch time, sectors per track, bandwidth, rotational latency and a complete seek profile. Benchmarks such as these can be run by the file system when a new storage system is attached to the system. The benchmarks in [Talagala et al., 1999] run in under two seconds. The results provide data needed by adaptive methods that are sensitive to disk performance characteristics.

In addition, single disk drives use a predictable scheme for mapping the array offsets used by the file system onto actual physical locations. In fact, disks map the array offsets specifically to

minimize the time between subsequent addresses. As a result, the file system can use subsequent logical addresses as a good predictor of adjacent physical disk locations. For example, if array[n] is the last block on a track, array[n+1] will be on the next track in the same cylinder and will be rotationally offset to allow time for the head switch time. Similarly, if array[n] is the last block in a cylinder, arrray[n+1] will be on the first track on the next cylinder and will be rotationally offset to allow the time for a one cylinder seek. Over time, the disk may alter this mapping slightly to replace bad sectors with space sectors, but the effect of this error remapping should be minor. If the effect becomes significant, the file system could periodically run benchmarks to detect the remapping.

Disk arrays present the same simple array interface to the file system. However, disk arrays are significantly more complex internally than single disk drives. For example, some disk array systems do a significant amount of data relocation and simply maintain a translation from the logical address supplied by the file system to its chosen internal physical location [Wilkes et al., 1995][EMC, 1999]. As as result, accessing adjacent array locations does not always have low latency. Large transfers, however, are an efficient means for accessing disk arrays as well as single disks. In fact, large transfer sizes are even more important for disk arrays because they can have higher latency (RAID controller latency, RAID-5's read-modify write penalty, possible network latency) and higher bandwidth than single disks.

2.3. Trends in Processor Performance

For CPU performance, the Standard Performance Evaluation Corporation (SPEC) endorses a collection of benchmarks suites which are used for standardized evaluation of processor performance [SPEC, 1992][SPEC, 1995]. Vendors may submit benchmark results along with disclosures of the testing procedures to SPEC for publication in SPEC's periodic newsletters.

The SPEC CPU95 Benchmark is designed to test the CPU-intensive portion of commercial and numeric/scientific applications. It contains a set of integer intensive programs referred to as SPECINT and a set of floating point intensive programs referred to as SPECFP. SPEC requires vendors to report both a peak and baseline measurement. The peak result can be achieved using any compiler optimization; while the base result can use at most four compiler flags and has to use the same compiler flags for all programs in the benchmark suite. The baseline measurements should be more reflective of the actual performance improvements seen by applications compiled with common optimization.

Figure 2-2 plots baseline SPECINT 95 benchmark results for every machine reporting results in the SPEC newsletter on the y-axis with the date of the SPEC newsletter in which they were published on the x-axis. The figure also plots the trend line using simple linear regression to fit an exponential to the data. This shows that processor performance has been improving at a rate of approximately 57% per year. This dramatic improvement in processor performance is due to increases in the number of transistors per chip and increases in the speed of individual transistors, both caused by steady improvements in the feature size of integrated circuits. In Section 2.2, we will see that storage technology has been unable to follow this rate of improvement because disk performance is limited by mechanical access delay.

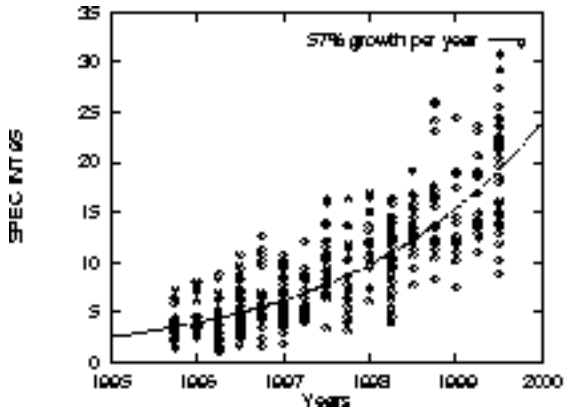

FIGURE 2-2. **SPEC CPU95 Integer Base Results, 1995-1999.** This figure plots the SPECint95 results reported in SPEC's quarterly newsletters from third quarter of 1995 through third quarter of 1999. The points are baseline CINT95 speed measurements. These speed measurements are expressed as the ratio of execution time on the test machine compared to the execution time on a reference machine, a Sun SPARCstation 10/40. The single figure reported is actually the geometric mean of the SPEC ratios for each of the 6 programs in the CINT92 benchmark suite. The 6 programs are written in C and represent the CPU intensive part of system or commercial application programs. The ratios for the 6 individual programs on a single machine can vary widely. Baseline refers to the fact that all the programs are compiled with only conservative compiler optimizations (at most four compiler flags and the same set used for all benchmarks). This figure indicates that processor performance has recently been improving by about 57% per year.

2.4. Trends In Memory Technology

Table 2-6 recalls the average improvement rates and 1998 baseline figures for access latency, bandwidth and cost/capacity for memory and disk as reported in [Patterson, 1998]. In absolute terms memory is much faster than disk and disk is much cheaper than memory. However, latency, bandwidth and cost/capacity are all improving more slowly for memory than for disk. Figure 2-3 illustrates one result of these trends. It shows that the ratio of disk capacity to memory capacity in complete systems has been increasing overtime.

TABLE 2-6. **Relative Performance and Cost/Capacity of Disk and Memory** This data is taken from data presented in Professor David Patterson's keynote address at SIGMOD98, "Hardware Technology Trends and Database Opportunities". Access latency refers to the time for an average seek plus one-half rotation.

	DRAM		**Disk**	
	Improvement Per Year	**1998 Baseline**	**Improvement Per Year**	**1998 Baseline**
Access latency	7%	10 nanoseconds	8%	9 milliseconds
Bandwidth	20%	800 MB/sec	40%	15-21 MB/sec
Cost/Capacity	25%	$1.19/MB	60%	$0.09/MB

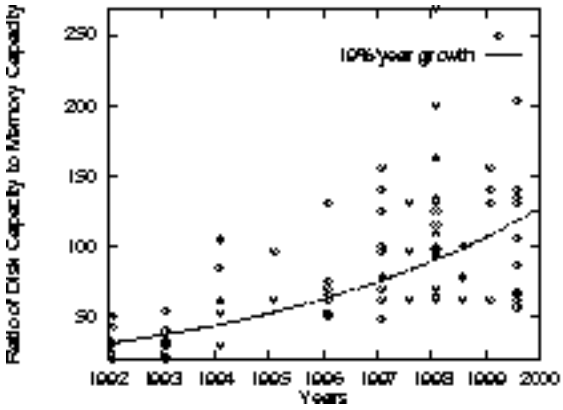

FIGURE 2-3. **Ratio of Disk Capacity to Memory Capacity, 1992-1999** This figure plots the ratio of disk size to memory size for complete systems advertised in Byte or PC Magazine. Byte is used from January 1992 through July 1998. Byte went out of print in July 1998 so the January 1999 and July 1999 data points are from PC Magazine. The systems represented include low-end to high-end desktop and server machined advertised by Gateway, Micron, IBM, etc. I use simple linear regression to fit an exponential to the data and find that total disk capacity relative to total memory capacity is rising by about 19% per year.

DRAM is a volatile storage medium—meaning that it does not retain its contents across power failures. As a result, DRAM can be used as a cache of unmodified data to improve read performance, but modified data is not stably stored until it reaches disk. Non-volatile DRAM (NVRAM), however, can be used to safely buffer disk writes. Such buffering can improve write performance by allowing data to be overwritten or deleted before it is ever written to disk. However, NVRAM is more expensive than volatile DRAM and is not considered as reliable as disk. In

practice, most file server machines have NVRAM, but desktop machines rarely do and even systems with NVRAM limit the amount of time data can be buffered before reaching disk.

2.5. Conclusions

In this section, I have discussed a variety of technology trends and their implications.

1. Disk bandwidth is improving more rapidly than access latency (seek and rotational delay). In order to use the disk efficiently, latency must be avoided through larger transfers and reducing seeks.
2. Increases in disk capacity imply the need for efficient crash recovery mechanisms that do not increase in latency with increases in total storage capacity.
3. Overall system performance is increasingly limited by storage system performance, due to rapidly increasing processor performance and the slower rate of improvement in storage system performance. This indicates the need to optimize for the I/O bottleneck.
4. Memory is improving more slowly than disk in terms of access latency, bandwidth and cost/capacity. As a result, file systems do not have the luxury of relying on memory to hide all the effects of disk latency.

In the next chapter, I discuss how major existing file system designs have wrestled with these issues. In Chapter 4, I discuss variations in workload characteristics and how those variations affect the relative merits of those file system designs.

3 A Brief History of File Systems

File systems provide the interface between the raw storage system hardware and applications. In the last chapter, I discussed hardware trends especially relevant to file system design. In this chapter, I present a brief history of the evolution of file system architectures.

3.1. What is a File System?

File systems provide permanent storage and retrieval of named objects. These objects, called files, are usually untyped byte arrays whose internal structure and interpretation is left to the user or application that created them. Files are typically named within a hierarchical name space in which special files called directories contain regular files and other directories. A directory is referred to as the parent of all files and directories it contains. Figure 3-1 contains a sample directory hierarchy.

Main memory, which is smaller, faster and less reliable than disk, is used as a temporary staging area for file system data. Functionally, the file system makes the movement of data between memory and disk transparent to the end user. However, the vast performance differences between memory and disk make it difficult to truly hide this movement. As a result, most file system performance enhancements revolve around hiding disk latency. Some common optimization techniques, such as caching recently used data in memory, are designed to increase the percentage of data accesses that can be satisfied without a disk access. Other common techniques, such as batch-

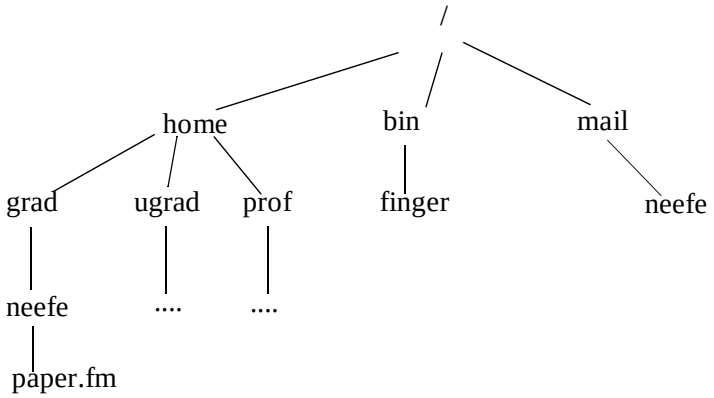

FIGURE 3-1. **Illustration of a Directory Hierarchy** The directory hierarchy begins with a special directory called the root directory, /, which has no parent. Files can be executable, like the program "finger" or text files like /mail/neefe or application specific data files like paper.fm. Names are only meaningful within the context of this directory structure. For example, neefe is both a directory within /home/grad and a file within /mail.

ing disk requests to adjacent blocks, are designed to access the disk more efficiently by minimizing seek and rotational delay.

The file system is responsible for the persistence of data across system failures. Therefore, it must write all modified data to a disk or other persistent media.[1] In addition to files and directories, the file system maintains system information, called meta-data, which is needed to interpret the contents of the disk. For example, it must keep track of which portions of the disk are free for writing new data. Each file is normally divided into pieces (either fixed sized pieces called blocks or variable sized pieces called extents) and the location of each piece must be recorded. In addition to files and directories, these and all other types of meta-data must be stored durably. [2]

Many file system actions actually involve multiple updates to the persistent storage. For example, moving a file from one directory to another involves changing the contents of both directories

1. Magnetic disks retain their data across power failures, but they are not immune to data loss from other sources including mechanical failure and user error. In many environments, for added protection against data loss, periodic backups of file system data are written to an off-line media like magnetic tape or writable CD-ROM. In some cases, the file system controls this backup process directly.

on disk. Both modified directories must be written to disk before the operation is complete. Without careful coordination of these separate updates, a system failure could might leave the file in either directory, or both directories, or neither depending on the implementation of the operation and the time of the failure.

Transactions are a model for grouping updates to persistent storage which have the following properties [Gray, 1981]:

- **Atomicity**. Either all updates that are part of the transaction occur or none of them do.
- **Consistency.** Transactions leave the system in an internally consistent state.
- **Isolation**. The effects of a transaction are not visible to other applications until the transaction completes.
- **Durability**. When a transaction completes and is made visible to the rest of the system, the effects of that transaction are guaranteed to survive system failures.

File systems generally maintain strict transactional semantics for particular sets of updates to system meta-data and to the directory hierarchy. This ensures that the file system structure will remain consistent. For user data, they tend to make looser guarantees. For example, many file systems guarantee that data written will be stably recorded to disk within 30 seconds, but ensure nothing about the order in which updates reach disk nor about the atomicity of multi-part operations. However, this window of vulnerability and weak semantics allow for increased performance as: (i) the operating system can return to the application immediately after buffering the write, (ii) writes

2. The specific structures used to store and organize meta-data vary considerably between file systems. For example, some file systems maintain most of their meta-data in special files which can grow in size like user data files [Rosenblum and Ousterhout, 1992][Hitz et al., 1995]. Other systems maintain fixed size regions of meta-data. [McKusick et al., 1984]. There must always be some meta-data located in a fixed location where the file system can begin reading to recover after a system crash in order to boot strap crash recovery. However some file systems fix the location of all of their meta-data [McKusick et al., 1984], while others allow all but the essential boot-strapping information to be allocated anywhere on disk [Rosenblum and Ousterhout, 1992][Hitz et al., 1995].

may be deleted or overwritten while waiting to go to disk thus completely avoiding the disk transfer and (iii) writes to user data may be written to disk in any order allowing optimization of disk head movement.

3.2. Early File Systems

The original UNIX file system is a typical example of an early file system design. UNIX, like many other systems, manages disk storage with a linked list of free blocks [Ritchie and Thompson, 1974][Thompson, 1978]. When the system needs to allocate a new file block, it simply removes the first block on the free list. Once a block is allocated a position on disk all future modifications (and reads) to that block are directed to the same disk location. For this reason, this type of system is referred to as an update-in-place system.

Initially, the system's free list can be ordered optimally with respect to the actual locations on disk (i.e. blocks next to each other in the list are next to each other on disk). However, over time, the list is randomized as blocks are placed back on the free list when they are no longer needed (e.g. due to file deletions). In general, blocks need not be deleted in the same order they are created. The disk is treated as a random-access heap. As a result, related data becomes increasingly fragmented or scattered across the disk. If a fragmented file is accessed sequentially, seek and rotational delay is required between each block. This approach has poor performance for both read and write traffic because performance scales with mechanical access latency not with disk bandwidth. I address solutions to this problem in later sections.

In addition, an update-in-place system can become corrupted during a system failure. As I discussed in Section 3.1, some updates require multiple disk writes and a failure that occurs between such related updates may leave the system in an inconsistent state. In an update-in-place system, such inconsistencies can be located anywhere on disk because each write is directed to its previ-

ously allocated position. In addition, because each write is placed on top of the previous version of the data, if a failure causes a write to be interrupted, the data could be left in an inconsistent state (partially old and partially new).

Fsck is a crash recovery procedure developed to address some of these problems [Kowalski, 1978]. It is based on an understanding of how operations are implemented and specifically it depends on multi-part operations being performed synchronously and in an specified order. For example, creating a new file requires several separate updates: to the freelist, to the file's parent directory, to a special file descriptor called an i-node and to the file's data blocks. Fsck relies on knowing the exact order on these operations. First, the freelist is changed to allocate space for the new file. Second, an i-node is assigned to the file and is modified to point the allocated blocks. Third, the file's data is written into the allocated blocks. Finally, an entry for the file is added to its parent directory. Each update must be synchronously written to disk before the next one may occur. These small synchronous updates limit write bandwidth because they are unable to hide mechanical access latency.

After a crash, fsck scans the entire disk, locating and repairing operations that did not complete cleanly and atomically. It detects inconsistencies such as disk blocks that belong neither to a file nor to the freelist and files that are not pointed to by any directory entry. Fsck repairs these inconsistencies by returning to the free list any allocated blocks not pointed to by an inode and placing any files not pointed to by a directory into a special "lost and found" directory. While this returns the file system's internal structure to a consistent state, it can lose user data written to disk and may require the user to manually recover a file from the lost and found directory. In addition, because this recovery procedure scans the entire disk, crash recovery time scales with total storage capacity. This approach becomes increasingly inefficient in the presence of rapidly increasing

storage capacities. Finally, it does not solve the problem of partial writes corrupting existing data — even system meta-data.

In the next sections, I describe how file systems have evolved to improve performance during normal operation and to reduce the time required for crash recovery on modern large disks.

3.3. The Fast File System

The Fast File System (FFS), introduced in 1984, makes several important performance improvements to the update-in-place model [McKusick et al., 1984]. First, it increases the block size so that the mechanical access latency for each block is amortized over a larger transfer.[1] Second, it further reduces seek and rotational delay by dividing the disk into sections called cylinder groups and allocating semantically related data to nearby blocks in the same cylinder group. FFS continued to use fsck for crash recovery.

FFS can be classified as a read-optimized file system. Most read-optimized file systems attempt to locate files within the same directory together and to place each file sequentially. Rather than writing to the first block on the free list or to the free space closest to the current disk head position, FFS incurs additional seek and rotational delay for each write in order to place the data in a specifically chosen location on disk. The lower write performance is intended as an investment in higher read performance; thus FFS can be classified as a read-optimized file system.

When a block that has not yet been allocated to a disk location is written, FFS first chooses the cylinder group in which it would like the block to be placed. If this is the first block in the file to be allocated, FFS places it in the first free block in the same cylinder group as its parent directory. As

1. FFS combined this larger block size with the ability to allocate *fragments* of blocks to small files or to the last block in a larger file. This allowed larger transfer sizes without a corresponding increase in internal fragmentation.

additional blocks in the file are allocated, it places them in the same cylinder in a free block that is rotationally close to the previous block in the file.[1]

If the desired cylinder is full, FFS looks for free space in the other cylinders of the same cylinder group. If the entire cylinder group is full, another group is chosen by way of a quadratic hash on the original cylinder group number. Finally, if this hash finds another full cylinder group, FFS resorts to an exhaustive search of all the cylinder groups.

This search strategy has some important performance ramifications as pointed out in [Smith and Seltzer, 1994]. Cylinder groups tend to fill from beginning to end - resulting in a higher density of free space at the end of the group. Files tend to become fragmented as they fill gaps left by earlier deletions — their early blocks are located at the beginning of the group and their later blocks at the end. Small files tend to become especially spread out since they often do not have enough blocks to reach the contiguous runs of free blocks that are still available at the end of the group. Studies of actual file layout in FFS have shown a significant degree of fragmentation, especially at higher disk utilizations [Smith and Seltzer, 1994]. In addition, since blocks do not move once allocated, files that are created during periods of high disk utilization stay fragmented even if the disk utilization is later reduced.

Since the original FFS design, several techniques have been developed to reduce fragmentation. One version of BSD uses a reallocation algorithm that attempts to relocate[2] logically sequential blocks within the same file together on disk [BSD4.4-Lite Source, 1994]. This reduces the

1. There is an exception to this rule for large files. When a file exceeds 48 KB and at every megabyte there after, FFS forces the selection of a new cylinder group. This is done to prevent a large file from occupying an entire cylinder group forcing all other files in that same directory to be scattered to other cylinder groups.
2. When a new block is added to a file, this algorithm will attempt to relocate the previous blocks (up to a maximum chunk size) to an area of free space large enough to accommodate both the old blocks and the new one [Smith, 1999].

fragmentation of individual files and therefore improves the performance of sequential read and write access to those files [Smith and Seltzer, 1996]. Ganger and Kaashoek reduce the fragmentation of whole directories by grouping small files that belong to the same parent directory together on disk. This can be beneficial especially for common utilities, like grep, that tend to operate on many files in the same directory [Ganger and Kaashoek, 1997]. Both of these techniques are consistent with the stated goal of FFS allocation policies which is to group the semantic units of files and directories together on disk. However, we note that the logical namespace of the directory hierarchy is not necessarily matched by the read patterns in a real workload. For example, the object files from several directories may be read together to produce an executable and the executable itself may be read out of order to avoid error handling code.

3.4. Write-ahead Logging File Systems

To address the crash recovery problems of update-in-place systems, file system designers borrowed a technique from databases called write-ahead logging. Write-ahead logging file systems [Hagmann, 1987] [Chutani et al., 1992] [Berkeley Trace Repository, 1999] [Custer, 1994] [Veritas, 1995] [Sweeney et al., 1996] record updates to a log. These updates are eventually propagated into a traditional file system organization much like that of FFS, but they become durable as soon as the transaction of which they are a part is completely written or committed to the log. Once propagated, the log entries can be re-used. Typically, systems periodically force all committed data in the log to prevent the log from growing arbitrarily long. This is called checkpointing.

In the event of a system crash, the system reads the log and completes any unfinished processing for transactions committed in the log. This combination of checkpointing and roll-forward processing is like that found in database systems [Mohan et al., 1992]. With write-ahead logging, the time to recover from a crash is proportional to the size of the log and the duration between check-

points rather than with the size of the total storage. Thus, write-ahead logging, unlike systems using fsck, remains efficient even as storage systems increase in size at a rate of 60% per year (Section 2.2). In addition, replaying the log after a crash can repair a block corrupted by a partial write.

Most write-ahead logging systems place only meta-data operations in the log. Writes of ordinary file data are still sent asynchronously to the traditional update-in-place storage. Thus, write-ahead logging avoids the predefined sequences of small synchronous meta-data updates, but it does not extend the consistency benefits to user data.

Using a log has another somewhat surprising effect— despite having to write everything twice (first to the log and then to the final storage location), write-ahead logging can actually improve write performance. The writes into the log are more efficient because they can be delayed and batched without loss of atomicity. By varying the amount of time between periodic log writes, the risk of losing the most recent updates can be weighed against the performance benefits of batching. Many database systems invest in a separate disk for the log so that seek delay can be even further reduced by ensuring that the disk head is left in the track to which the next log write will be directed. Copying updates into the traditional file system organization can also be made more efficient. Once the update is safely recorded in the log, these second propagation writes can be safely delayed and rearranged into an order that minimizes seek and rotational delay [Seltzer et al., 1990] [Jacobson and Wilkes, 1991]. The read performance of write-ahead logging systems is similar to that of update-in-place systems like FFS because the final layout strategy is the same.

3.5. The Log-structured File System

The log-structured file system (LFS) took the logging approach one step further by avoiding the second propagation writes and treating the log as the entire file system [Rosenblum, 1992]

[Rosenblum, 1992]. In this way, both file system structures and user data can benefit from the recovery properties of the log. Figure 3-2 illustrates the LFS architecture.

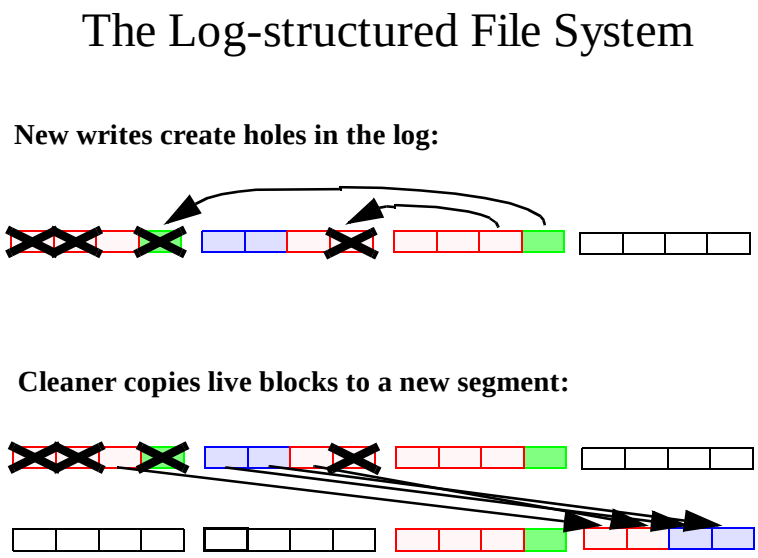

FIGURE 3-2. **Illustration of the Log-structured File System Architecture.**

In an LFS, all writes, regardless of whether they are the first write of a new piece of data or an overwrite of existing data, are batched to the end of the log. Holes are produced when a block of data is deleted or overwritten. On an overwrite, the new contents are placed at the tail of the log — instead of being updated in place as with earlier file systems. The previous contents of such a block are still located in their original location, but they are no longer a part of the active file system.

In the common case, writes are batched to the log in large units called *segments*. The disk is divided into segment-sized chunks; a newly written segment can be placed in any unused position regardless of the location of the previously written segments. In the original LFS design, the segment size was chosen to be large enough that the transfer of a single segment was sufficient to

make mechanical access delay small compared to the transfer time. (I will discuss other factors that influence the choice of segment size in Chapter 5.)

Writing segments amortizes seek and rotational delay and allows write performance to scale with disk bandwidth rather than with mechanical access latency. However, LFS is not always able to write in units of full segments. If an application requests that data be sent synchronously to disk (i.e. the sync or fsync system calls), LFS must send whatever data has accumulated in memory directly to disk even if only a partial segment must be written.

Although LFS treats the disk logically as an append-only log, it will, of course, over time, run out of free space for new log writes. Therefore, the LFS architecture includes a garbage collection process called the *cleaner*. It is the cleaner's job to compact useful data. The cleaner reads partially empty segments (i.e. segments containing both holes and live data which has not been overwritten or deleted) and rewrites only the live data to the tail of the log. Once this has been done, a segment can be declared free and is then available for future segment writes.[1]

LFS has many advantages. LFS provides batching of writes, efficient crash recovery and even extends the benefits of logging to user data as well as file system meta-data. A principal concern with the LFS architecture has been the cleaning overhead. Studies have shown that in most environments there is sufficient idle time to accomplish cleaning in the background [Blackwell et al., 1995][Gibson, 1992][Gribble et al., 1998]. However, when insufficient idle time exists, cleaning overhead can delay both reads and writes. These delays are particularly significant in the case of random updates to a full disk. Seltzer et al. found that cleaning overhead led to dramatically lower performance for a transaction processing workload at high disk utilizations [Seltzer et al., 1993]

1. The fact that new versions of data blocks do not immediately replace the old means that there is the possibility of easily integrating an archival system into this architecture. The cleaner could be modified to rewrite live blocks to the end of the log and dead blocks into the archival system.

[Seltzer and Smith, 1995]. Despite the numerous benefits of the LFS architecture, its dramatically lower performance for some workloads led many to question the wisdom of adopting LFS.

In addition, there is the question of read performance. The designers of the log-structured file system argued that LFS could still offer excellent read performance [Rosenblum and Ousterhout, 1992]. They hypothesized that since data is initially written to disk in large contiguous chunks, the layout would reflect the temporal locality of the current update stream. If read patterns followed write patterns, this layout would also perform well for reads.

However, the read performance of write-optimized systems has been largely unexamined. Studies of LFS read performance have indicated little beyond its ability to efficiently read files sequentially that were also written sequentially [Rosenblum, 1992] [Seltzer et al., 1993]. LFS read performance has not been evaluated on a wide range of workloads and there have been no detailed studies to quantify the impact of garbage collection on read performance.

Garbage collection disrupts the initial temporal locality by grouping pieces of unrelated segments together. The cleaner groups of partially empty segments and rewrites the live data from them into new full segments. Thus, depending on which segments are chosen to be cleaned together, cleaning can disturb the write locality inherent in the original log by grouping together data from potentially unrelated segments.

In addition, read patterns do not always follow write patterns. Related data that is incrementally modified may become dispersed over the disk. For example, in a software development workload, actively modified source files migrate to the tail of the log, but during each compilation they are read at the same time as unmodified source files located earlier in the log. Multiplexing the log amongst multiple clients, users, and applications can also diminish the efficacy of temporal locality by grouping data from many streams together.

3.6. Write-Anywhere File Systems

Unlike earlier file systems, LFS allows disk layout to be determined by what is convenient for writes. It does not directly attempt to group related data into cylinder groups, but rather allows data to be placed in any segment on disk. Current trends in storage systems are towards write-anywhere designs that take the write optimization of LFS one step further [Chao et al., 1992] [English and Stepanov, 1992] [Hitz et al., 1995] [Wang et al., 1999].

These write-anywhere systems use intimate knowledge of the storage system, like current disk head position or the number of free blocks in each RAID stripe, to write new data to the most readily accessible free space. Like LFS, write-anywhere systems benefit from large extents of free space. Unlike LFS, they do not require garbage collection to generate extents of free space for new writes. They can write into any available free space regardless of its size or location.

Write-anywhere systems vary in how they handle crash recovery. WAFL has one of the most interesting approaches [Hitz et al., 1995]. It maintains read-only copies of the entire file system called snapshots. Each snapshot is a completely self-consistent image of the file system to which the system could revert in the event of a crash. WAFL relies on NVRAM to protect any data written between snapshots.[1] The space required to maintain snapshots scales with the amount of update activity because snapshots use a copy-on-write technique in which only the data that changes between snapshots must be duplicated. When data is overwritten, it is placed in a free location and the previous contents continue to be accessible through the previous snapshot.

A new snapshot can be created by copying only the root node of the file system tree to a well-known location on disk. All other blocks in the snapshot can be reached from there. Between snap-

1. WAFL is specific to NFS file server appliances produced by Network Appliance Corporation. In this environment, the presence of NVRAM can be guaranteed.

shots, the system copies those nodes in the tree that are modified by each update operation. Previous snapshots continue to point to the old copies and snapshots created in the future will point to the new copy.

In the Network Appliance implementation, snapshots are taken frequently (e.g. every 30 seconds) and up to 20 different snapshots may be available simultaneously. Disk blocks do not become available for reuse until all the snapshots that refer to them have been deleted. This can create problems when disk space is at a premium. For example, if a large file is deleted, the space it occupies will not become available until all the snapshots that reference it have been deleted. However, snapshots also allow users to recover accidentally deleted files or compare current files to previous version.

Loge and Mime accomplish write-anywhere optimizations at the disk driver level without the knowledge of the file system [English and Stepanov, 1992] [Chao et al., 1992]. The main data structure in these systems maps the block addresses used by the file system to the block's actual storage location. Periodically, Loge and Mime write a checkpoint containing this map and a list of the blocks that are currently free. A header is written with each block identifying its contents and write time. In the event of a crash, Mime begins with the most recent checkpoint and examines every free block indicated by the checkpoint to see if it had been subsequently written[1]. Thus, the recovery time is proportional to the amount of free space reserved on disk rather than the time between checkpoints.

One important parameter for a write-anywhere design is the algorithm used to search for a convenient free space for writing. English and Stepanov describe a problem with simply choosing the closest free space in either direction from the disk head [English and Stepanov, 1992]. This

1. Loge actually scanned the entire disk to recover the indirection map.

minimizes seek distance in the short term, but can trap the disk head in a small area with a few fragmented, nearby free spaces while never reaching larger contiguous regions of free space that are further away. To avoid this problem, Wang suggests searching the disk for free space in one direction completely, switching directions when the edge of the disk is reached [Wang et al., 1999]. The WAFL allocation policies are similar in flavor, but focus on filling in multiple holes in the same RAID stripe rather than writing to free spaces near the disk head on a single disk [Hitz et al., 1995].

While providing high write performance, the read performance of these systems has been largely unexamined. Data that is written together may land in adjacent free spaces when they are available, but the layout does not reflect the temporal locality of the write stream as directly as in LFS. At higher disk utilizations, it becomes more difficult to locate adjacent free spaces and even data written together may become scattered over the disk.

3.7. The Effect of Caching and Prefetching

Caching and prefetching are two common techniques used to hide storage system latency for reads. As such, they can change the character of read traffic and alter the balance between read and write traffic.

Caching improves read performance by retaining recently read data in memory resulting in quicker access if it is read again before being replaced. Caching is limited by the amount of memory devoted to the buffer cache relative to the amount of data stored in the file system and the current workload's working set. LRU ("Least Recently Used") is the most commonly used cache replacement strategy, although there are others like FIFO ("First In, First Out") which benefit some specific workloads. Other systems improve cache efficiency by allowing applications to give hints about whether specific blocks read should be retained in the cache [Cao et al., 1995].

Caches can significantly reduce the amount of read traffic to disk. Roselli captured file system traces in an academic environment and found that reads vastly outnumbered writes at the system call level, but after the cache, disk traffic contained twice as many writes as reads [Roselli, 1998]. However, that study as well as others have documented that larger caches offer diminishing returns [Baker et al., 1991][Roselli, 1998]. There is very little that can be done to avoid cache misses for data that has not been accessed in some time. Once the current working set has been accommodated, there is less locality in the remaining access stream. In addition, caching offers little benefit to streaming read access patterns.

The designers of write-optimized systems have at times argued that large caches could absorb a sufficiently large percentage of the read traffic that disk reads would become a rare event for which there would be no need to optimize [Rosenblum and Ousterhout, 1992]. This argument was especially compelling at a time when the price and performance of memory were improving more rapidly than for disk. However, as I discussed in Chapter 2, this is no longer the case. Coupled with the diminishing returns from larger caches, caches do not offer a sufficient replacement for a disk layout strategy that effectively supports read accesses.

Another approach for improving read performance is prefetching. Prefetching anticipates future accesses and tries to fetch data from disk before it is requested. Sequential look-ahead is one of the simplest and most common prefetching strategies. When a block is read, the next several blocks that follow the block sequentially in the same file are also fetched. Unfortunately, this limits the maximum prefetch benefit to the size of the file. Griffeon and Appleton showed how to prefetch multiple files by dynamically building a graph in which files are nodes and edges represent subsequent accesses. They prefetch files that follow the current file with a probability greater than a given threshold [Griffioen and Appleton, 1994]. Other systems provide a mechanism for applications to give hints about which blocks to prefetch (those they are likely to use in the future)

[Patterson, 1995]. Chang and Gibson include an additional thread in each application to execute speculatively ahead of the main thread in order to generate prefetching requests [Chang and Gibson, 1999].

In order to be effective, prefetching must be started far enough in advance of the time data needed by the application for the data to be read from disk. This is not always possible; worse, the amount of advance warning needed, relative to application execution time, increases as processors become faster. Prefetching is also limited by the amount of additional disk bandwidth that is available after the application requests have been satisfied. Prefetching can be successful at hiding latency from the application, but it does not decrease the demands on the disk and can in fact significantly increase them. (No additional bandwidth would be required if the prefetching algorithm was always able to correctly predict application demands. However, mistaken prefetching requests take both disk bandwidth and space in the cache away from legitimate requests.) Without careful layout, the disk may be too busy to prefetch data before it is needed, especially since file requests tend to be bursty [Baker et al., 1991] [Gribble et al., 1998].

Caching and prefetching complement, but do not replace, effective disk layout. Caching relieves the disk of repeated requests for the same items and effective disk layout can capture the locality of data that is too large to be captured by a fixed sized cache. Effective disk layout improves the efficacy of prefetching by reducing the lag time between the prefetching request and the return of the data from disk.

3.8. Conclusions

File system goals include providing reliable access to data across system failures, high performance write access and high performance read access. Unfortunately, these can be conflicting goals. Minimizing data loss requires frequently writing data to disk which can be costly due to the

large performance gap between memory and disk. Performance can be improved with data layout that minimizes seek and rotational delay. For write access, this can be accomplished by batching many writes together or by writing to readily accessible free space. However, for read access, this requires grouping related data together on disk. Thus, write optimizations that write to the most convenient location can work at cross purposes to the read optimizations that attempt to group related data. Caching and prefetching are important techniques for improving read performance, which complement, but do not replace disk layout that effectively clusters related data for efficient read access.

Over time, various file system architectures have balanced these competing demands in different ways. So far, no single approach has been shown to be good in all dimensions. FFS and write-ahead logging systems incur higher write cost to arrange data semantically to improve read performance, even though read patterns do not always follow this organization. LFS and write-anywhere systems achieve high write performance, but can scatter related data lowering read performance. In the next chapter, I will further discuss how workload variations affect the trade-off between these file system designs and why adaptive methods offer a natural approach to capturing the strengths of these various systems in one single file system.

4 Workload Characteristics Affecting File System Performance

In Chapter 2, I discussed major technology trends affecting file system design. In Chapter 3, I discussed a variety of existing file system architectures. In this chapter, I discuss how the comparison between them is highly dependent on workload characteristics. I describe major characteristics by which file system workloads can be categorized. I discuss how variations in these characteristics can affect file system performance and the impact they can have on the trade-off between the major file system paradigms. I conclude that each file system paradigm described in Chapter 3 has regions of good performance and other regions of poor performance depending on the workload and the hardware available to the system. Adaptive methods offer the potential to develop a file system that combines the best features of each system.

In Section 4.1, I describe several specific workloads and how they differ. In Section 4.2, I discuss specific axes along which all workloads vary and how those variations affect the performance of various file system designs.

4.1. Characteristics of Several Important Workloads

Table 4-1 gives a high-level characterization of five important workloads that ideally should be well supported by any file system design: multimedia, software development, transaction processing, decision support and scientific simulation. Of course, there can be great variation within each category. For example, one scientific simulation may read huge data input files and produce small output files while another may read small input files and produce large output files. How-

ever, Table 4-1 illustrates that even at a high level various applications produce very different access patterns.

Multimedia workloads are characterized by steady, sequential access to large files. Video and audio playback require constant data rates to avoid interrupting the presentation to the user. Files are often large and read sequentially in their entirety. Files are often written once and then read many times.

Software development is characterized by the production of many intermediate source files that are frequently updated and many repeatedly overwritten. Source modifications are interspersed with compilations; each compile produces object files which are overwritten whenever the code is modified. Often, the directory groupings of files are helpful in identifying related files. Temporal locality is less helpful as frequently modified files will move away from related, but unchanging files.

Transaction processing is characterized by frequent, random accesses to large files. There is typically some locality of access to the high levels of indices for the data. Transaction processing is often a mission critical application. Constant availability is important as well as the need for frequent data commits.

Decision support workloads are characterized by routine data insertions and read access to a large portion of accumulated data. In a typical environment, business records, such as the most recent updates in a transaction processing system, are periodically inserted into an historical repository. Ad-hoc queries are run on vast repositories of this historical data to aid in the making of business decisions. Read access patterns can be unpredictable as decision makers look for new ways to view their data to yield insights into successful business practices. If the same types of questions are frequently repeated, indices may be created to efficiently support the resulting access

TABLE 4-1. Common Characteristics of Various Workloads

Workload	Write Order	Write Locality	Read Order	Read Locality	Read and Write Interaction	Request Rate	Size	Data Lifetime
Multimedia	whole file sequential	no	whole file sequential	no streaming	write once read many	sustained	big files	long
Software Development	sequential for executables; local updates in code	yes	whole file sequential; groups of related files	yes	interleaved	bursty	bimodal many small files; few large files	bimodal many short; a few long
Transaction Processing	random	no	random	no for data; yes for indices	interleaved	sustained	varies	varies
Decision Support	update/append	no	random, patterns for common queries	no data; some for indices	write once, read many	varies	huge	long
Scientific Simulation	sequential output	no	varies; often sequential	varies	write once, read once	varies	big files	many long; short for repeated experiments

patterns. The historical nature of the data repository makes deletions rare and therefore the amount of data collected can grow to be extremely large.

Scientific simulations, such as climate modeling, are similar to decision support in that they process huge repositories of collected data. Many scientific simulations also produce large output files detailing the processing performed and the results computed. Both input and output is frequently in sequential streams—predictable but with little locality. Repeated, unsuccessful experiments overwrite existing results. Input data and the results of successful experiments are likely to be saved indefinitely.

4.2. Workload Characteristics

Section 4.1 described the characteristics of several specific workloads at a high-level. In this section, I further characterize how workloads differ in ways relevant to file system design. I discuss workload characteristics along the following axes: write pattern, read pattern, read and write mixture, rate of data access, data size and distribution, disk utilization, data lifetime, frequency of data commit and availability requirements.

4.2.1. Write patterns

4.2.1.1. Order of Write Accesses

The sequence of writes to a file system may or may not follow the semantic units of files and directories. For example, files may tend to be written sequentially in their entirety, sequentially to only a portion of the file or nonsequentially. Subsequent writes may be to the same file or to another file in the same directory. If the sequence of writes does not follow the directory hierarchy, then the writes may be random or there may be other patterns that are predictable even though they do not follow the semantic units of files or directories.

Write-optimized file systems are not sensitive to write order because regardless of the patterns they direct all new writes to a convenient location either through batching in LFS or by choosing a free location near the disk head in write-anywhere systems. However, read-optimized file systems, like FFS, are sensitive to write order because they choose a disk location for a block based on the directory and file to which it belongs and on its offset into the file. Systems like this are willing to incur additional seek and rotational delay for write patterns that do not match the desired organization in order to sort the data based on its semantic properties.

4.2.1.2. Temporal Locality of Write Accesses

A range of update patterns are possible: there may be a high degree of locality to the set of data being actively updated or updates may be directed with equal probability to any block on disk.

Read-optimized systems are more sensitive to write pattern than to write locality. Even a write pattern with a high degree of locality will cause additional seek and rotational delay if it does not match the semantic organization of data on disk. Read-optimized systems are sensitive to the write locality to the degree to which it contributes to the fragmentation of free space as disk blocks are allocated and freed over time.

Write-optimized systems, which are insensitive to write order, are highly sensitive to the degree of locality in the write stream. In the case of LFS, increasing locality lowers garbage collection overhead by decreasing the number of segments that accumulate free space. In the case of write-anywhere systems, increasing locality causes there to be more free space available in one area of the disk. This makes it less expensive to find free space when the disk head is in that area.

4.2.2. Read Patterns

4.2.2.1. Order of Read Accesses

Like the write pattern, the sequence of reads may or may not follow the semantic units of files and directories. Even if the sequence of reads does not follow the directory hierarchy, reads may follow other patterns predictable patterns. Read patterns may or may not match write patterns.

All systems are sensitive to read order. Read performance is higher when read patterns match the system's disk layout policy. Read-optimized systems will perform well when read patterns follow the units of files and directories. Write-optimized systems will perform well when read patterns follow write patterns. Thus, no single system is good for all read patterns.

4.2.2.2. Locality of Read Accesses

There may be a high degree of locality to the set of data being actively read or reads may be directed with equal probability to any block on disk. A cache is more effective for read traffic with a high degree of locality.

4.2.3. Mixture of Reads and Writes

4.2.3.1. Ratio of Reads and Writes

The ratio of reads to writes, especially disk reads and disk writes, determines whether it makes sense to optimize one type of access at the expense of another. Where reads are more common than writes and read patterns are statically predictable, it is appropriate to incur additional seek and rotationally delay to write data near the data with which it will be read. Where writes are more common than reads, it is appropriate to minimize write cost by placing data into convenient locations which minimizes write cost. The mixture of reads to writes can also vary over different subsets of data within the same file system.

4.2.3.2. Interleaving of Reads and Writes

If a given workload consists of well defined phases that are either read or write dominated then it is easier to reason about how each phase affects the other. As reads and writes are interleaved at a finer granularity, they can affect each other in interesting ways. For example, writes in log-based systems are most efficient when the disk head is left sitting at the tail of the log ready for the next batch of writes. Read traffic can move the head resulting in additional seek and rotational delay to locate free space for the next write. Many write-ahead logging systems deal with this situation by dedicating a separate disk exclusively to the log (obviously this has higher cost). LFS, although it would derive a similar benefit from a separate disk, deals with this by batching many writes together in order to amortize seek and rotational delay. Interleaving of reads and writes also has an impact on read-optimized systems; these systems have better performance when the stream of both reads and writes displays significant semantic locality (i.e. data from the same files and directories is being read and updated at the same time).

4.2.4. Rate of Data Access

File system workloads that alternate between idle periods and periods of intense activity have been well documented [Ruemmler and Wilkes, 1993][Gribble et al., 1998]. In other environments, the rate of data access may be more steady (e.g. multimedia workloads, transaction processing workloads). Riedel and Gibson document user dissatisfaction with lightly loaded servers because of poor burst behavior [Riedel and Gibson, 1996]. Therefore, if frequent idle time exists, it is beneficial to perform background tasks that prepare the systems to provide peak performance during periods of heavy load. Systems can be designed to take advantage of bursty traffic by delaying tasks such as garbage collection and data reorganization to idle periods [Rosenblum and Ousterhout, 1992][Blackwell et al., 1995] [Golding et al., 1995]. When such activities are accomplished in idle time, users perceive no performance degradation as a result. However, for other workloads

that demand long-term sustained performance systems that do not defer work to idle time may provide more consistent service.

4.2.5. Data Size and Distribution

Read-optimized systems are particularly sensitive to how data is distributed into semantic units (i.e. the distribution of file sizes and directory sizes). In general, semantic clustering is less effective when the semantic units are either too small or too big for effective clustering. For example, if all the data in the file system is in a single directory, grouping data by directory is ineffectual. Write-optimized systems are less sensitive to these distributions because they make no attempt to actively cluster semantic units.

4.2.6. Disk Utilization

All systems are sensitive to the overall data size in relationship to the storage capacity of the system. In LFS, cleaning performance degrades dramatically as the disk approaches full utilization. The cleaner must be run more frequently to collect more scattered log holes; increased disk utilization decreases the chances that the cleaning can be deferred until the next idle period. Similarly, at higher disk utilizations, write-anywhere systems have lower write performance because they must search farther to find free space and lower read performance because data written together may become more scattered. The performance of FFS also degrades as the disk fills, although somewhat more gracefully. With fewer free slots, FFS is less likely to be able to allocate a new block near other related blocks in the same file or directory. Furthermore, data allocated in this way will remain fragmented even if the disk utilization later drops.

4.2.7. Data Lifetime

All systems benefit from data that is overwritten or deleted quickly enough that old versions need not be sent to disk. In many systems, the majority of data created is overwritten or deleted within a few minutes, but the majority of data alive at any one point may be months old [Baker et al., 1991][Ruemmler and Wilkes, 1993][Roselli, 1998].

Once data has reached disk, frequent, non-localized, deletes cause fragmentation of existing allocation units. Free space produced by deletes are available for new allocations in FFS and write-anywhere systems. LFS however must first compact free space with garbage collection.

In addition, data that lives longer may be a better candidate for data reorganization. In many environments, data that has lived for a long time becomes less likely to be deleted and without reorganization, it will show the effects of poor layout every time it is accessed. Also, data that lives longer has more opportunities to be reorganized and more information can be collected about the way it is accessed in order to reorganize it effectively.

4.2.8. Frequency of Data Commit

All systems are sensitive to the frequency with which sync operations force data to be committed to stable storage. This frequency determines how long data can be buffered in memory before it is written. Increasing this frequency decreases the amount of data that is never written to disk because it is overwritten or deleted in the write buffer. Write-optimized systems are especially sensitive to this frequency because they rely on batching of writes to decrease write cost.

4.2.9. Availability Requirements

Workloads vary in the amount of system downtime they can tolerate. As I discussed in Chapter 3, update-in-place systems such as FFS are susceptible to longer downtimes after a system

crash due to the need to scan the entire disk for inconsistencies. Since the time required to scan the disk is proportional to the disk's capacity, increasing disk capacities have significantly increased the recovery time and thus decreased the effective availability of FFS systems. Both LFS and write-ahead logging systems only need to process the most recently written data on disk and so the time for crash recovery is dependent on the time since the last checkpoint and not on total file system capacity.

4.3. Conclusions

In the presence of rapidly changing technology and diverse workloads, it is difficult to conclude that one file system paradigm is better than another. The comparison depends on too many factors like the ratio of read traffic to write traffic, disk characteristics, the amount of idle time in the workload, the presence of special hardware support like NVRAM, etc. Each of the file system paradigms has regions of good performance and other regions of poor performance. For example, LFS performs well relative to traditional update-in-place or write-ahead logging systems when there are frequent small disk writes, where disk writes dominate reads, where disk activity is bursty, where there is ample free disk space, and with RAIDs. However, traditional file systems perform well relative to LFS when reads are statically predictable, where reads dominate writes and where sustained disk performance is important.

As a result, the best paradigm for building file systems remains a topic of active debate [McVoy and Kleiman, 1991][Rosenblum and Ousterhout, 1992][Seltzer et al., 1993][Seltzer et al., 1995][Ousterhout, 1995a][Ousterhout, 1995b][Seltzer and Smith, 1995][Ousterhout,1995c][Ganger and Kaashoek, 1997][Sweeney et al., 1996]. Adaptive methods offer the potential to develop a file system that combines the best features of each system. Adaptive method are a natural approach to offering consistently high performance in the face of varied workloads

and rapidly changing technology. In the rest of this dissertation, I demonstrate simple adaptive algorithms that can significantly improve the way file systems balance the varied and often competing demands of their environment. I will focus on adaptive modification to the log-structured file system as a way of building a system that is good for both read and write traffic.

5 Providing Robust File System Write Performance In Log-Structured File Systems

In the first four chapters, I have motivated the need for adaptive file system services to provide high performance in the presence of changing technology and varied workloads. In this chapter, I focus on write performance and present adaptive extensions to the log-structured file system (LFS), a file system whose performance is known to be highly dependent on workload. I show how its performance is also highly dependent on variations in disk performance.

5.1. Motivation

Recall that in an LFS, all writes, regardless of whether they are the first write of a new piece of data or an overwrite of existing data, are batched to the end of the log in units called *segments*. The disk is divided into segment size chunks and each new batch of writes can be placed in any free disk segment regardless of the location of last segment written.

Logically, LFS treats the disk as an append-only log. However, over time, it runs out of free segments for new log writes. The LFS architecture includes a garbage collection process called the *cleaner*. Free blocks in previously written segments or "holes" are produced when data is deleted or rewritten and the new data placed at the tail of the log. It is the cleaner's job to re-collect the free space that has accumulated in these holes. The cleaner reads partially empty segments and rewrites just their live data to the tail of the log. Once this has been done, a segment can be declared free and it is then available for future segment writes.

The log-structured file system has many desirable characteristics relative to other existing file system designs. LFS has location independent writes — data can be placed anywhere on disk regardless of the location of other blocks in the same file or directory and even regardless of its own previous location. This flexibility offers greater potential for high write performance than earlier update-in-place systems because it is well suited to scale with trends in storage technology that give preference to large data transfers. The log provides transactional semantics and rapid recovery with the minimal number of writes during normal operation. Garbage collection costs can often be completely hidden in idle time [Blackwell et al., 1995][Golding et al., 1995]. LFS is able to provide high performance during bursts of high load by deferring until idle time work that many file systems force into the critical path[Riedel and Gibson, 1996].

However, LFS had dramatically lower performance for some workloads. In particular, [Seltzer et al., 1993] demonstrates that a random update workload, like the TPC-B transaction processing benchmark, results in expensive garbage collection which renders the system virtually unusable. Regardless of its many benefits, the dramatic worst case performance of LFS has led many to conclude that it was an unacceptable choice for real systems. In this chapter, I will show that with the addition of some relatively simple adaptive methods the worst case performance of LFS can be avoided while retaining its benefits in the common case by (i) choosing an LFS segment size to balance transfer efficiency against cleaning efficiency, (ii) adapting the garbage collection mechanism to changes in disk utilization and workload to avoid poor performance even for random updates at high disk utilization, while preserving the benefits of the original LFS at lower disk utilizations and for workloads with more locality and (iii) reducing cleaning costs by taking advantage of cached data when cleaning. Together, these three optimizations reduce garbage collection costs by up to a factor of six and improve overall write performance by up to a factor of four, showing how adaptive methods can be used to provide consistently high write performance.

5.2. Methodology

To evaluate my adaptive methods, I used a combination of synthetic and measured workloads.

5.2.1. Traces

To explore both the extreme and more common workload characteristics, I use both measured traces of real system behavior and synthetically generated traces to evaluate my modifications to LFS. This is necessary in order to demonstrate that the proposed algorithms can indeed deliver robust performance in extreme conditions and to verify that the algorithms are helping, or at least not harming, average case performance. In this section, I describe the specific workloads I use and discuss how these workloads affect some important experimental parameters.

5.2.1.1. Synthetic Random Update Workload

I use a synthetic random update workload with frequent data commit to examine the worst case performance for LFS. To initialize the disk, I first write enough blocks sequentially to fill the disk to the desired utilization. I then make ten times as many random updates as blocks initially written. I make frequent requests for data commit with a sync call after every fourth write. All writes issue from a single client. This workload approximates a TPC-C transaction processing workload [TPC-C, 1990], but does not include index accesses.

This workload, with its random updates and frequent sync requests, represents an extreme stress test for LFS. With random updates, each segment written produces holes which are scattered throughout many previous segments. To recollect the free space, the cleaner must read many segments and rewrite their live blocks. For comparison, consider the opposite extreme, if the newly written segment contains the same set of data blocks as the previous segment, then the previous segment is emptied completely of its live data; the cleaner only needs to declare it clean in order to generate space for a new log write.

The frequent sync requests also stress LFS write performance. The more often applications request that data be committed to disk, the more difficult it is to accumulate enough data for an entire segment. Partial segment writes reduce the system's ability to amortize access latency over large transfers.

Disk utilization also affects LFS write performance because it determines how long the system can wait for segments to empty. The emptier segments are, the less live data the cleaner must recopy to generate free space. Even for the random update workload, segments will eventually empty on their own if there is sufficient free space on disk for the new segment writes to proceed.

In several experiments in this chapter, I examine the effect of variations in disk utilization on LFS write performance. For synthetic workloads, disk utilization can be kept constant throughout the trace. For example, in this random update workload, I initialize the disk with a sequential write of all data that will exist during the trace. As I will discuss next in Section 5.2.1.2, controlling disk utilization for measured traces can be less straight-forward.

5.2.1.2. Auspex Trace

I also examine one measured trace to represent more common levels of write locality and data commit frequency. (Of course, this trace reflects only one common workload and not "the common case".) I use the Berkeley Auspex Trace [Dahlin et al., 1994a]. This trace follows the NFS activity of 236 clients serviced by an Auspex file server over the period of 1 week during late 1993. It was gathered by snooping Ethernet packets on four subnets. The clients are the desktop workstations of the University of California at Berkeley Computer Science Division. There are approximately 4 million reads and 1 million writes, each to 8 KB blocks, in the trace. In addition, there are approximately 40,000 file deletes. Data is committed with a sync request every 30 sec-

onds. Because these traces are of NFS activity, the accesses that hit in the local cache are not reflected in the traces. Accordingly, the size of the client caches is set to zero in the simulations.

Like most file system traces, the Auspex trace does not include a snapshot of the state of the disk before the trace began (i.e. which files already existed, their size, their last modification time, etc.). Therefore, many of the writes contained in the trace would have actually been overwrites to data initially written before the trace began, rather than new writes as they appear to be. Despite delete operations and frequent overwrites, the amount of live data climbs steadily throughout the trace from an initially empty disk to a peak of 1.2 GB at the end of the trace. Over the lifetime of a file system, disk utilization does naturally increase as more and more data is created. However, in the real file system, this variation would have represented a much smaller percentage of the total data stored.

To address this concern, I examine the trace and infer as much as I can about the data that existed on disk before the trace began. If the trace contains the read or delete of a block that has not yet been written, then I assume that it existed on disk before the trace began. In addition, if there is any access to a block at offset x in a file, I assume that the entire file existed and was at least as large as x. All files identified in this way are written sequentially to disk before the trace begins. With initialization, the amount of live data increases from 2.66 GB before the trace begins to 3.15 GB at the end of the trace, an increase of 19%.

To vary the disk utilization, I choose the disk size such that when the amount of live data is at its peak, the disk is filled to the desired disk utilization. This underestimates the disk utilization for the majority of the trace, but avoids overflowing the disk when the data size reaches its peak.

5.2.2. Simulation Environment

To evaluate adaptive methods on a variety of workload patterns and technology trends, I used a flexible simulation environment. In this section, I describe the simulator I used. I also discuss the ways in which it is a simplification of the real world and the impact of these simplifying assumptions.

The bulk of the simulation infrastructure is an LFS simulator which is approximately 15,000 lines of C++ code. It interprets trace records that request a variety of operations including read, write, delete, truncate and sync. Records specify a unique file identifier and block offset for referenced blocks. If the trace contains traffic from more than one client, a unique client identifier indicates the origin of the request.

For the Auspex trace, I simulate both client and server caches.[1] When a block is read, the simulator checks the caches—first the client cache and then the server cache. If the data was found in the server cache, it is added to the client cache possibly replacing the least-recently used cache entry. If the data was not found in memory, the system determines the block's disk location and issues a disk read. In this case, a copy of the block is placed in both the server and the client cache. Unless specified, the client caches are 16 MB and the server cache is 128 MB. For the random update workload, I simulate a single machine; in this case, the client caches are eliminated.

Blocks written are placed first into a write buffer. Each write buffer contains as many blocks as a single segment. Writes to data already in the write buffer overwrite the existing contents, thus reducing disk traffic. When a write buffer is either full or receives a sync request, the data is written into an empty segment on disk. The write buffer is flushed every 30 seconds in addition to any sync requests. In a client-server simulation, more than one write buffer can be used in order to sep-

1. When simulating a client-server environment, it is the server's file system being simulated.

arate the traffic by originating client. This models the case in which the client buffers writes for up to 30 seconds before sending them to the server to be committed to disk.

The disk model computes the latency of each request as the sum of average seek time, average rotational latency and data transfer time. This does not capture the complexity of modern disks. Average access latency will be an overestimate for many disk requests because seek latency varies with seek distance and rotational latency varies with the position of the disk head in relationship to the requested data. This model also uses a single bandwidth for the whole disk, while modern disks have multiple zones with different bandwidths. The file system could deal with this added complexity by treating each zone (or group of zones with similar bandwidths) differently.

The cleaner runs when there are no more empty segments available for new data. The cleaner can choose several different garbage collection methods, including *traditional LFS cleaning*, *hole-plugging* [Wilkes et al., 1995] and an adaptive combination of cleaning and hole-plugging. (Each of these methods will be discussed in more detail in Section 5.4.) The cleaner can also chose from among a variety of policies for choosing segments to garbage collect, including *greedy*, which simply chooses the least utilized segment at each opportunity, and *cost-benefit* [Rosenblum and Ousterhout, 1992]. The cost-benefit policy chooses the segment that minimizes the formula $\frac{1+u}{a\times(1-u)}$, where u is the utilization of the segment and a is the age of the segment. For the rest of this chapter, I will refer to this policy as *cost-age* in order to avoid confusion with the other cost-benefit formulas presented in Section 5.4.2. The maximum amount of data that the cleaner may process at one time may be varied.

System meta-data, like segment utilization information and file meta-data, is maintained in the simulator, but is not reflected in the disk traffic. In terms of write performance, the main effect of this simplification is to reduce the overall traffic to disk. In terms of read performance, it will not

capture the degree to which data is located near its related meta-data on disk. Original LFS keeps the updated meta-data together with the updated data because they are written at the same time. Some of the techniques I discuss in this chapter improve write performance, but disturb the temporal locality of the write log. This could degrade read performance by separating data from its meta-data. In Chapter 6, I present a method for identifying and regrouping related data that becomes separated. This algorithm could also be used to regroup data and related meta-data.

Write performance is also sensitive to the data rate or the amount of idle time present in the workload because garbage collection costs can be completely hidden if sufficient idle time exists to perform garbage collection in the background. In this study, I always perform garbage collection in the foreground so that no garbage collection costs are hidden. Any idle time present in a real workload would only improve performance.

5.2.3. Performance Model

As a starting point, I use the write cost metric originally used in evaluating LFS write performance [Rosenblum and Ousterhout, 1992]. This original write cost model can be expressed with the formula found in Equation 1.

(EQ 1)

$$\begin{aligned} &WriteCost \\ &= \frac{SegmentsTransferred_{Total}}{SegmentsTransferred_{NewData}} \\ &= \frac{SegsWritten_{NewData} + SegsRead_{Clean} + SegsWritten_{Clean}}{SegsWritten_{NewData}} \end{aligned}$$

Write cost is an reflection of the cleaner overhead. It is the ratio of total work to the work necessary to initially write the new data to disk. The total number of segments transferred includes both the initial writes of new data ($SegsWritten_{NewData}$) and cleaner reads and writes ($SegsRead_{Clean} + SegsWritten_{Clean}$). Ideally, the data would be written once and never moved by the cleaner; this happens if all data in the segment is overwritten before the segment is reclaimed. In the best case, then, $SegsRead_{Clean} + SegsWritten_{Clean}$ is 0 and the write cost is 1.

Similarly, we can isolate simply the cleaner overhead as:

(EQ 2)

$$CleanerOverhead = \frac{SegsRead_{Clean} + SegsWritten_{Clean}}{SegsWritten_{NewData}}$$

In LFS, all writes, both the initial writes of new data and writes of old live data being coalesced by the cleaner, are performed in segment-sized batches. The cleaner also reads partially live segments in their entirety in order to clean them, before writing the live data back out as part of a new coalesced segment.

In the next section, I discuss how to modify these metrics to account for the impact of disk performance characteristics.

5.3. Understanding Write Cost: The Effect of Segment Size

In this section, I discuss why segment size plays a larger role in the write performance of LFS than has been previously suggested. In [Rosenblum and Ousterhout, 1992], segment size is chosen to be large enough that the access time becomes insignificant when amortized over the segment transfer. In Sprite LFS, a relatively large 1 MB segment is used.

On the other hand, there is a countervailing benefit to choosing a smaller segment size. [Rosenblum, 1992] observes that at smaller segment sizes the variance in segment utilizations is larger; allowing the cleaner to choose less utilized segments. In particular, smaller segments are more likely to empty completely before cleaning. Empty segments can simply be declared clean without requiring any disk transfers by the cleaner. In the limit, with one-block segments, cleaning costs would always be zero because all segments would be either full or empty and no data would need to be compacted. Of course, single block segments would eliminate any advantage from batched transfers.

In this section, I describe a way to quantify this trade-off between amortizing disk access times across larger transfer units and reducing cleaner overhead.

Figure 5-1 shows the results of varying the segment size for the Berkeley Auspex trace. According to the original definition of write cost in Equation 1, write cost is minimized at small segments because smaller segments reduce cleaner overhead. However, this does not reflect the inefficiency introduced by transferring smaller segments.

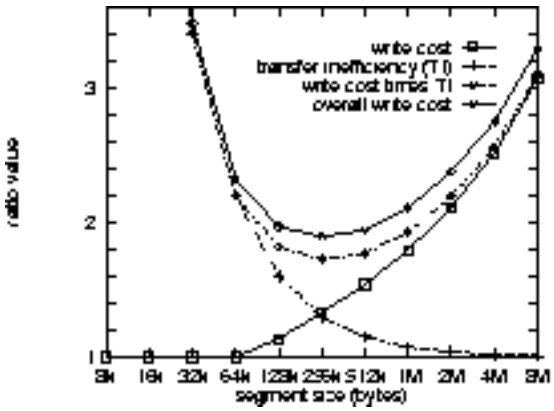

FIGURE 5-1. **Varying segment size for the Auspex workload.** Disk utilization is 85%; access time is 15 ms and bandwidth is 5 MB/s. Small segments are inefficient due to seek and rotational delay; large segments are inefficient due to fewer opportunities to find nearly empty segments. Overall write cost includes the impact of partial segments; write cost times TI does not. Write cost and overall write cost are simulated quantities. Transfer inefficiency is computed.

In Equation 3, I introduce a quantity to reflect this inefficiency. I define *transfer inefficiency* to be the ratio between the actual segment transfer time and the time it would have taken to transfer the segment at full disk bandwidth. Figure 5-1 plots this computed value across a range of segment sizes for a typical disk with a 15 ms access time and 5 MB/s bandwidth. As segments become large, the access time becomes insignificant relative to the time for transferring the segment, and therefore the transfer inefficiency approaches one.

(EQ 3)

$$TransferInefficiency_{Segs} = \frac{SegTransferTime_{Actual}}{SegTransferTime_{Ideal}} = \frac{AccessTime + \frac{SegmentSize}{DiskBandwidth}}{\frac{SegmentSize}{DiskBandwith}} = AccessTime \times \frac{DiskBandwidth}{SegmentSize} + 1$$

In Figure 5-1, we see that the write cost approaches 1 as the segment size decreases and the cleaner overhead drops to 0. Also, the transfer inefficiency approaches 1 as the segments become larger and we use the disk more efficiently.

The write cost in Equation 1 measures the overhead of cleaning. The transfer inefficiency in Equation 3 measures the bandwidth degradation caused by seek and rotational delay. In Equation 4, I introduce a new quantity, *overall write cost*, that captures both of these effects. The overall write cost is the total time required to write new data and clean segments, divided by the time to write just the new data at full disk bandwidth. If all disk transfers are in units of full segments, then this is simply the product of the original write cost in Equation 1 times the transfer inefficiency in Equation 3.

.

$$OverallWriteCost$$

$$= \frac{TransferTime_{Total}}{TransferTime_{Ideal}} \quad \textbf{(EQ 4)}$$

when all transfers are done in units of segments

$$= \frac{SegmentsTransferred_{Total} \times SegTransferTime_{Actual}}{SegmentsWritten_{NewData} \times SegTransferTime_{Ideal}}$$

$$= WriteCost \times TransferInefficiency_{Segs} \quad \textbf{(EQ 5)}$$

Figure 5-1 shows that this quantity does allow us to see the impact of the competing effects I have discussed. It is minimized at an intermediate segment size. Note that when the transfer inefficiency is 1, the overall write cost is equal to the original write cost. This is consistent with the assumption made in [Rosenblum, 1992] that the segment size is large enough that access time becomes insignificant. In Figure 5-1, the difference between Equation 4 and Equation 5 is due to the impact of partial segments.

Changes in disk characteristics affect the trade-off between cleaner overhead and transfer inefficiency. Figure 5-2 shows that the optimal segment size for the Auspex workload is approximately four times the product of disk access time and bandwidth (i.e., four times the amount of data that could be transferred during the time necessary to position the disk head). Figure 5-2 shows the overall write cost curves for the disks used in Sprite LFS (17.5 ms access time and 1.3 MB/s bandwidth) and for more modern disks (15 ms, 5 MB/s and 10 ms, 15 MB/s). This graph shows that for the Auspex workload a segment size of 64–128 kB would have been better than the 1 MB segments used in Sprite LFS. The optimal segment size has been increasing since then. This

suggests that to be able to scale with disk technology improvements, an LFS file system should measure and adapt to its underlying disk performance.

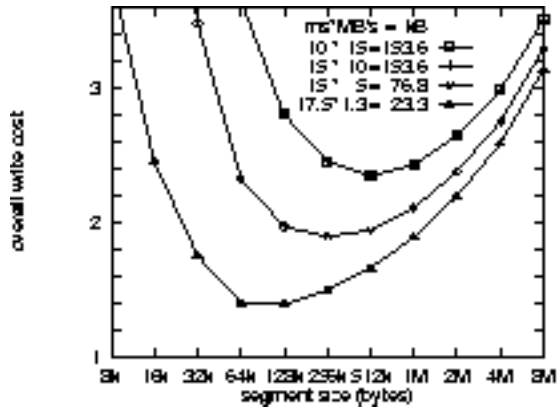

FIGURE 5-2. **Effect of disk characteristics on overall write cost for the Auspex workload.** Disk utilization is 85%. The bottom curve with access time of 17.5 ms and bandwidth of 1.3 MB/s represents the disks measured in Sprite LFS; note that Sprite chose a segment size of 1 MB. The middle curve represents the baseline disk simulated and the top curve represents a higher performance disk. Note that the curve is the same for different disks with the same access time bandwidth product. For all curves, overall write cost is minimized for a segment size of roughly four times bandwidth times access time. Overall write cost increases for faster disks because it is harder to match the peak disk performance.

Figure 5-3 shows overall write cost for the random update workload. Despite the inefficiency of single-block transfers, overall write cost is still lowest for single block segments (8 kB) because all cleaning overhead is avoided. (Note, however, segment header overhead is not included in the estimate of overall write cost.) With more than one block, there is little benefit to smaller segments. Because blocks are not overwritten in groups, segments empty slowly; even small segments stay nearly as full as the disk.

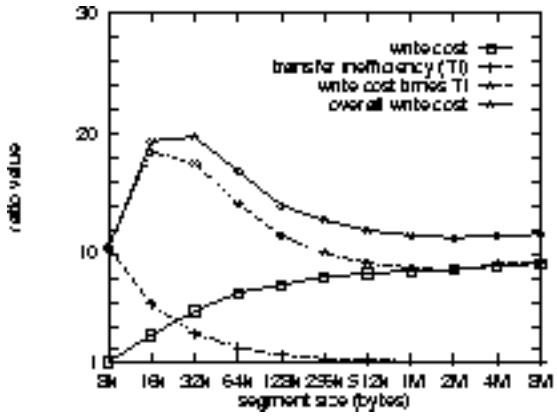

FIGURE 5-3. **Varying segment size for the random update workload.** Disk utilization is 85%; access time is 15 ms and bandwidth is 5 MB/s. One-block segments avoid all cleaning costs. Large segments benefit from larger transfers even though it is difficult to find low utilization segments to clean. Overall write cost includes the impact of partial segments; write cost times TI does not. Write cost and overall write cost are simulated quantities. Transfer inefficiency is computed. Note that the scale of the y-axis for the random workload graphs in this chapter differ from that for the Auspex graphs, for example in Figure 5-1 and Figure 5-2.

It may even be possible to vary the segment size dynamically by enabling the cleaner to observe the average length of the runs of holes in the segments it cleans; a workload with short runs might benefit from a smaller segment size. Another possibility would be to format the disk with several fixed segment sizes. One use would be to exploit the fact that different zones of the disk have different performance characteristics; the bandwidth between inner and outer tracks can vary by as much as 50%. Another use would be to allow data to be written into the smaller segments initially and then cleaned into the larger ones. For workloads with locality, recently written data is more likely to be overwritten; this would suggest using smaller segments to maximize the likelihood of emptying segments as all of their data is overwritten. By contrast, cleaned data tends to be older and less likely to be overwritten; this suggests using larger segments to better amortize disk access times. For the random update workload, newly written segments are not any more likely to empty and so would not benefit from the smaller segments, but at least the cleaned segments could benefit from the larger ones.

5.4. Adaptive Cleaning: Choosing the Best Garbage Collection Mechanism Based on Usage Patterns

In this section, I present an LFS cleaning algorithm that avoids the dramatic performance degradation seen at high disk utilization while retaining the good performance of traditional LFS cleaning at lower utilizations. It does this by dynamically choosing between two mechanisms: traditional LFS cleaning and hole-plugging [Wilkes et al., 1995]. The adaptive method successfully chooses the lowest cost mechanism based on the observed usage patterns.

5.4.1. Comparing Traditional Cleaning With Hole-plugging

In traditional cleaning, the live blocks in several partially empty segments are combined to produce a new full segment, freeing the old partially empty segments for reuse. In many environments, traditional cleaning performs very well. Idle time can often be exploited to hide cleaning costs from users; for the workloads examined in [Blackwell et al., 1995], 97% of cleaning could be done in the background. [McNutt, 1994] shows that cleaning costs are relatively low at disk utilizations below 80%. If segment updates show a high degree of locality, then some segments will be emptier than others and will yield more free space when cleaned. In addition, traditional cleaning preserves the temporal locality of the log because it keeps the live data gathered from segments together.

The problem with cleaning appears at high disk utilizations, especially for workloads with many random updates and insufficient idle time [Seltzer et al., 1993][Seltzer et al., 1995]. Because segments do not have a chance to empty before they must be cleaned, the cost of cleaning can skyrocket. In order to coalesce one free segment's worth of space, the cleaner must process many nearly full segments. Each segment must be read, and all but the few holes rewritten into a new segment. Recalling Equation 1, this translates into high $SegsRead_{Clean}$ and $SegsWritten_{Clean}$ and

therefore high write cost. In an extreme case, the entire disk might need to be cleaned in order to coalesce a single contiguous segment.

In hole-plugging, partially empty segments are freed by writing their live blocks into the holes found in other segments. In order to produce one free segment's worth of space, it is only necessary to read one segment and rewrite each of its live blocks. These writes are more expensive per block than writing complete segments because each block write requires additional seek and rotational delay. However, despite the higher per-block cost, at high disk utilizations, hole-plugging is still better than cleaning because we avoid processing so many segments. At lower disk utilizations, the larger cost of writing individual blocks makes hole-plugging more expensive than traditional LFS cleaning.

In order to compare traditional cleaning with hole-plugging, I introduce a write cost formula for hole-plugging in Equation 6. In the traditional LFS cleaning mechanism, all transfers are done in units of whole segments. However, with hole-plugging, some transfers are done in units of whole segments (the initial writes of new data, $SegsWritten_{Data}$, and segments read to be broken up into patches for holes, $SegsRead_{Clean}$), while other transfers are in units of individual blocks (the patches, $BlocksWritten_{Hole\text{-}plugging}$). In practice, the $TransferTime_{Block}$ varies based on the locality of blocks written. Seek latency is only incurred for the first block written to the segment, but each block does incur a full half rotation. It would make sense to minimize rotational delay by ordering the block writes when possible. The file system might have difficulty estimating the disk head position as it is constantly changing. However, some disk drives perform such reordering themselves. It should be possible by sending multiple discontiguous requests to the same segment to determine experimentally how well a disk drive is able to reorder multiple block writes to the same segment. There are, however, other per transfer overheads (e.g. controller overhead) that will

contribute to the experimentally determined per block overhead, but which cannot be removed by request reordering.

(EQ 6)

$$OverallWriteCost_{Hole\text{-}plugging} = \frac{TransferTime_{Total}}{TransferTime_{Ideal}} = \frac{TransferTime_{Total}}{SegmentsWritten_{NewData} \times SegTransferTime_{Ideal}}$$

$$where\ TransferTime_{Total} = TransferTime_{Seg} \times \langle SegsWritten_{Data} + SegsRead_{Clean} \rangle + TransferTime_{Block} \times BlocksWritten_{Hole-plugging}$$

There are several ways that hole-plugging could be integrated into an LFS. In existing LFS implementations, each segment has a segment header that contains information about its constituent blocks. In order to maintain this structure, the header would need to be read and updated for each segment patched. Two headers per segment would be required to prevent corruption. Alternatively, the per-block information in the segment header could be distributed into individual block headers. A 512-byte block header for each 8 kB block would be an overhead of 6.25%. Interspersed block headers would also reduce read bandwidth by the same amount. In Figure 5-4, I evaluate the space-time trade-off between these two strategies for the random update workload. The block header approach performs better at 99% utilization than the segment header approach does at 85% utilization—more than allowing for the 6.25% space overhead. Therefore, I use the block header approach for the rest of the experiments in this chapter.

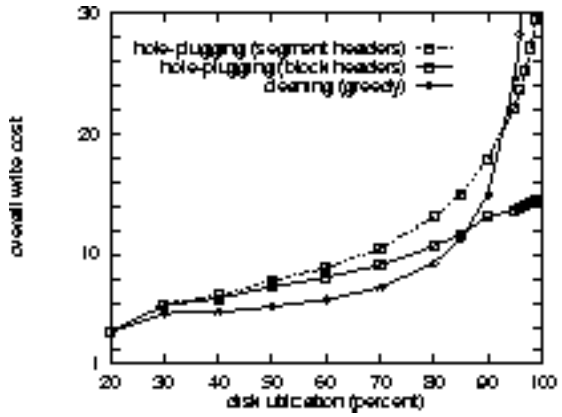

FIGURE 5-4. **Cleaning and hole-plugging for the random update workload**. Segment size is 256 KB; access time is 15 ms; bandwidth is 5 MB/s. Hole-plugging with block headers requires updating the block and its contiguously allocated header; otherwise the segment header must be read and written as well. Greedy cleaning is used because it is optimal for this workload; see Figure 5-6 for a comparison with cost-age. Although this point is not shown, at 99% utilization, the overall write cost for cleaning soars to 64.5.

I am careful to preserve the atomicity properties of the log when using the hole-plugging mechanism. Only after a hole is plugged do I update the log to point to the new locations. If the system crashes while overwriting the hole, there is no harm because, upon recovery, the system will continue to treat any partially overwritten hole as dead space. Also, the segments freed by hole-plugging should not be reused until the log has been updated to point to the new locations. If the system crashes before the updated meta-data is written, the old meta-data still points to the original segments which are intact. Therefore the following order should maintain the atomicity properties of the log: 1) plug holes in segments 2) update the log to point to the new locations 3) reuse the segments that were freed by hole-plugging.

Hole-plugging also requires a means to quickly identify which of a segment's blocks are really holes. To facilitate cleaning, a segment usage file is already maintained which lists the current utilization of each segment. The values in this file are update as newly written data occupies free segments and creates holes in previously written segments. It would be a minor change to replace the

segment utilization value with a bit vector indicating which blocks are alive. It would be updated in the same manner.

Figure 5-4 compares the write cost of cleaning with hole-plugging. Even for this worst-case workload, cleaning performs better than hole-plugging up through 85% disk utilization. However, above 85%, the overall write cost of cleaning shoots from below 10 to above 64. Hole-plugging degrades much more gracefully, staying below 15 for the block header approach.

Figure 5-5 shows the behavior of both cleaning and hole-plugging for the Berkeley Auspex workload. Cleaning performs as well or better than hole-plugging up to 99% utilization for this workload.

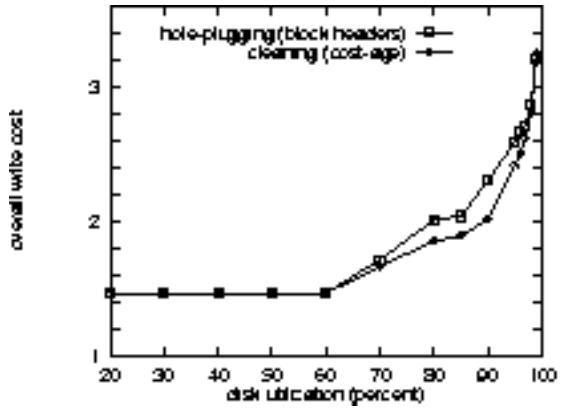

FIGURE 5-5. **Cleaning and hole-plugging for the Berkeley Auspex workload.** Segment size is 256 KB; access time is 15 ms; bandwidth is 5 MB/s. Cost-age cleaning is used. Cleaning performs as well or better than hole-plugging except above 99% disk utilization. Note the change in the scale of the y-axis relative to Figure 5-4.

Hole-plugging can be viewed as bringing some benefits of write-ahead logging to LFS. Write-ahead logging writes the new updates to the log and then later writes them on top of the “holes” that those updates created. Write-ahead logging offers consistent performance by paying the con-

stant cost of one batched write plus one in-place block write per block written. Similarly, one might expect hole-plugging costs to remain fairly constant. However, in Figure 5-4 and Figure 5-5, the cost of hole-plugging decreases with lower disk utilization. This is because at low disk utilization many segments empty completely before they must be processed and hole-plugging, unlike write-ahead logging, can benefit from this effect.

5.4.2. Adaptive Cleaning Policy

In order to retain the advantages of traditional cleaning while avoiding its dramatic performance degradation at high disk utilizations, I introduce a policy that chooses adaptively between cleaning and hole-plugging at each garbage collection opportunity. (Note: This is orthogonal to the policy used to choose which segments to clean.)

When garbage collection is needed, candidate segments are chosen for both traditional cleaning and hole-plugging. For cleaning, the candidates are those segments that will be compacted to form new segments. I simulated both greedy and cost-age cleaning policies. For hole-plugging, the candidates are those segments whose live blocks will be used to fill in the holes found elsewhere. As in AutoRAID, I use blocks from the least utilized segments to plug the holes in the most utilized segments.

Once the candidates have been identified, a cost-benefit estimate is calculated for each approach using the formulas found in Equation 7 and Equation 8.

In these equations, cost is expressed in terms of the total time to perform the garbage collection and benefit is expressed in terms of free space reclaimed. For hole-plugging, the cost is the time to read the candidate segments and write their live blocks into holes found in other partially empty segments; the space freed is the size of all the candidate segments read. For cleaning, the cost is the time to read the candidate segments and rewrite their live blocks as whole segments to

the end of the log; the space freed is the size of all the empty blocks found in the candidate segments.

(EQ 7)

$$CostBenefit_{Cleaning} = \frac{TransferTime_{Cleaning}}{SpaceFreed_{Cleaning}}$$

$$\text{where } TransferTime_{Cleaning} = (CandidatesRead + LiveBlocks/BlocksPerSeg) \times TransferTime_{Seg}$$

$$\text{and } SpaceFreed_{Cleaning} = EmptyBlocks \times BlockSize$$

(EQ 8)

$$CostBenefit_{Hole\text{-}plugging} = \frac{TransferTime_{Hole-plugging}}{SpaceFreed_{Hole-plugging}}$$

$$\text{where } TransferTime_{Hole\text{-}plugging} = CandidatesRead \times TransferTime_{Seg} + LiveBlocks \times TransferTime_{Block}$$

$$\text{and } SpaceFreed_{Hole\text{-}plugging} = CandidatesRead \times SegmentSize$$

Once these cost-benefit estimates have been calculated, we simply choose the mechanism with the lower estimate. Note that this decision applies only to the current garbage collection opportunity. At the next opportunity, the other approach may be chosen.

In Figure 5-6, I show cleaning, hole-plugging and the adaptive policy for the random update workload. I include greedy cleaning as well as cost-age since greedy has been shown to have slightly better performance than cost-age on a random workload [Rosenblum and Ousterhout, 1992][Seltzer and Smith, 1995]. The adaptive policy correctly shifts from cleaning to hole-plugging at the appropriate point. I show that we are indeed able to retain the good common case per-

formance of traditional cleaning while avoiding its dramatic performance degradation at high disk utilizations.

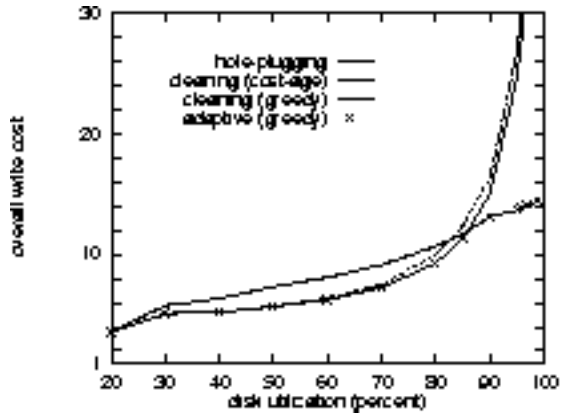

FIGURE 5-6. **Adaptive cleaning for the random update workload.** Note that the hole-plugging and greedy cleaning curves are the same as in Figure 5-4. The adaptive algorithm chooses between hole-plugging and greedy cleaning; it correctly follows the lower cost mechanism at each point.

In Figure 5-7, I show cleaning, hole-plugging, and the adaptive policy for the Berkeley Auspex workload. Notice that at some points the adaptive policy performs better than the minimum of cleaning and hole-plugging by doing each when appropriate. Also note that below 60% disk utilization, the overall write cost is constant. In this region, no garbage collection is required because there is enough free space that segments have time to empty on their own before the system runs out of clean segments for new log writes.

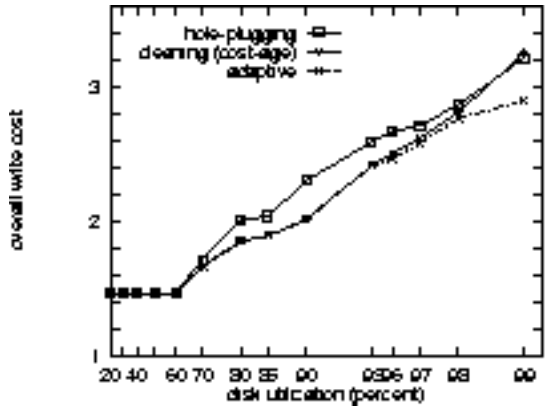

FIGURE 5-7. **Adaptive cleaning for the Berkeley Auspex workload.** Note that the hole-plugging and cost-age cleaning curves are the same as in Figure 5-5. The x-axis is on a reverse log scale in order to show clearly the region above 90%. Adaptive outperforms both hole-plugging and cleaning because it can choose the appropriate method at each garbage collection opportunity.

This adaptive method could also be used to adapt between any additional garbage collection mechanisms given a correct cost-benefit model of their behavior.

Changes in disk characteristics also have an impact on the trade-off between cleaning and hole-plugging, making the need for adaptive cleaning even more acute. Disk bandwidth has been improving faster than disk access times, resulting in higher relative block transfer costs. Figure 5-8 shows that on a faster disk the gap between hole-plugging and cleaning is larger at lower disk utilizations and that the crossover point is later. Similarly, on RAID systems, hole-plugging would be penalized relative to cleaning because of the need to read the old data in blocks being plugged in order to update parity.

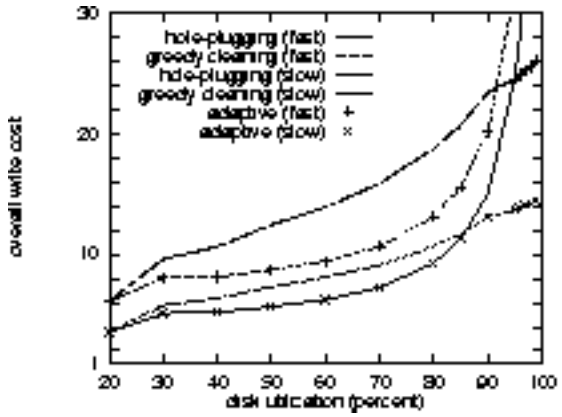

FIGURE 5-8. **Effect of disk characteristics on the trade-off between cleaning and hole-plugging, for the random update workload.** Note that the slow curves are the same as in Figure 5-6, using a disk with 15 ms access time and 5 MB/s bandwidth. The fast curves use a disk with 10 ms access time and 15 MB/s bandwidth. Cleaning performs relatively better than hole-plugging on the fast disk because of the larger gap between block transfer efficiency and segment transfer efficiency. Overall write cost increases for the fast disk because it is harder to match the peak disk performance.

5.5. Using Cached Data To Reduce Write Cost

In this section, I describe how to further reduce cleaning costs by taking advantage of data that is already cached. When a segment is completely cached, it can be cleaned by writing its live blocks—there is no need to do a disk read. This lowers the $SegmentsRead_{Clean}$ component of write cost in Equation 1. As far as I know, no LFS implementation performs this optimization.

In exploring this possibility, I consider two different cleaning policies: normal cost-age in which cached data is not used, and a modified cost-age (*cost-age-cache*) in which fully cached segments are preferentially chosen by taking into account that a segment is cached in the cost-age formula — in effect, adapting to the contents of the cache. For this modified policy, when a segment is cached, the cost portion of the cost-age function includes only the cost to write out the live blocks and not the cost to read the complete segment.

I implemented the modified cleaning policy in the LFS simulator by keeping an in-memory set of cached segments; its size is limited to the number of complete segments that fit in memory. As a block leaves the cache, I check this set and remove its segment if necessary. I track only segments that remain completely cached after being written; detecting when full segments re-enter the cache would complicate the implementation for only marginal benefit. I show significant improvement even though I do not take advantage of segments that are re-cached.

In Figure 5-9, I show the impact of increasing server cache size on overall write cost for the Berkeley Auspex workload at various disk utilizations. The top group of lines illustrates the behavior when the disk is 95% utilized. The next two groups of lines are with the disk at 85% and 60% utilization, respectively.

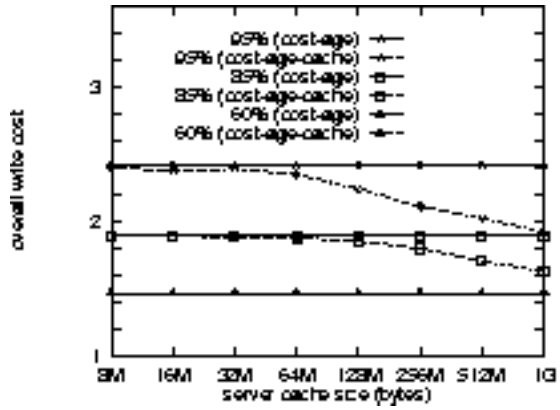

FIGURE 5-9. **Varying server cache size for the Auspex workload.** Segment size is 256 KB; access time is 15 ms; bandwidth is 5 MB/s. Three different disk utilizations are shown for both normal cost-age and cost-age that uses cached data. The client cache size is set to zero as described in Figure 5.2.2. This graph shows the reduction in overall write cost obtained by exploiting cached data during cleaning. The benefit is greater at higher disk utilizations.

As expected, the performance of the cost-age policy is insensitive to cache size. Indeed, all of the cost-age lines are flat. For the cost-age-cache policy, there is more benefit with larger caches as one would expect.

There is incremental benefit even up through 1 GB, indicating that the working set of the Auspex trace is larger than 1 GB. This is unsurprising considering that the total data held on the server was approximately 100 GB of binaries and home directories. At a cache size of 256 MB and disk utilization of 95%, there is an 11% reduction in overall write cost corresponding to a 30% reduction in cleaning overhead.

In addition, the benefit of cleaning from cache increases as the utilization of the disk increases. To illustrate why, consider the effect on the cost-age formula for cleaning a single segment as the utilization of that segment increases. For a highly utilized segment, less space is reclaimed and therefore the read that we avoid has higher cost relative to the amount of space reclaimed. To see this quantitatively, consider the cost-age formulas. Recall from Section 5.2.2 that when we are unable to use cached data to avoid reading the segment, the cost-age of the segment is $\frac{1+u}{a\times(1-u)}$, where u is the percentage of live blocks in the segment and a is the age of the segment. When we are able to use cached data to avoid the read, the cost-age drops to $\frac{u}{a\times(1-u)}$. Their difference, $\frac{1}{a\times(1-u)}$, is larger for segments with greater utilization.

As overall disk utilization increases, LFS will have to clean segments with higher utilization. As a result, the increased benefit for fuller segments translates directly into increased benefit for fuller disks. Interestingly, this means that using cached data is especially helpful in addressing the worst-case performance of LFS at high disk utilizations.

Also, notice that we begin to see benefit at smaller cache sizes as the utilization increases. At lower utilizations, we can wait longer to clean; in that time, more of the segments we are interested in cleaning have been evicted from the cache.

5.6. Putting It All Together

Figure 5-10 shows the combined impact of the optimizations I have discussed in this chapter relative to original LFS. There is up to a 20% reduction in overall write cost for the Berkeley Auspex trace and an up to four-fold reduction for the random workload. That corresponds to a 42% and almost six-fold reduction in cleaner overhead, respectively. Log scale is used to clearly display both workloads.

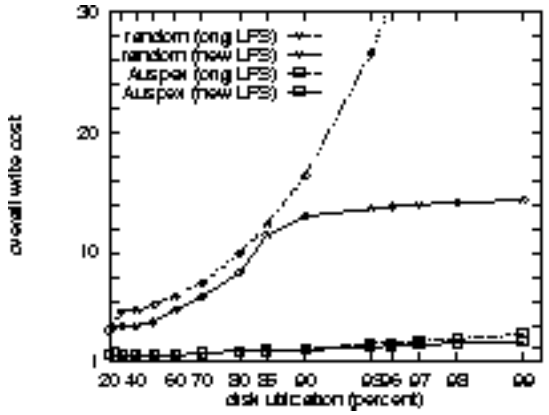

FIGURE 5-10. **Overall write cost of original LFS versus modified LFS**. Segment size is 256 KB; server cache size is 128 MB; access time is 15 ms; bandwidth is 5 MB/s; client cache size is 16 MB. Note the log scale on both axes. This graph shows the aggregate effect on overall write cost of using both adaptive cleaning and cached data. The segment size is the same for all curves. However, an additional benefit would be obtained for the Auspex trace if the Sprite LFS segment size of 1 MB was used for the original LFS curves (see Figure 5-1–Figure 5-2).

5.7. Related Work

In AutoRAID, hole-plugging is used rather than traditional LFS cleaning or an adaptive combination of the two, because AutoRAID is structured such that hole-plugging always performs better. The AutoRAID consists of two areas: the mirrored or RAID-1 area, which houses recently updated data, and the RAID-5 area, which houses older data. The RAID-5 area is the part of AutoRAID that is log-structured. It is always at high utilization because it is constantly cleaned in order to return PEGs (segments) to the free pool where they can be used for mirrored writes or fresh demotions into the RAID-5 log. In addition, the updates to the RAID-5 area are fairly random

because the mirrored storage area absorbs the updates to the hot data. The remaining update stream reaching the RAID-5 applies to cold data and shows very little locality in practice. Thus, the AutoRAID environment is the worst possible case for traditional cleaning (high utilized segments that are updated randomly). AutoRAID does PEG-cleaning only in the special case that there are no holes to be plugged; that is when all PEGs but one are full or empty. In this case, the live blocks from that single PEG are appended to the end of the RAID-5 write log [Wilkes et al., 1995].

Golding et al. explored useful optimization tasks that could be performed in idle time [Golding et al., 1995]. They discuss adapting the amount of idle processing attempted based on a prediction of how long the current idle period will last.

In many ways, this work had its roots in Margo Seltzer's analysis of the performance problems in LFS, especially for the TPC-B transaction processing benchmark and in the discussion of the relative merits of LFS and FFS which arose from that analysis [Seltzer et al., 1993] [Ousterhout, 1995a] [Seltzer et al., 1995] [Ousterhout, 1995b][Seltzer and Smith, 1995] [Ousterhout,1995c].

5.8. Conclusions

In this chapter, I have presented adaptive methods that have targeted LFS write performance. They enable LFS to avoid its dramatic worst case performance for random updates at high disk utilization. In addition, they improve common case performance by allowing write performance to scale with improvements in disk performance. In the next chapter, I will focus on improving read performance.

6 Providing Efficient File System Read Access

In Chapter 5, I focused on LFS write performance. In this chapter, I consider how to improve file system read performance and how to balance read performance with write performance.

6.1. Motivation

In Chapter 3, I discussed how existing systems are either read-optimized or write-optimized. LFS and write-anywhere systems are write-optimized because their data layout policies focus on minimizing seek and rotational delay for writes — LFS by batching writes and write-anywhere systems by writing to free space close to the current disk head location. On the other hand, update-in-place and write-ahead logging systems invest in additional seek and rotational delay in order to group data based on its position in the directory hierarchy (i.e. blocks in the same file together and files in the same directory together).

In Chapter 4, I discussed how workloads can vary along many dimensions including the ratio of read to write traffic and the pattern of read and write accesses. Certainly, the relative effectiveness of read and write optimized systems will be affected by these variations in workload. Both read and write optimized systems have "blind spots". Read-optimized systems invest in careful placement for all data—even data that is quickly deleted or overwritten, data that is not read in any predictable way and data that is read predictably but not with other data located nearby in the directory hierarchy. Write-optimized systems do not invest in careful placement for any data—

even when data that is routinely read together is scattered across the disk by the default write policies.

To address these problems, I present an adaptive reorganization algorithm that monitors actual access patterns and invests in careful data placement only where past access patterns suggest a benefit. I show how to integrate this algorithm into a write-optimized file system to balance read and write performance. I examine the performance under a variety of access patterns, disk utilizations and disk performance characteristics.

6.2. Methodology

6.2.1. Workloads

As in Chapter 5, I use both measured and synthetically generated traces. I use synthetically generated traces to examine how the read performance of existing file systems and my dynamic reorganization algorithm vary with factors such as read and write patterns, disk utilization and disk performance.

I use three simple microbenchmarks. The first is a sequential write of a large file followed by sequential read of the same file. The second is random write of a large file followed by sequential read. The third writes the file in a non-sequential pattern and then reads it in the same non-sequential pattern.

I also use a portion of a long term trace in an academic environment to evaluate the impact of dynamic reorganization on a real workload [Roselli, 1998]. Roselli monitored fourteen desktop machines belonging to the graduate students, staff and faculty of a computer science research group for over one year [Roselli, 1998]. She records user level file system activity by capturing all system calls made to the file system.

In addition, I use a trace of the TPC-D decision support benchmark running on a qualification scale database in Microsoft SQLServer [Matthews, 1999]. TPC-D consist of 17 complex business oriented queries [TPC-D, 1995]. The database uses 22 separate files: 8 files containing the TPC-D tables, 8 files containing non-clustered indices on those tables, 5 containing database meta-data including the recovery log and 1 containing a temporary working space for intermediate results. Each file represents a fixed size portion of space reserved within SQLServer for the specified purpose. The data is accessed through the file system and I collected this trace using Filemon, a file system tracing tool implemented as a filter driver interposed above the NTFS drivers [Russinovich and Cogswell, 1997].

6.2.2. Simulation Environment

To evaluate various workloads on various file system architectures, I use a series of file system simulators. For LFS, I use the LFS simulator described in Chapter 5. The adaptive write optimizations discussed there are used throughout this chapter. I also add a dynamic reorganization algorithm which I describe in Section 6.4.

In addition, I developed simple simulation models of FFS and write-anywhere systems. More sophisticated simulators would be required to evaluate their performance on more general workloads, but for the three microbenchmarks, these simple simulators are sufficient. The FFS simulator simply places the single large data file accessed by each microbenchmark sequentially on disk. (Note that write-ahead logging systems produce the same semantic layout as FFS does so there is no need to consider the layout of such systems separately.) The write-anywhere simulator allocates each new block to the free location closest to the disk head. It continues searching for free locations in one direction until it has reached the edge of the disk at which point it turns and searches for free locations in the opposite direction [Wang et al., 1999]. Like the LFS simulator, neither the FFS or write-ahead simulators model accessing system meta-data on disk.

For all three simulators, I record the sequence of disk accesses they produce and feed them into several low level disk simulators. I use the Dartmouth disk simulator [Kotz et al., 1994] which simulates a 1.2 GB HP97560 disk. For this disk, the bandwidth is 2.10 MB/sec and the average access time (i.e. time for a random seek and one-half rotation) is approximately 19 msec. For comparison, I also use the DiskSim disk simulator [Ganger et al., 1998] to simulate a 1.05 GB HPC3323A. For this disk, the average bandwidth is 4.0 MB/sec and the average access time is approximately 14 msec. I allow the disks to prefetch up to 128 KB of adjacent data with each read request. These disks are both significantly slower than modern disks. However, as detailed models of two generations of disk drives, they allow me to explore the impact of improvements in disk drive technology (Section 2.2 on page 14).

I quantify the effect of the fragmentation and disk utilization on both existing file system layout policies and on dynamic reorganization. To control disk utilization, for the synthetic traces I simply adjust the total amount of data written to fill the disk to the desired degree. For the measured traces, I adjust the amount of disk space using portions of a single disk or multiple disks if necessary.

6.3. Microbenchmark Read Performance of Existing File Systems

In this section, I illustrate the read performance characteristics of LFS, FFS and write-anywhere systems using several microbenchmarks. LFS and write-anywhere systems are considered write-optimized because their layout policies are based on making write accesses more efficient. FFS is considered read-optimized because it pays a higher write cost to place data in a specific organization designed to improve read performance.

To evaluate the layout policies of these systems, I use three simple microbenchmarks, each involving a single 625 MB file. All benchmarks begin by writing the entire file sequentially. The first benchmark follows this with a sequential write of the file and a single sequential uncached read of the same file. The second follows with a random write and a sequential read. The third writes the file in a non-sequential pattern and then reads it in the same non-sequential pattern.

These microbenchmarks are intended not to show real-life performance, but rather to illustrate the differences between existing systems. They will show that no single system performs well across the board, even for these simple access patterns. Intuitively, we expect LFS and write-anywhere systems to perform well for read patterns that match the write pattern and FFS to perform well for sequential read patterns.

To include the effects of fragmentation and garbage collection, I examine the performance of these benchmarks at two different disk utilizations. In the first case, the file is written to an empty disk; the file is 50% of the total disk space. In the second case, I first fragment the free space on the disk by allocating 40% of the disk blocks to data that is not referenced again during the benchmark. This preallocated data is placed evenly throughout the disk in chunks of 8 blocks at a time. The benchmark file is again 50% of the total disk space, yielding 90% utilization.

In Figures 6-1 through 6-6, I report the cumulative disk access times as reported by the Dartmouth disk simulator on the x-axis. The y-axis reflects the logical phases of the benchmark. In each graph, the region from 0 to 625 MB is the initial sequential write of all the benchmark data. The region from 625 to 1250 MB is the write pattern specified by the benchmark and the region from 1250 to 1875 MB is the read pattern specified.

The cumulative time on the x-axis reflects all data transferred to disk including data transferred by LFS during garbage collection; the y-axis reflects only user data transferred. This allows

the progress of each system on the benchmark to be directly compared. Steeper slopes indicate higher performance as more user data is transferred in less time.

6.3.1. Sequential Write Followed By Sequential Read

Figures 6-1 and 6-2 show the results for sequential write followed by sequential read. We would expect all three systems to perform well for this test because the read pattern follows both the write pattern captured by the write-optimized systems and the sequential pattern captured by FFS. In Figure 6-1, we see that the behavior clearly matches this expectation at 50% utilization.

Figure 6-2 shows similar behavior at 90% utilization, but all three systems show the effects of the fragmentation of free space at the higher disk utilization. In Figure 6-1, nearly full disk bandwidth is achieved, but performance is lower for all systems in Figure 6-2. (Notice the different scales of the x-axes.)

During the initial sequential write, LFS performance is slowed due to garbage collection costs. Before it can write the data, it must first generate free segments for new data writes by compacting the scattered free space. LFS garbage collection costs are negligible during the second sequential write — once the file is written, each new segment write tends to free one entire segment of previously written data.

The second sequential write and the sequential read proceed more efficiently for LFS than for either FFS or write-anywhere because LFS pays garbage collection costs initially to move the preallocated blocks out of the way. FFS and write-anywhere allocate the benchmark data around the preallocated blocks and therefore pay the penalty to skip over them for each write and read. The flatter portions of the write-anywhere curve correspond to writing data (or reading data written) in regions of higher disk utilization. Notice that the write-anywhere curve begins to change when approximately 740 MB or approximately 60% (50% data and 10% free space) of the 1.2 GB disk

has been written. This is the point at which the disk has been completely written once and the 10% free space is scattered across the disk.

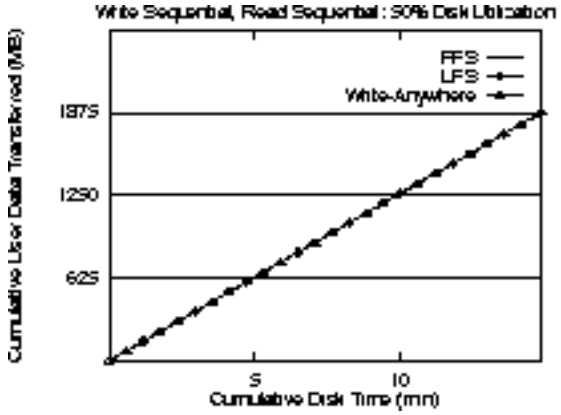

FIGURE 6-1. **Write Sequential/Read Sequential, 50% Disk Utilization.** This figure shows the performance of FFS, LFS and write-anywhere for a microbenchmark that consists of an initial sequential file write followed by a second sequential file write and a sequential file read. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB is another sequential write of all the benchmark data. From 1250 to 1875 MB is a sequential read of all the benchmark data. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator . The LFS simulator used includes the adaptive optimizations presented in Chapter 5.

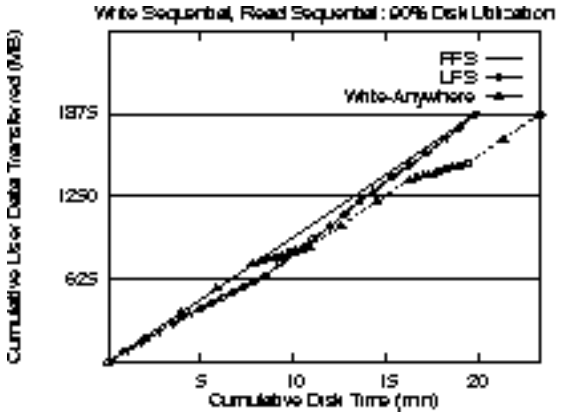

FIGURE 6-2. **Write Sequential/Read Sequential, 90% Disk Utilization.** This figure shows the performance of FFS, LFS and write-anywhere for a microbenchmark that consists of an initial sequential file write followed by a second sequential file write and a sequential file read. Before the benchmark begins, the disk is fragmented and the disk utilization increased by allocating 40% of the disk blocks to data not referenced again during the benchmark. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB is another sequential write of all the benchmark data. From 1250 to 1875 MB is a sequential read of all the benchmark data. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator. The LFS simulator uses the adaptive optimizations presented in Chapter 5.

6.3.2. Random Write Followed By Sequential Read

Figures 6-3 and 6-4 show the results for the microbenchmark that performs a random write followed by a sequential read. We expect FFS to perform poorly for the random write, since it must seek for every block; by contrast, the read should proceed quickly since it follows the sequential file layout. In contrast, we expect the write-optimized systems to perform well for the file write, but poorly for the file read since it does not match the write pattern. Figure 6-3 shows that at 50% utilization the behavior clearly matches these expectations. However, the behavior changes somewhat at 90% disk utilization as shown in Figure 6-4.

As in Figure 6-2, LFS performs slightly worse than FFS and write-anywhere for the first sequential write due to the initial garbage collection overhead. In the random write portion of the benchmark, write-anywhere performs the best because it is able to place new data writes into any available free space. Using the techniques outlined in Chapter 5, LFS is able to maintain reasonable performance despite the random updates and high disk utilization. FFS performs the worst as it struggles to maintain the sequential layout despite the non-sequential write patterns. However, this investment in maintaining the sequential layout enables FFS to outperform both write-anywhere and LFS for the sequential read. If the file were read repeatedly, FFS would extend its lead over the other alternatives.

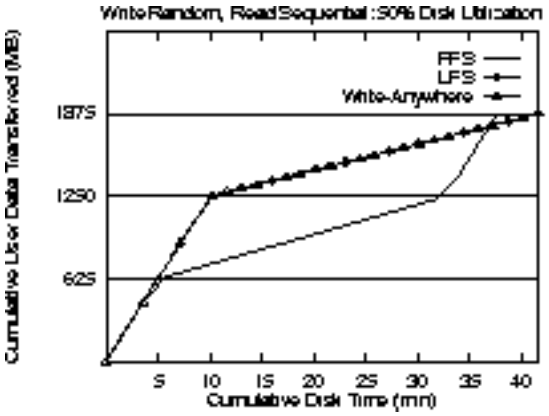

FIGURE 6-3. **Write Random/Read Sequential, 50% Disk Utilization.** This figure shows the performance of FFS, LFS and write-anywhere for a microbenchmark that consists of an initial sequential file write followed by a random file write and a sequential file read. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB is a random write of all the benchmark data. From 1250 to 1875 MB is a sequential read of all the benchmark data. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator. The LFS simulator uses the adaptive optimizations presented in Chapter 5.

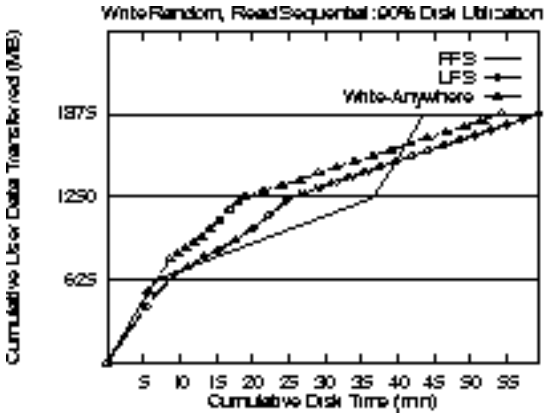

FIGURE 6-4. **Write Random/Read Sequential, 90% Disk Utilization.** This figure shows the performance of FFS, LFS and write-anywhere for a microbenchmark that consists of an inital sequential file write followed by a random file write and a sequential file read. Before the benchmark begins, the disk is fragmented and the disk utilization increased by allocating 40% of the disk blocks to data not referenced again during the benchmark. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB is a random write of all the benchmark data. From 1250 to 1875 MB is a sequential read of all the benchmark data. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator. The LFS simulator uses the adaptive optimizations presented in Chapter 5.

6.3.3. Writing and Reading the Same Non-Sequential Pattern

Figures 6-5 and 6-6 show the results for writing and reading the same non-sequential pattern. We expect FFS to perform poorly for both reads and writes in this case and the write-optimized systems to perform well for both reads and writes. This is clearly seen for the 50% disk utilization experiment shown in Figure 6-5. Figure 6-6 shows the 90% utilization test. During the nonsequential write, FFS again performs the worst among the systems being studied because it maintains the sequential layout despite the nonsequential write stream. However, in this case, the effort FFS invests in careful data placement is not justified by an improvement in read performance. During the read portion, LFS outperforms both FFS and write-anywhere because it best preserves the pattern of the write stream, although doing so takes more time for writing the file than with the write-anywhere system. This nearly balances in this case, but would favor LFS if the file were read repeatedly.

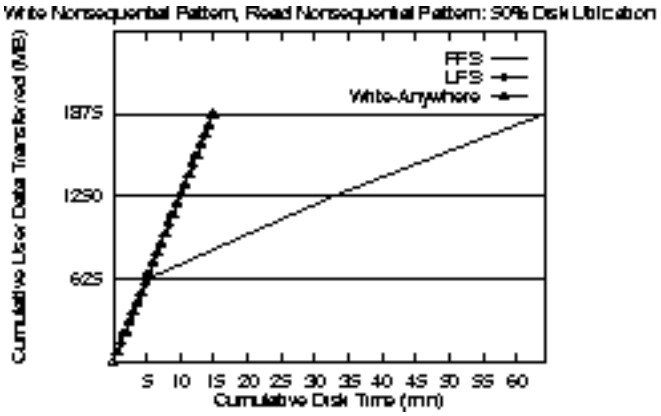

FIGURE 6-5. **Write and Read Non-Sequential Pattern, 50% Disk Utilization.** This figure shows the performance of FFS, LFS and write-anywhere for a microbenchmark that consists of an inital sequential file write followed by writing and reading the file in the same non-sequential pattern. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB, all the benchmark data is written in a non-sequential pattern. From 1250 to 1875 MB all the benchmark data is read in the same pattern. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator. The LFS simulator uses the adaptive optimizations presented in Chapter 5.

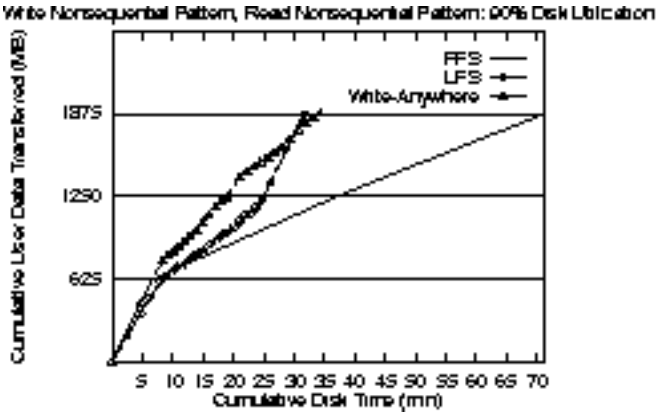

FIGURE 6-6. **Write and Read Non-Sequential Pattern, 90% Disk Utilization.** This figure shows the performance of FFS, LFS and write-anywhere for a microbenchmark that consists of an inital sequential file write followed by writing and reading the file in the same non-sequential pattern. Before the benchmark begins, the disk is fragmented and the disk utilization increased by allocating 40% of the disk blocks to data not referenced again during the benchmark.The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB, all the benchmark data is written in a non-sequential pattern. From 1250 to 1875 MB all the benchmark data is read in the same pattern. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator. The LFS simulator uses the adaptive optimizations presented in Chapter 5.

6.3.4. Microbenchmark Summary

These three simple benchmarks illustrate that neither LFS nor FFS nor write-anywhere systems dominate all workload patterns. Of course, their performance in any given environment will depend on the interaction of reads and writes in the workload, the degree to which access patterns follow file and directory units and the other factors I discussed in Chapter 4. Still, it is clear from these results that LFS, FFS and write-anywhere systems are all unresponsive to actual read patterns. They perform well when read patterns happen to match the default layout and poorly when they do not. Data layout is further complicated by fragmentation and garbage collection, especially at higher disk utilizations, making it difficult to determine when read performance might be poor even for read patterns that do match the systems default layout. In the next section, I will present an algorithm for reorganizing data on disk to match actual read patterns and for balancing read performance with write performance.

6.4. Dynamic Reorganization

In this section, I discuss how to produce a disk layout that effectively supports a wider variety read access patterns by (i) dynamically identifying related data, (ii) regrouping the related data and (ii) balancing read and write performance. My intent is not to identify the ideal data reorganization algorithm, but rather to show that it is feasible and that such adaptation is necessary to provide robust file system read performance in a single system.

6.4.1. Identifying Related Data

Access graphs are a natural way of dynamically identifying related data [Tsangaris and Naughton, 1991] [Griffioen and Appleton, 1994]. I build an access graph in which each node represents a file block. An edge from node A to node B indicates that block B was read from disk immediately after block A. The edges are weighted by the number of such accesses. Many of the

graph edges can be eliminated while still capturing the strong relationships for which it is most important to optimize. Intuitively, data blocks either have a few strong connections or many weak connections. Specifically, I record four outgoing edges for each data block and use LRU replacement within this set. For 8KB data blocks, this requires a storage overhead of about 1%.

The next challenge is to find a disk layout strategy to optimize for the read patterns captured in the access graph. The optimal layout strategy would place blocks that are frequently accessed together close to one another in order to minimize seek and rotational delays. Finding such a layout is an application of the more general irregular graph partitioning problem. Maximizing the number of internal edges within partitions and minimizing the number of external edges produces a partitioning of the file blocks such that blocks in the same partition are frequently accessed together, while blocks in different partitions are rarely accessed together.

The general irregular graph partitioning problem is NP-complete, but there exist heuristic solutions in the literature [Barnard and Simon, 1993][Hendrickson and Leland, 1995]. I use a simple dynamic graph partitioning algorithm based on these heuristic solutions. For each disk read, I increment the weight of the edge between the current block and the previous block read or create a new edge if one does not already exist. If the previous block is in a different partition than the current block, I shift existing partition boundaries to bring the nodes in question together if doing so results in new partitions that reduce the total weight of inter-partition edges.

6.4.2. Data Reorganization

I use the partitioned access graph to guide targeted improvements to the existing data layout. This information could actually be used in any of the existing systems I have discussed to regroup related data that is scattered by the system's default layout policy, but in this study, I investigate dynamic reorganization within the context of an LFS.

LFS is an especially attractive choice for several reasons. First, given the current trend towards write-optimized file systems, it is particularly interesting to investigate improving read performance in the context of a write-optimized system. Second, LFS includes support for on-line data movement as it is required for garbage collection.

Such reorganization of the data on disk is difficult in some systems. For example, WAFL supports its snapshots by way of a file mapping each physical disk block to the snapshots which refer to it [Hitz et al., 1995]. (A block can be reused when no remaining snapshot refers to it.) Moving data to a new physical location would require transferring this information and modifying all the snapshots which point to it. Since snapshots are read-only once taken, this would violate a principle design constraint in the system.

To support dynamic reorganization, I conceptually divide the LFS log into two logs: the main write log and the reorganization log. These two logs are logically separate, but are threaded through the same collection of physical disk segments much like mirrored and RAID-5 data coexist in AutoRAID [Wilkes et al., 1995]. Any segment can contain data from the main write log or the reorganization log.

Reorganizing a partition of the access graph consists of writing all of its members together into the reorganization log. Unlike in the main write log, I attempt to place logically sequential segments of the reorganization log together on disk to accommodate groups of data that span segment boundaries (e.g. streams of related data or groups of related data that are larger than a single segment). When I need to write a segment to the reorganization log, I choose the segment immediately following it on disk if it is available.

Partitions are eligible for reorganization if the partition members are not already clustered together and if the predictive value of the edges joining the partition members has been demon-

strated. (I define the value of a partition to be the sum of the weight of all inter-partition edges divided by the number of data blocks in the partition and require a partition's value to exceed 2 before it can be reorganized.) I currently reorder candidate partitions immediately when they become eligible, but partitions could be reorganized at other convenient times, such as during idle periods, when a partition is brought into memory, or when a partition is about to be evicted from memory.

6.4.3. Balancing Reads and Writes

The access graph also serves as a tool for balancing read and write performance. When a block that is currently in the reorganization log is written, I first write a copy of the new data at the tail of the main write log. Then, while the block is still in memory, I patch a copy on top of its previous location in the reorganization log. (This preserves the recovery property of the log without disturbing the read locality of the cluster.) This can be viewed as a special case of hole-plugging garbage collection in which a new block is placed in its previous location.

The second write into the reorganization log adds to the write cost, but whenever a block in the access graph is written to disk, I decrement the weight of that block's edges. Therefore, every time a block is written, its ties to the other partition members weaken. Eventually, if the block is written more frequently than it is read, its connections to other partition members become severed and it will no longer be re-written into the reorganization log.

Overall, this approach allows data to move adaptively between read-optimized and write-optimized storage depending on the dominant access pattern. Data written, but not read in a predictable manner, will be write-optimized. Data written and then read repeatedly in a predictable way will be read-optimized. When access patterns alternate between reads and writes, the data will also shift between read-optimized and write-optimized storage.

This is similar to the write-ahead logging approach I discussed in Chapter 3. Write-ahead logging file systems first write new updates into a log and then into the semantically-organized main file system. This approach, however, allows for non-semantic organizations. Another important difference is that much of the data written does not migrate into the reorganization log at all, e.g. data for which the default policy arranges data as it is read, data which is overwritten or deleted before being read frequently and data which is not read predictably. In this way, dynamic reorganization focuses reorganization efforts— investing in careful placement only when the value of that investment has been demonstrated.

6.5. Performance of LFS With Dynamic Reorganization

In this section, I examine the performance of dynamic reorganization. First, I use microbenchmarks to explore how its performance is affected by various read and write patterns, disk utilizations and disk performance characteristics. Second, I examine its performance on a long-term trace of file system usage taken in an academic environment and a trace of the TPC-D decision support benchmark.

6.5.1. Microbenchmarks

I begin my evaluation by comparing LFS to LFS with dynamic reorganization for the microbenchmarks described in Section 6.3.

6.5.1.1. Adjusting to Observed Read Patterns

Figure 6-7 shows all three microbenchmarks at 50% disk utilization. The area from 0 to 1875 MB is the original benchmark. In all cases, there is no change between LFS and LFS with dynamic reorganization in this region because the read pattern has only occurred once and no prediction can

be made without historical information. This could mean lost organizational opportunities for workloads in which data is read only once. However, it allows reorganization efforts to be focused on data for which the value of a new organization has been demonstrated.

I extend the read portion of the benchmark by repeating the same read pattern four additional times. During the first read, dynamic reorganization observes the read pattern. During the second read, the graph edges demonstrate their predictive value and dynamic reorganization begins to adjust the disk layout where appropriate. For two of the access patterns, the original layout produced by LFS already matches the read pattern and therefore no corrections are made. However, for random write followed by sequential read, disk requests are issued to adjust the data layout to the observed patterns. This reorganization traffic causes a temporary drop in performance, but the remaining reads benefit from the resulting disk layout. In this test, it takes approximately 2.5 sequential read passes to amortize the cost of the reorganization. Although I do not attempt it here, it may be possible to decide whether to reorganize a partition based on both the projected cost of reorganization and a prediction of the number of times the partition will be read before being overwritten or deleted. Such a prediction might be based on the number of times the data has already been read since it was last written or on observations of the rate at which data in the entire workload is overwritten or deleted.

To focus on read performance, Figure 6-8 plots the average disk read time over the course of the same three benchmarks at 50% disk utilization. As suggested by the previous figure, both LFS and LFS with dynamic reorganization maintain a consistently low average read time for the benchmarks in which data is read as it is written. For random write followed by sequential read, both begin with a high average read time. This level of performance persists for LFS throughout the benchmark. For LFS with dynamic reorganization, reorganization traffic first increases read latency by forcing the disk head to alternate between writing to the reorganization log and reading

user data. However, once reorganization is complete, read latency drops to match the performance for those read patterns that matched the original write pattern. This performance improvement is enjoyed by all subsequent iterations of the same read pattern.

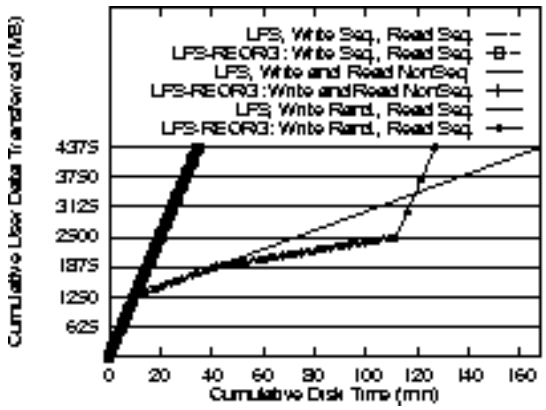

FIGURE 6-7. **LFS and LFS with Dynamic Reorganization at 50% disk utilization.** This figure shows the performance of LFS and LFS with dynamic reorganization for the three microbenchmarks discussed in Section 6.3. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB is a write of all the benchmark data in the pattern indicates by the benchmark. From 1250 to 4375 MB is five reads of all the benchmark data in the pattern indicated by the benchmark. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator.

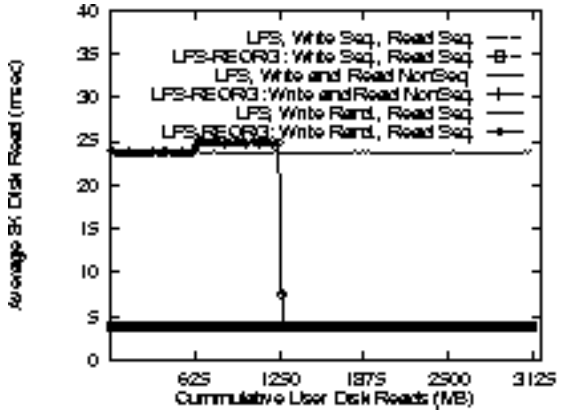

FIGURE 6-8. **Average Disk Read Time At 50% disk utilization.** This figure plots the average time for LFS with and without dynamic reorganization to read an 8KB block from disk for the three microbenchmarks discussed in Section 6.3. The total amount of user data read from disk by the benchmark is shown on the x-axis. The sequential read pattern repeats every 625 MB. The y-axis shows the average 8KB read time as reported by the Dartmouth disk simulator.

6.5.1.2. The Effect of Disk Utilization

Figure 6-9 and Figure 6-10 show the impact of higher disk utilization on dynamic reorganization. In Figure 6-9, we see that the overhead of dynamic reorganization is higher at higher disk utilization. At higher disk utilization, it is more difficult to produce the extents of free space required to rewrite the data into clusters. Reads to the reorganized data are also less efficient because it is more difficult to find adjacent free segments in which to keep subsequent pieces of the reorganization log together. This can be clearly seen in Figure 6-10 where reorganization does not achieve the same level of performance as with 50% disk utilization.

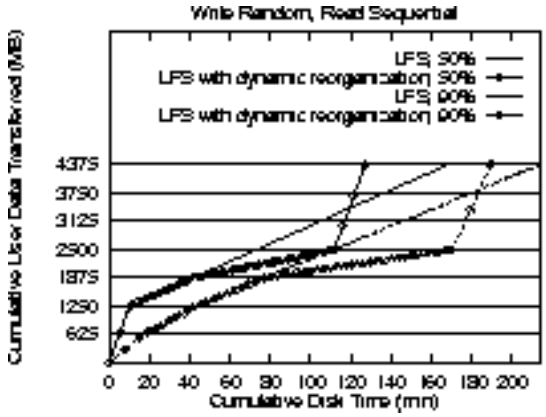

FIGURE 6-9. **The Effect of Disk Utilization on Dynamic Reorganization.** This figure shows the performance of LFS and LFS with dynamic reorganization for random write followed by sequential read at both 50% and 90% disk utilization. At 90% disk utilization, the disk is fragmented and the disk utilization increased before the benchmark begins by allocating 40% of the disk blocks to data not referenced again during the benchmark. The y-axis shows cumulative user data transferred by the benchmark. From 0 to 625 MB is a initial sequential write of all the benchmark data. From 625 to 1250 MB is a random write of all the benchmark data. From 1250 to 4375 MB is five sequential reads of all the benchmark data. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during LFS garbage collection, during the benchmark as reported by the Dartmouth disk simulator.

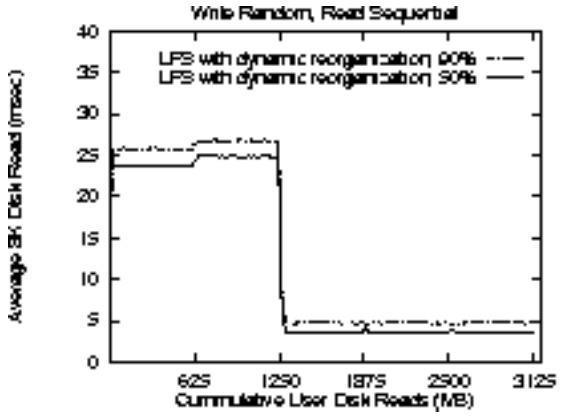

FIGURE 6-10. **The Effect of Disk Utilization on Average Disk Read Time.** This figure plots the average time to read an 8KB block from disk for random write followed by sequential read at both 50% and 90% disk utilization. At 90% disk utilization, the disk is fragmented and the disk utilization increased before the benchmark begins by allocating 40% of the disk blocks to data not referenced again during the benchmark. The total amount of user data read from disk by the benchmark is shown on the x-axis. The sequential read pattern repeats every 625 MB. Reorganization is completed at 1250 MB. The y-axis shows the average 8KB read time as reported by the Dartmouth disk simulator.

6.5.1.3. The Effect of Improving Disk Performance

Figure 6-11 examines the effect of improving disk performance on the behavior of dynamic reorganization by plotting performance for both the Dartmouth disk simulator and DiskSim. In the pre-reorganization phase of Figure 6-11, the difference in average read performance is dominated by the difference in seek and rotational delay. In the post-reorganization phase, the difference in performance is dominated by the difference in disk bandwidth. DiskSim represents a faster disk than the Dartmouth disk simulator; however, DiskSim has lower performance than most disks available at the time this dissertation was written. Since bandwidth is improving faster than latency, this gap is likely to increase over time. We would expect to see an even bigger relative performance gain for the faster modern disks.

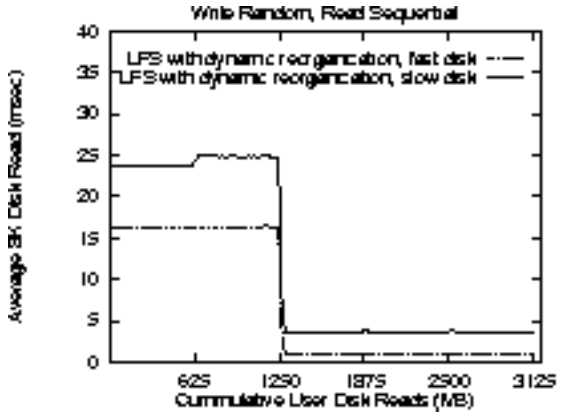

FIGURE 6-11. **The Effect of Improving Disk Performance on Average Disk Read Time.** This figure plots the average time to read an 8KB block from disk for a random write pattern followed by a sequential read. The total amount of user data read from disk by the benchmark is shown on the x-axis. The sequential read pattern repeats every 625 MB. Reorganization is completed at 1250 MB. The y-axis shows the average 8KB read time as reported by the Dartmouth disk simulator (slow disk) or DiskSim (fast disk).

6.5.2. Measured Traces

Up to this point, I have examined the performance of dynamic reorganization only on a select few synthetic traces. We have seen that dynamic reorganization improves performance for repeated read patterns. We have seen that it can both increase performance by achieving a layout better suited to actual read patterns and decrease performance with the reorganization traffic. We have also seen the impact of changes in disk utilization and disk performance characteristics. To examine the net effect of dynamic reorganization on more realistic workloads, I examine its performance on measured traces as well.

6.5.2.1. The Effect of a Long-term Trace

Long term traces are important because they can capture how read patterns change over time and how writes, deletes and garbage collection interact over time to produce the actual data layout. Roselli monitored fourteen desktop machines belonging to the graduate students, staff and faculty of a computer science research group for over one year [Roselli, 1998]. I examine a three month portion of this trace directed to a file system housing primarily executable files. In Figures 6-12 and 6-13, the performance of LFS with and without dynamic reorganization is compared for this trace at 90% disk utilization using the Dartmouth disk simulator. Dynamic reorganization improves overall disk performance by 12% and average read performance by 16%. Also, notice that the improvement in average read performance is larger towards the end of the trace where more information about past read patterns can be exploited.

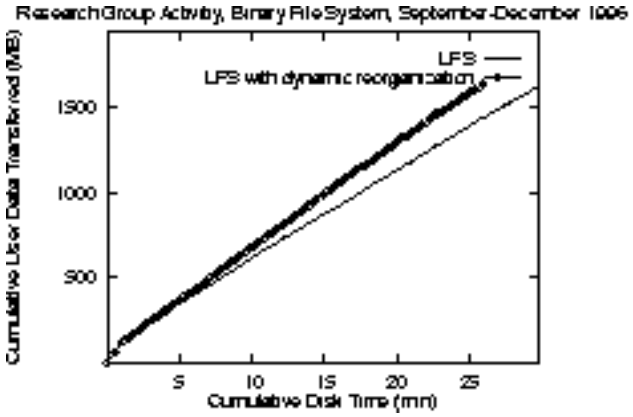

FIGURE 6-12. **LFS and LFS with Dynamic Reorganization at 90% disk utilization.** This figure shows the performance of LFS and LFS with dynamic reorganization for the research group file system activity. The x-axis shows cumulative disk times for all data transferred, including any additional data transferred during garbage collection, as reported by the Dartmouth disk simulator. The y-axis shows cumulative user data transferred by the benchmark.

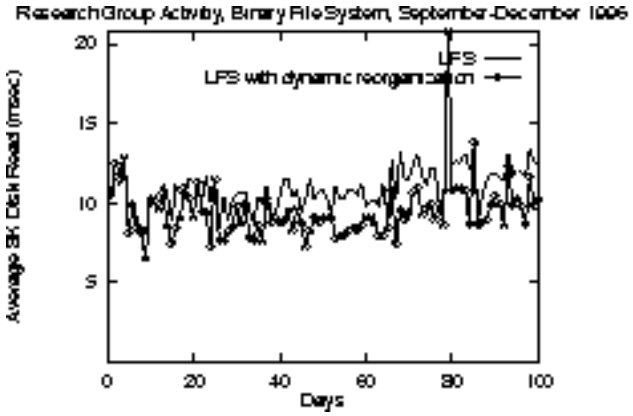

FIGURE 6-13. **Average Disk Read Time.** This figure plots the average time for LFS with and without dynamic reorganization to read an 8KB block from disk for the research group file system activity. The total amount of user data read from disk by the benchmark is shown on the x-axis. The y-axis shows the average 8KB read time as reported by the Dartmouth disk simulator.

6.5.2.2. The Effect of Changing Access Patterns

I also examine LFS and LFS with dynamic reorganization on a trace of the TPC-D decision support benchmark run on SQLServer [Matthews, 1999]. The 17 TPC-D queries are specifically designed to represent a wide variety of access patterns. For example, some tables are scanned sequentially in one query and accessed through their primary index in another. The temporary area is especially subject to changing access patterns. For example, a portion of it might be used to build an index needed at one point in the query; the index is then discarded and the same space reused to sort intermediate results. In this environment, there are many read and write accesses to the same data and there is often little time to accumulate useful historical information before the read patterns change. Therefore, this workload stresses the ability of dynamic reorganization to handle competing access patterns to the same data — both read and write access to the same data and multiple patterns for reading the same data.

In Figure 6-14, I show average read performance with 10 repetitions of the entire TPC-D workload. Overall, dynamic reorganization results in a 6.5% improvement in average read performance. In Figure 6-15, I show the average read performance with 10 repetitions of each query to isolate the impact of dynamic reorganization on individual queries. By analyzing the query plans produced by SQLServer for each query, I observe that queries which access tables through the indices, rather than with a sequential scan, benefit most from dynamic reorganization. This indicates that dynamic reorganization is clustering data in the tables together with their indices. Clustering based on the directory hierarchy would not capture this effect as each table is stored in a different file than the index on that table.

Figures 6-16 and 6-17 show the overall disk performance for both cases. For this workload, it would be necessary to perform the reorganization in idle time to improve overall disk performance.

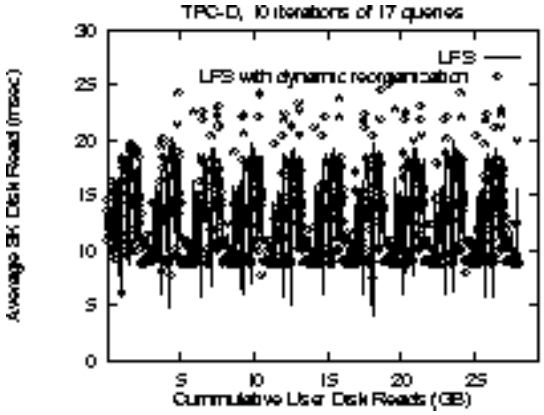

FIGURE 6-14. **Average Disk Read Time.** This figure plots the average time for LFS with and without dynamic reorganization to read an 8K block from disk for 10 iterations of all of the 17 TPC-D queries. The total amount of user data read from disk by the benchmark is shown on the x-axis. The y-axis shows the average 8k read time as reported by the Dartmouth disk simulator.

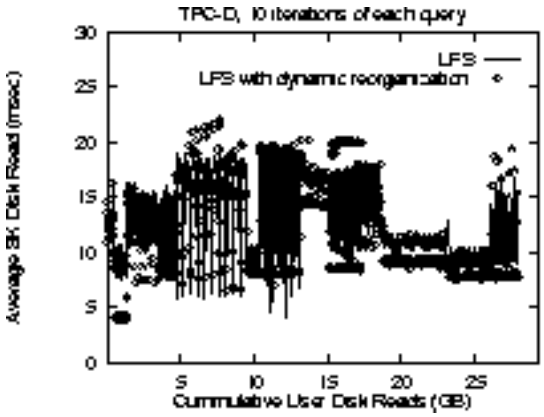

FIGURE 6-15. **Average Disk Read Time.** This figure plots the average time for LFS with and without dynamic reorganization to read an 8K block from disk for 10 iterations of each of the 17 TPC-D queries. The total amount of user data read from disk by the benchmark is shown on the x-axis. The y-axis shows the average 8k read time as reported by the Dartmouth disk simulator.

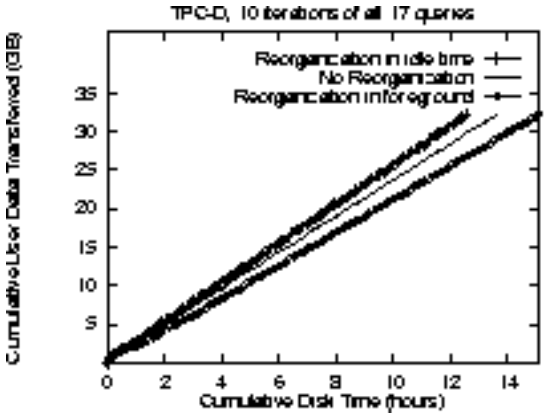

FIGURE 6-16. **LFS and LFS with Dynamic Reorganization at 90% disk utilization.** This figure shows the performance of LFS and LFS with dynamic reorganization for 10 iterations of all of the 17 TPC-D queries at 90% disk utilization. The y-axis shows cumulative user data transferred by the benchmark. The x-axis shows cumulative disk times for all data transferred (including any additional data transferred during garbage collection).

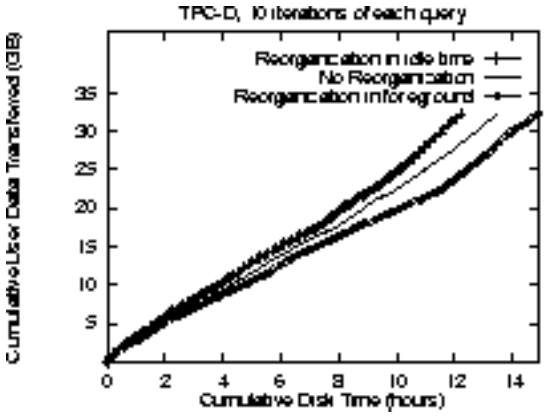

FIGURE 6-17. **LFS and LFS with Dynamic Reorganization at 90% disk utilization.** This figure shows the performance of LFS and LFS with dynamic reorganization for 10 iterations of each of the 17 TPC-D queries at 90% disk utilization. The y-axis shows cumulative user data transferred by the benchmark. The x-axis shows cumulative disk times for all data transferred (including any additional data transferred during garbage collection).

6.6. Related Work

Defragmenting tools have been in widespread use for some time especially for PC file systems [Diskeeper, 1999][Norton Utilities, 1999]. These tools reorganize the entire file system at once and require that they have exclusive access to the system while performing their changes. Therefore, they are not well suited to on-line data reorganization. In addition, most simply regroup individual files that have become scattered over time. Since most files are small, there is a limit to the benefit that can be achieved with this approach even when files are accessed sequentially in their entirety. The Intel Application Launch Accelerator included in Windows 98 actually profiles application execution and reorganizes data on disk to serialize disk blocks needed for the application's launch sequence and other predictable portions of the application's execution [Grimsrud, 1998] [Intel, 1998]. This reorganization is not limited to regrouping whole files and the designers have documented that non-sequential access patterns are actually quite common.

Some systems have shown a benefit to reorganizing the data on disk based on its frequency of access [Wong, 1983] [Musser and Schimke, 1992] [Ruemmler and Wilkes, 1991] [Vongsathorn and Carson, 1990] [Akyürek and Salem, 1995]. These systems move the most frequently accessed data to the center of the disk where it is most likely to be near the current disk head position at any given time. The next most frequently accessed data is moved to the cylinders adjacent to the center most cylinder and so on. This arrangement is referred to as an organ pipe and has been shown to be optimal for a biased random request stream [Wong, 1983]. Systems using this technique have reported up to 30% improvement, but it can also decrease performance when related data is located on opposite sides of the organ pipe. Staelin combined this approach with the whole-file grouping by migrating whole files to the center of the disk based on frequency of access [Staelin, 1991]. Similarly, this approach could be combined with dynamic reorganization by migrating frequently accessed partitions to the center of the disk, but I did not attempt this.

Geist et. al examine cylinder request streams and find a strong Markovian dependence [Geist et al., 1994]. They also advocate cylinder remapping as a technique for reducing mean seek distance in both single and mirrored disk systems. They present expressions to reflect mean seek distance for both single and mirrored disk systems. These expressions are used as energy functions in simulated annealing and geometric annealing optimization routines in order to find permutations that deliver reduced mean seek distances under stochastically equivalent workloads.

Identifying groups of objects that are accessed together in a persistent object base is much like identifying file blocks that are read together. Tsangaris and Naughton present a formal framework for evaluating the problem of optimal clustering [Tsangaris and Naughton, 1992]. Within this framework, they illustrate that although frequency-based clustering is ideal for a request stream that is as sequence of independent and identically distributed random variables [Wong, 1983], it is nonoptimal for workloads with locality. They define an optimal clustering to be one that minimizes the average working set of the problem as a formal optimization problem for determining the optimal object clustering in object based as a formal optimization problem and show the effectiveness of a clustering method based on weighted access graph partitioning.

Griffeon and Appleton dynamically building a graph in which files are nodes and edges represent subsequent accesses in order to guide prefetching [Griffioen and Appleton, 1994]. They prefetch files that follow the current file with a probability greater than a given threshold.

6.7. Conclusions

The layout policies of existing systems are unresponsive to actual read patterns and perform well only for read patterns that match the default policy. Especially in the presence of high disk utilization, fragmentation of free space and garbage collection complicate data layout and can degrade performance even for those read patterns that do match a system's default policy.

Dynamic reorganization builds a block access graph of past read patterns and uses that information to regroup related data. As write-optimized systems have found ways to make disk layout responsive to actual write patterns, dynamic reorganization provides a tool for making disk layout responsive to actual read patterns as well. In addition, it helps to find the right balance between read and write performance with adaptive methods that invest in reorganization only when the value of a new organization is demonstrated. However, it is questionable whether the benefit of dynamic reorganization on common case workloads is sufficient to justify the additional complexity and overhead of the adaptation.

7 Conclusions

In this final chapter, I summarize the contributions of this dissertation, propose directions for future work and reflect on how this work relates to the broader context of designing future computer systems.

7.1. Summary

In this dissertation, I propose and evaluate a variety of adaptive file system methods. I wanted to expose the problems of non-adaptive file systems and to demonstrate that relatively simple adaptive methods can produce systems that are significantly more robust than existing systems to changing hardware and diverse workload characteristics.

In pursuit of this goal, I make three main contributions to improving LFS write performance. First, I show how to choose the LFS segment size by trading transfer efficiency against cleaning efficiency. Second, I present an adaptive garbage collection algorithm that combines traditional LFS cleaning with hole-plugging. This algorithm dynamically adapts to changes in disk utilization and workload to avoid the traditional LFS performance cliff at high disk utilizations, while still preserving the advantage LFS has at lower disk utilizations. Third, I show how to further reduce cleaning overhead by taking advantage of cached segments when cleaning. Together, these enhancements make LFS more stable over a wider range of workloads by eliminating its dramatic worst case performance.

I also contribute to providing robust read performance. First, I demonstrate how non-adaptive layout policies fail to provide robust read performance when read patterns do not match the default layout or when the default layout is disturbed by fragmentation or garbage collection. Second, I present a dynamic reorganization algorithm that uses past read access patterns to predict future read patterns and groups data together on disk accordingly. Third, within the context of LFS, I show how dynamic reorganization can be used to augment a write-optimized system by suggesting targeted improvements to existing data layout.

The adaptive methods I evaluate vary in their complexity and in their demonstrated benefits. The relatively simple write-optimizations presented in Chapter 5 demonstrated up to a 10 fold increase in overall write performance relative to original LFS for the workloads I examined. The dynamic reorganization algorithm presented in Chapter 6 was more complex and demonstrated modest benefits (under 20%) for real workloads. In this light, I believe that the write optimizations I presented met the stated goal of simple, adaptive algorithms that provide high performance across a wider range of disk parameters and workloads. However, in my estimation, dynamic reorganization would require a simpler algorithm or a more dramatic benefits to warrant to its inclusion in a production file system.

7.2. Future Directions

There are several ways in which I think it would be interesting to extend this work. The first would be to perform additional experiments in the current simulation environment. The second would be to implement a fully functional file system based on these principles. The third would be to develop a formal model of the system in which proofs about its optimally or robustness could be made. Finally, it would be interesting to reexamine the role of the directory hierarchy in both the file system interface and in the organization of data on disk.

7.2.1. Additional Simulation Experiments

There are a huge number of design choices that could be studied and potentially made adaptive.

I would especially like to investigate simpler policies for read reorganization. For example, it would be interesting to see if the overhead of maintaining the access graph could be significantly reduced by assuming semantic access patterns and only recording non-sequential access patterns. One initial step would be to maintain a file rather than a block access graph.

In Chapter 6, I maintained a reorganization log in addition to the main write log. It would also be interesting to maintain different logs based on the source of the data and adapt the degree to which data is demultiplexed into separate logs. For example, separate logs could be maintained per client, per user or per application. Data from separate source could either be combined or isolated depending on the amount of traffic or certain characteristics like the frequency of data of data commit or the ratio of read to write traffic. In a system with multiple granularities of logging, it would be interesting to characterize how data flows between and the degree of read sharing among various logs.

In the current implementation of dynamic reorganization, I reorganize partitions as soon as they become eligible (i.e. the members are not already clustered together and the partition's value exceeds 2). In practice, it would certainly make sense to take advantage of idle time to accomplish reorganization. This brings up some interesting issues including how to detect when it would be globally optimal to reorganize immediately, even if some user activity is delayed in the short term, and how to prioritize the use of idle time between cleaning, reorganizing and other potential background activities [Golding et al., 1995].

It would be interesting to examine the relationship between dynamic reorganization and prefetching. Once reorganized, we can prefetch entire clusters. This may reduce the cache miss rate compared to whole file prefetching and automatic prefetching algorithm presented in [Griffioen and Appleton, 1994]. A logical next step would be to dynamically adjust the cluster size to balance transfer efficiency of larger clusters with the greater risk of prefetching unrelated data.

7.2.2. Adaptive File System Implementation

Another logical extension of this work would be the full implementation and deployment of an adaptive file system. Many researchers have noted the important synergy between implementation and simulation. They each contribute to complete view of the problem being examined — simulation by focusing on the essential elements of the problem and implementation by ensuring that all the complexities of the real world have been accounted for. Alone, each can have blind spots - simulation from modeling error and implementation from the presence of overheads that are not fundamental to the approach being studied. However, the man power required to proceed on both fronts is substantial.

In the earlier part of my graduate career, I had a taste of this synergy as a member of the xFS serverless network file system project [Anderson et al., 1996]. The xFS design was based on extensive simulations and the xFS team spent long hours implementing, debugging, configuring and testing a fairly fragile prototype. The implementation provided an important proof-of-concept, but simulation provided many of the important ideas— new caching algorithms, provably correct coherency algorithms and a sketch of the potential performance improvements which were not fully realized in the prototype.

7.2.3. Formal File System Models

The control theory community studies adaptivity from the point of view of formal mathematical models that can be used to express strong guarantees about a system's behavior. Some examples include proving the optimally of a given algorithm and proving that certain constraints on system behavior can always be satisfied. Control theory is concerned with feedback-based adaptation which is provably stable. At this time, the control theory community is actively engaged in modeling diverse and complex systems, like air traffic control systems and biological systems. I believe it would be interesting and fruitful to model storage system behavior in a similar way. From a control theory perspective, modeling new systems provides opportunities to apply existing results and may motivate new results. From an operating system perspective, an optimal solution even if it was not achievable in practice may inspire heuristic solutions and would provide an upper bound as a point of comparison. Some specific issues left open by this thesis include proving an upper bound on garbage collection costs and bounding the number of times a block may be shifted between competing disk organizations.

7.2.4. File System Organization

Dynamic reorganization builds a graph of data relationships. The directory hierarchy visible to the user is also a graph of data relationships. I can envision two possible directions for leveraging this overlap. First, it would be interesting to detect when the directory structure is adequate for capturing the dynamic access patterns and maintain additional access information only when the existing structure is not sufficient. Secondly and perhaps more naturally, it would be interesting to explore other ways to expose the data available in the file system to the user. From the user's perspective, the directory hierarchy is an unnecessarily restrictive way of naming and categorizing data; why should a user have to choose a single long stream of pathnames to express how a given file relates to all the other data in the file system? From the perspective of data location, I have

illustrated that it is more efficient to base storage organization on dynamic access patterns instead of on the directory hierarchy.

7.3. The Bigger Picture

7.3.1. Additional Benefits of Log-Structure

I have mentioned many benefits of log-structure in earlier sections of this dissertation. In this section, I discuss some less recognized benefits of log-structure that I think makes it particularly well-suited to be the underlying architecture for storage systems of the future.

First, log-structure is inherently flexible. Data can be located anywhere in the log. This flexibility allows it to easily incorporate desirable features from other systems where they are appropriate and to avoid the overhead when it is not needed. All but essential tasks can be deferred. This flexibility allows the system to provide high-burst performance by streamlining the critical path and deferring garbage collection and reorganization into idle time.

Second, log-structure is a natural way of accommodating the division of labor between the logical data management and the internal details of the underlying storage system. For example, viewing the file system as a collection of large data containers called segments is much easier to extend to complex storage systems than viewing it as a collection of cylinder groups.

Finally, I believe that the log-structure captures the fundamental characteristics of data within a storage system - newly written data flows through the levels of a storage hierarchy where it forms various pools for efficient storage. The most recent updates are grouped together allowing them to flow through the storage system, isolating them for special processing and protecting the more stable, organized data from their volatility. As these updates age, they are integrated more tightly into the rest of the system. New updates which began as a unit are separated to allow more

efficient storage and retrieval. This same structure exists whether the recent updates are staged in memory for transfer to disk, staged on a PDA for transfer to a connected host or staged at one site for backup to a redundant site.

Log-structure is also a natural way to collect data into distinct streams based on certain logical properties of the data. It is natural to imagine streams of data merging and diverging according to their source, their destination or their intended use. For example, traffic from multiple users and applications are channeled into a single log in order to amortize the costs of committing data to stable storage. This traffic may later be diverged in order to fill in the small areas of fragmented free space or diverged into groups of related data.

7.3.2. Adaptive Methods and Computer Systems of the Future

Computer scientists have successfully delivered tools of such power that they have revolutionized how we work, play and communicate. Computers are improving at an exponential rate according to almost every metric. Yet, many users find themselves saying "But I can't get it to do anything!". In many ways, it is a testimony to the power of the tool computer science has created that many of the key challenges today seem to be making all this power user friendly and maintainable. It is not surprising that many areas of computer science are turning to self-tuning, self-monitoring, self-configuring computer systems. We are beginning to turn the power of the tool inward to make itself more automatic, more usable and therefore more useful.

Building adaptive systems requires innovation on many fronts. It requires methods for observing workload and environmental characteristics. It requires models that predict the performance of alternatives under a variety of circumstances and that evaluate the trade-offs between alternate algorithms. It often requires a "common currency" into which various factors can be weighted. It

requires an analysis of when the benefits of adaptation outweigh the costs and when the adaptation algorithms are stable.

For example, computer architecture has been investigating general purpose versus reconfigurable versus customized processors. I would expect that same trend to move its way up through the building blocks of computer systems. For each layer in turn, optimizing for some preconceived common case will become unacceptable and the tug-of-war between customization and adaptation will need to be played out. In operating system design, optimization for the common case has been the norm. My opinion is that the future of operating systems lies in adaptive systems that can truly fulfill the promise of efficiently, reliably and robustly mediating the interaction between varied applications demands and rapidly changing hardware.

7.4. Conclusion

My thesis is that the systematic application of simple adaptive methods to file system design can produce systems that are significantly more robust to changing hardware and diverse workloads than existing systems. I present modifications to the log-structured file system that allow it to provide robust write performance in a wide range of environments. I also present a dynamic reorganization algorithm that makes disk layout responsive to actual read patterns. I evaluate these adaptive algorithms with trace driven simulation on a combination of synthetic and measured traces. I find that simple adaptive algorithms can dramatically improve worst case performance and can allow average case performance to scale with improvements in disk technology.

8 Bibliography

[Akyürek and Salem, 1995] S. Akyürek and K. Salem, "Adaptive Block Rearrangement," *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[Anderson et al., 1995] T. Anderson, M. Dahlin, J. Neefe, D. Patterson and R. Wang, "Serverless Network File Systems," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 109-126, December 1995.

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

[Arlitt and Williamson, 1996] M. Arlitt and C. Williamson, "Web Server Workload Characterization: The Search for Invariants," *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 126–137, May 1996.

[Arpaci-Duseeau et al., 1998] A. Arpaci-Dusseau, D. Culler, and A. Mainwaring, "Scheduling with Implicit Information in Distributed Systems," *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 233-243, June 1998.

[Baker et al., 1991] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a Distributed File System," *Proceedings of Thirteenth ACM Symposium on Operating Systems Principles*, pp. 198–212, December 1991.

[Barnard and Simon, 1993] S. Barnard and H. Simon, "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partioning Unstructured Problems," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 711-718, 1993.

[Berkeley Trace Repository, 1999] Berkeley File System Trace Repository, http://dawn7.berkeley.edu:8080.

[Birrell et al., 1993] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, "The Echo Distributed File System," Digital Equipment Corporation Systems Research Center Technical Report 111, September 1993.

[Blaze and Alonso, 1992] M. Blaze and R. Alonso, "Dynamic Hierarchical Caching in Large-scale Distributed File Systems," *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, June 1992.

[Blackwell et al., 1995] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proceedings of the 1995 Winter USENIX Technical Conference*, pp. 277-288, January 1995.

[Bolosky et al., 1996] W. Bolosky, J. Barrera, R. Draves, R. Fitzgerald and Givson, "The Tiger Video Fileserver," Microsoft Research Technical Report MSR-TR-96-09, April 1996.

[Bolosky et al., 1997] W. Bolosky, R. Fitzgerald and J. Douceur, "Distributed Schedule Management in the Tiger Video Fileserver," *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* , pp. 212-223, October 1997.

[Borowsky et al., 1997] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes, "Using Attribute-managed Storage to Achieve QoS," *Fifth International Workshop on Quality of Service*, June 1997.

[Cao et al., 1995] P. Cao, E. Felten, A. Karlin, and K. Li, "Implementation and Performance of Integrated Application-Controlled Caching, Prefetching, and Disk Scheduling," *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 188–197, May 1995.

[Chang and Gibson, 1999] F. Chang and G. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 1–14, February 1999.

[Chao et al., 1992] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes, "Mime: A High Performance Parallel Storage Device with Strong Recovery Guarantees," Hewlett-Packard Technical Report HPL-CSP-92-9, March 1992.

[Chen et al., 1994] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, 26(2):145-188, June 1994.

[Chen et al., 1996] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 74–83, October 1996.

[BSD4.4-Lite Source, 1994] Computer Systems Research Group, University of California at Berkeley, 4.4BSD-Lite Source CDROM, USENIX Association and O'Reilly & Associates, 1994.

[Chutani et al., 1992] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, and R. Siedbotham, "The Episode File System," *Proceedings of the 1992 Winter USENIX Technical Conference*, pp. 43–60, January 1992.

[Clark, 1992] R.N. Clark, *Introduction to Automatic Control Systems*, Wiley, 1992.

[Custer, 1994] H. Custer, *Inside the Windows NT File System*, Microsoft Press, 1994.

[Dahlin et al., 1994a] b Dahlin, Michael, Randy Wang, Thomas Anderson, and David Patterson, "A Quantitative Analysis of Cache Policies for Scalable Network File Systems," *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 150–160, May 1994.

[Dahlin et al., 1994b] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative Caching: Using Remote Memory to Improve File System Performance," *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 267-280, November 1994.

[Dahlin, 1996] M. Dahlin, "Serverless Network File Systems," University of California at Berkeley Dissertation, Technical Report UCB/CSD-96-900, March 1996.

[Compaq Digital, 1999] Compaq DIGITAL FX!32, "White Paper: How DIGITAL FX!32 works," http://www.digital.com/amt/fsw32/fx-white.html, 1999.

[Diskeeper, 1999] Diskkeeper Windows NT Defragmentation Utility. http://www.defragmentation-tools.com, 1999.

[DISK/TREND, 1998] DISK/TREND, Inc., "1998 DISK/TREND Report," 1925 Landings Drive, Mountain View, CA 94043, http://www.disktrend.com.

[DISK/TREND, 1999] DISK/TREND, Inc., "1999 DISK/TREND Report," 1925 Landings Drive, Mountain View, CA 94043, http://www.disktrend.com.

[EMC, 1999] The Enterprise Storage Company, http://www.emc.com/.

[English and Stepanov, 1992] R. English and A. Stepanov, "Loge: a Self-Organizing Disk Controller," *Proceedings of the 1992 Winter USENIX Technical Conference*, pp. 237–251, January 1992.

[Feeley et al., 1995] M. Feeley, W. Morgan, F. Pighim, A. Karlin, H. Levy, and C. Thekkath, "Implementing Global Memory Management in a Workstation Cluster," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 201-212, December 1995.

[Feiertag and Organick, 1971] R. Feiertag and E. Organick, "The Multics Input-Output System," *Proceedings of the Third ACM Symposium on Operating Systems Principles*, October 1971.

[Franklin et al., 1994] G. F. Franklin, J. D. Powell and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 3rd ed., Addison-Wesley Publishing Company, 1994.

[Ganger and Kaashoek, 1997] G. Ganger and F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," *Proceedings of the 1997 Annual USENIX Technical Conference*, pp. 1–17, January 1997.

[Ganger et al., 1998] G. Ganger, B. Worthington, and Y. Patt, "The DiskSim Simulation Environment Version 1.0 Reference Manual," University of Michigan Technical Report CSE-TR-358-98, February 1998.

[Ghormley et al., 1998] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson, "SLIC: An Extensibility System for Commodity, Operating Systems," *Proceedings of the 1998 Annual USENIX Technical Conference*, June 1998.

[Gibson, 1992] G. Gibson, "Redundant Disk Arrays: Reliable, Parallel Secondary Storage," *ACM Distinguished Dissertations*, MIT Press, Cambridge Massachusettes.

[Geist et al., 1994] R. Geist, D. Suggs, and R. Reynolds, "Minimizing Mean Seek Distance in Mirrored Disk Systems by Cylinder Remapping", *Performance Evaluation*, 20:97-114, 1994.

[Golding et al., 1995] R. Golding, P. Bosch, C. Staelin, T. Sullivan and J. Wilkes, "Idleness Is Not Sloth," *Proceedings of the 1995 Winter USENIX Technical Conference*, January 1995.

[Gray, 1981] J. Gray, "The Transaction Concept: Virtues and Limitations," *Proceedings of the Seventh International Conference on Very Large Data Bases*, pp. 144-154, September 1981.

[Gribble et al., 1998] S. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller, "Self-Similarity in File Systems," *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 141–150, June 1998.

[Griffioen and Appleton, 1994] J. Griffioen and R. Appleton, "Reducing File System Latency using a Predictive Approach," *Proceedings of the 1994 Summer USENIX Technical Conference*, pp. 197–207, June 1994.

[Grimsrud, 1998] K. Grimsrud, "Method And Apparatus For Improving Disk Drive Performance," United States Patent 5802593, September 1998.

[Grochowski, 1996] E. Grochowski, "IBM Leadership In Disk Drive Technology," http://www.storage.ibm.com/storage/technolo/grochows/grocho01.htm, 1996.

[Grochowski, 1998] E. Grochowski, "Magnetic Hard Disk Drives—Advances Through The Year 2000 And Beyond," Public Seminar at University of California at Berkeley, October 29, 1998.

[Hagmann, 1987] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 155–162, October 1987.

[Hartman and Ousterhout, 1995] J. Hartman and J. Ousterhout, "The Zebra Striped Network File System," *ACM Transactions on Computer Systems*, 13(3):279-310, August 1995.

[Harvest, 1998] The Harvest Object Cache. http://harvest.transarc.com/, 1998.

[Hendrickson and Leland, 1995] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Proceedings of SUPERCOMPUTING '95*, 1995.

[Hennessy and Patterson, 1996] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[Hitz et al., 1995] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," Network Appliance Technical Report TR3002, March 1995.

[Howard et al., 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1):51-81, February, 1988.

[Intel, 1998] Intel Corporation, Intel Application Launch Accelerator, http://developer.intel.com/ial/ala/, 1998.

[Jacobson and Wilkes, 1991] D. Jacobson and J. Wilkes, "Disk Scheduling Algorithms Based On Rotational Position," Hewlett-Packard Technical Report, HPL-CSP-91-7rev1, March 1991.

[Jacobson and Karels, 1988] V. Jacobson and M. Karels, "Congestion Avoidance and Control," *Proceedings of the SIGCOMM Conference on Data Communication*, November 1988.

[Java, 1999] Sun Microsystems, "The Java HotSport[tm] Performance Engine Architecture," http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[Karedla et al., 1994] R. Karedla, J. S. Love, and B. Wherry, "Caching Strategies to Improve Disk System Performance," *Computer Magazine*, pp. 38–46, March 1994.

[Kotz et al., 1994] D. Kotz, S. Toh, and S. Radhakrishnan, "A Detailed Simulation Model of the HP 97560 Disk Drive," Dartmouth Technical Report PCS-TR94-220, July 1994.

[Kowalski, 1978] T. Kowalski, "FSCK: The UNIX System Check Program," Bell Laboratory, March 1978.

[Kroeger and Long, 1994] T. Kroeger and D. Long, "Predicting File System Actions from Prior Events," *Proceedings of the 1996 Annual USENIX Technical Conference*, June 1996.

[Lee and Thekkath, 1996] E. Lee and C.Thekkath, "Petal: Distributed Virtual Disks," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 84–92, October 1996.

[Lei and Duchamp, 1997] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," *Proceedings of the 1997 Annual USENIX Technical Conference*, pp. 272–288, January 1997.

[Lomet, 1995] D. Lomet, "The Case for Log-structuring in Database Systems," *International Workshop on High Performance Transaction Systems*, September 1995.

[Mackert and Lohman, 1986] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pp. 149-158, August 1986.

[Mashey, 1997] J. Mashey, "Big Data... and the Next Wave of InfraStress," Technical presentation given at University of California at Berkeley, 1997.

[Matthews et al., 1997] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson, "Improving the Performance of Log-Structured File Systems with Adaptive Methods," *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pp. 238–251, December 1997.

[Matthews, 1999] J. Matthews, "Reorganizing Disk Layout to Improve the Read Performance of the TPC-D Benchmark on a Log-Structured File System," http://www.cs.berkeley.edu/~neefe/papers/DBPROJ.ps, February 1999.

[McNutt, 1994] B. McNutt, "Background Data Movement in a Log-Structured File System," *IBM Journal of Research and Development*, 38(1):47-58, 1994.

[McKusick, 1999] M. McKusick, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *FREENIX Track Proceedings of the 1999 Annual USENIX Technical Conference*, pp. 1–17, June 1999.

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

[McVoy and Kleiman, 1991] L. McVoy and S. Kleiman, "Extent-like Performance from a UNIX File System," *Proceedings of the 1991 Winter USENIX Technical Conference*, pp. 33-43, January 1991.

[Mohan et al., 1992] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions of Database Systems*, 17(1):94-162, 1992.

[Mummert and Satyanarayanan, 1994] L. Mummert and M. Satyanarayanan, "Long Term Distributed File Reference Tracing: Implementation and Experience," Carnegie-Mellon Technical Report CMU-CS-94-213, November 1994.

[Musser and Schimke, 1992] D. Musser and N. Schimke, "Block Shuffling for Improved Read Performance in a Loge Disk Controller," Rensselaer Polytechnic Institute Technical Report, August 1992.

[Network Appliance, 1997a] Network Appliance, "NetCache Software Advantage," http://www.networkappliance.com/products/level3/netcache/webcache.html.

[Network Appliance, 1997b] Network Appliance, "Caching Tutorial for Web Authors and Webmasters," http://www.networkappliance.com/products/level3/netcache/cache_basics.html.

[Norton Utilities, 1999] Norton Utilities, http://www.symantec.com/nu/fs_nunt.html, 1999.

[Ousterhout et al., 1985] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *ACM Tenth Symposium on Operating Systems Principles*, pp. 15–24, December 1985.

[Ousterhout, 1995a] J. Ousterhout, "A Critque of Seltzer's 1993 USENIX Paper," http://www.sunlabs.com/people/john.ousterhout/seltzer93.html, 1995.

[Ousterhout, 1995b] J. Ousterhout, "A Critque of Seltzer's LFS Measurements," http://www.sunlabs.com/people/john.ousterhout/seltzer.html, 1995.

[Ousterhout,1995c] J. Ousterhout, "A Response To Seltzer's Response," http://www.sun-labs.com/people/john.ousterhout/seltzer2.html, 1995.

[Papadopolous, 1997] G. Papadopolous, "The Future of Computing," Unpublished talk at Network of Workstations workshop, Lake Tahoe, CA, July 27, 1997.

[Patterson, 1998] D. Patterson, "Hardware Technology Trends and Database Opportunities," Keynote Address at the 1998 ACM SIGMOD Conference, Slides available at http://www.cs.berkeley.edu/~pattrsn/talks.html, 1998.

[Patterson, 1995] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proceedings of the 15th Symposium on Operating System Principles*, pp. 79–95, December 1995.

[Quantum, 1999a] Quantum Corporation. "Firsts in Disk Drive Technology," http://www.quantum.com/src/tt/sh_firsts.htm, 1999.

[Quantum, 1999b] Quantum Corporation, "Storage Industry History and Trends," http://online1.quantum.com/src/tt/storage_history.htm, 1999.

[Riedel and Gibson, 1996] E. Riedel and G. Gibson, "Understanding Customer Dissatisfaction with Underutilized Distributed File Servers," *Proceedings of the 5th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, September 1996.

[Ritchie and Thompson, 1974] D. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, 17(7), July 1974.

[Roselli, 1998] D. Roselli, "Characteristics of File System Workloads," University of California at Berkeley Technical Report UCB//CSD-98-1029, December 1998.

[Roselli et al., 1999] D. Roselli, J.N. Matthews and T. Anderson, "Improving File System Disk Read Performance," submitted for publication, May 1999.

[Rosenblum and Ousterhout, 1992] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System for UNIX," *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[Rosenblum, 1992] M. Rosenblum, "The Design and Implementation of a Log-Structured File System," University of California, Berkeley Technical Report, UCB-CSD-92-696, June 1992.

[Ruemmler and Wilkes, 1991] C. Ruemmler and J. Wilkes, "Disk Shuffling," Hewlett-Packard Technical Report HPL-91-156, October 1991.

[Ruemmler and Wilkes, 1993] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proceedings of 1993 Winter USENIX Technical Conference*, CA, January 1993.

[Russinovich and Cogswell, 1997] M. Russinovich and B. Cogswell, "Examining the Windows NT File System," *Dr. Dobb's Journal*, February 1997.

[Schwaderer and Wilson, 1996] W. D. Schwaderer and A. Wilson, *Understanding I/O Subsystems*, Adaptec Press, Milpitas, CA, 1996.

[Seagate, 1998] Seagate Technology, Inc., "Specifications for ST-118202," http://www.seagate.com.

[Selinger et al., 1979] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the 1979 ACM SIGMOD Conference*, pp. 23-34, 1979.

[Seltzer et al., 1990] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisted", *Proceedings of the 1990 Winter USENIX Technical Conference*, January 1990.

[Seltzer et al., 1993] M. Seltzer, K. Bostic, M. McKusick and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the 1993 Winter USENIX Technical Conference*, January 1993.

[Seltzer et al., 1995] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File System Logging Versus Clustering: A Performance Comparison," *Proceedings of the 1995 Winter USENIX Technical Conference*, January 1995.

[Seltzer and Smith, 1995] M. Seltzer and K. Smith, "A Response to Ousterhout's Critique of LFS Measurements," http://www.das.harvard.edu/users/faculty/Margo_Seltzer/usenix.195/ouster.html, 1995.

[Smith and Seltzer, 1994] K. Smith and M. Seltzer, "File Layout and File System Performance," Harvard Technical Report TR-35-94, 1994.

[Smith and Seltzer, 1996] K. Smith and M. Seltzer, "A Comparison of FFS Disk Allocation Policies," *Proceedings of the 1996 Annual USENIX Technical Conference*, January 1996.

[Smith and Seltzer, 1997] K. Smith and M. Seltzer, "File System Aging—Increasing the Relevance of File System Benchmarks," *Proceedings of the 1997 ACM Conference on Measurement and Modeling of Computer Systems*, pp. 203–213, June 1997.

[Smith, 1999] K. Smith, Personal Communication, April 1999.

[Staelin, 1991] C. Staelin, "High Performance File System Design," Princeton University Dissertation, Technical Report TR-347-91, October 1991.

[SPEC, 1992] Standard Performance Evaluation Corporation, "The SPEC CPU92 Benchmarks," http://www/spec.org/osg/cpu92.

[SPEC, 1995] Standard Performance Evaluation Corporation, "The SPEC CPU95 Benchmarks," http://www/spec.org/osg/cpu95.

[Squid, 1998] The Squid Internet Object Cache, http://squid.nlanr.net/Squid/, 1998.

[Sweeney et al., 1996] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Proceedings of the 1996 Annual USENIX Technical Conference*, January, 1996.

[Thekkath et al., 1997] C. Thekkath, T. Man, and E. Lee, "Frangipani: A Scalable Distributed File System," *Proceedings of the 16th Symposium on Operating Systems Principles*, pp. 224–237, December 1997.

[Talagala et al., 1999] N. Talagala, R. Arpaci-Dusseau and D. Patterson, "Microbenchmark-Based Extraction of Local and Global Disk Drive Parameters", Computer Science Division, University of California, Berkeley, Submitted for publication, July 1999.

[Thompson, 1978] K. Thompson, "UNIX Implementation," Bell Systems Technical Journal, 57, 6, part 2, August 1978.

[TPC-B, 1990] Transaction Processing Performance Council, "TPC Benchmark B Standard Specification," August 1990.

[TPC-C, 1990] Transaction Processing Performance Council, "TPC Benchmark C Standard Specification," July 1990.

[TPC-D, 1995] Transaction Processing Performance Council, "TPC Benchmark D Standard Specification," April 1995.

[Tsangaris and Naughton, 1991] M. Tsangaris and J. Naughton, "A Stochastic Approach for Clustering in Object Bases," *Proceedings of the 1991 ACM SIGMOD Conference*, pp. 12-21, May 1991.

[Tsangaris and Naughton, 1992] M. Tsangaris and J. Naughton, "On the Performance of Object Clustering Techniques," *Proceedings of the 1992 ACM SIGMOD Conference*, pp. 144-153, 1992.

[Veritas, 1995] Veritas Software, "The VERITAS File System (VxFS)," http://www.veritas.com/products.html, 1995.

[Villasenor and Mangione-Smith, 1997] J. Villasenor and W. Mangione-Smith. "Configurable Computing", *Scientific American*, June 1997.

[Vongsathorn and Carson, 1990] P. Vongsathorn and S. Carson, "A System for Adaptive Disk Rearrangement," *Software: Practice and Experience*, vol. 20, no. 3, pp. 225–242, March 1990.

[Wang et al., 1999] R. Wang, T. Anderson, and D. Patterson, "Virtual Log Based File Systems for a Programmable Disk, "*Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 29–43, February 1999.

[Wilkes et al., 1995] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 96–108, December 1995.

[Wittle and Keith, 1993] M. Wittle and B. Keith, "LADDIS: The Next Generation in NFS File Server Benchmarking," *Proceedings of the 1993 Summer USENIX Technical Conferene*, pp. 111–128, June 1993.

[Worthington et al., 1995] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. "On-line Extraction of SCSI Disk Drive Parameters", *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1995.

[Wong, 1983] C. Wong, " Algorithmic Studies in Mass Storage Systems," *Computer Science Press*, 1983.