Trace-Based Analyses and Optimizations for Network Storage Servers

Daniel Joseph Ellard

TR-11-04

Computer Science Group
Harvard University
Cambridge, Massachusetts

Trace-Based Analyses and Optimizations for Network Storage Servers

A thesis presented

by

Daniel Joseph Ellard

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May, 2004

Advisor: Margo I. Seltzer Author: Daniel Joseph Ellard

Trace-Based Analyses and Optimizations for Network Storage Servers

Abstract

In this thesis, I show how network storage servers can infer useful information about the requests they are likely to see in the future by analyzing the history of requests they have observed in the past. I also show that this information can be used to improve future decisions about disk block allocation and read-ahead and thereby increase network storage server performance without any change to its clients or the applications running on its clients.

The contributions of this thesis are:

- A new suite of utilities for gathering and analyzing NFS traces.
- Long-term traces of three contemporary workloads.
- An analysis of these workloads, showing that they differ in important ways. These differences may offer an opportunity for workload-specific optimizations.
- Optimizations to the NFS read-ahead heuristic for sequential and non-sequential access patterns that have sequential subcomponents. For the latter type of access pattern, end-to-end read performance can be improved by more than 50%.
- The discovery that for most files, the create-time attributes of the file (such as its name) are strongly associated with the operations that will be performed on the file. These associations can be discovered automatically by analysis of the network traffic between an NFS server and its clients.
- A demonstration that these associations can be used to automatically construct models that accurately predict properties such as the eventual size, lifespan, and access patterns of new files.
- A demonstration that these models can be used to improve on-disk locality of reference by using predictions to enhance the file and directory layout heuristics and thereby arrange for the hottest blocks on the disk to be grouped together.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis is dedicated to my family. No part of my graduate studies would have been possible without their unflagging support and sacrifices.

I must also thank my thesis committee: Margo Seltzer, Michael Rabin, Jim Waldo, and Matt Welsh. I am forever indebted to them for the guidance and patience.

This work would have not been possible without assistance from the system administrators who let me gather traces from systems under their care: Peg Schaefer, Aaron Mandell (of EECS), Lars Kellog-Steadman, Chris Palmer (of DEAS), Bill Ouchark, Alan Sundell, and Scott McGrath (of HASCS).

I must also thank the many people who provided good advice or helpful information (on any of a number of topics) including my parents Tim and Pat Ellard, Jim Waldo, Brian Kernighan, Gregory Ganger, Gene McAfee, Marshall Poe, Pat Peterson, Jeff Berliner, Keith Smith, Chris Small, Kostas Magoutis, and David Holland.

Many parts of this thesis are based on papers that I have written or helped to write during the last three years. I would like to thank Remzi Arpaci-Dusseau, Chuck Lever, and Mario Obejas, who in their role as shepherds for my accepted papers helped to improve the quality of the exposition of my ideas. For the other papers, I thank the anonymous reviewers and program committee members for their remarks and for their uncompromisingly high standards.

I must also thank my co-authors, fellow graduate students and collaborators: Pia Malkani and Jonathan Ledlie, who helped with the initial analysis of the EECS and CAMPUS traces; Eno Thereska and Michael Mesnier, who taught better ways to build predictive models; and Ningning Zhu, who helped debug my benchmarking system and brain-stormed about related issues as she developed her own benchmarking system.

Finally, I thank the sponsors of my graduate work, which include IBM, the National Science Foundation (NSF grant DUE-9950239), the Advanced Computing Systems Association, (USENIX Scholarship, 1999), and the Courtland S. Gross Memorial Foundation (Scholarship for Academic Excellence, 1997).

To my family.

Chapter 1

Introduction

In this thesis, I show how network storage servers can infer useful information about the requests they are likely to see in the future by analyzing the history of requests they have served in the past. I also show that this information can be used to improve future decisions about disk block allocation and read-ahead and thereby increase network storage server performance without any changes to the client or the applications running on the client.

1.1 Motivation

Storage system performance is becoming the bottleneck for many applications. There are two reasons for this: first, the speed at which storage system performance has improved is slower than that of the other components of modern computer systems, and second, modern applications demand ever-increasing volumes of data.

In the long run, the performance of a system is determined by the performance of its slowest component. In a field where constant improvement seems to be normal, the component that improves least quickly will soon become the bottleneck. In modern computer systems this component is the disk. Although disk performance is improving, it is falling behind the other components. Historically, CPU speeds and memory densities have doubled every 18-24 months for the last two decades. Local-area network speeds increase in a more stepwise fashion (because how fast they run is specified as part of their protocol, so changing a network speed requires building a new network infrastructure rather than simply increasing the clock rate), but have steadily increased in bandwidth by at least a factor of 10 every decade, going from half-duplex shared access 10Mb links in the early 80's to the deployment of full-duplex fully-switched gigabit networks common today and 10-gigabit networks poised for deployment in the next few years. Wide-area networks have made even better progress, making it practical for hundreds or thousands of clients to access data on the other side of the world at bandwidth rates that a decade ago would have been considered excellent for local area networks.[1]

In the area of storage, most of the progress has been in the areas of density and cost. The ratio of capacity to cost has grown at a rate that is astonishing, even compared to CPU and memory. Unfortunately, storage system performance (and in particular disk performance) has increased much more slowly – typical disk latency has improved by only approximately a factor of two over the last decade.

Meanwhile, the amount of data demanded by modern applications is growing exponentially as well. The availability of inexpensive and capacious on-line storage has enabled new applications that were once thought intractable.

[1] Latency has also become an issue for networking, particularly wide-area networks. Network latency is ultimately bounded by the speed of light in a vacuum. This limit is not a problem for applications such as remote storage access if the distance the data must travel is less than several hundred meters because the latency created by this distance is miniscule compared with the latency added by other parts of the system. For access to data on other sides of the world, however, network latency is a serious issue.

Unfortunately, our ability to access this data is not growing at the same rate as our demand for it. Although disk bandwidth has improved at a steady pace (by at least a factor of ten during the past decade) the achievable speed with which data on a disk can be accessed has improved more slowly.

Barring the emergence of a new technology that replaces disks and has remarkably better performance characteristics, these problems will only get worse as storage system performance falls farther and farther behind the performance of the rest of the system.

One way to attack these problems is to increase the ability of the storage servers to adapt their behavior so that they may make the best possible use of their resources. The file and storage system community has developed many techniques to improve the performance of I/O-bound applications. The problem is that many of these techniques do not scale well (or work at all) in the context of network storage servers because they are designed to work well for a single workload or while being driven by a single application rather than being shared among hundreds or thousands of concurrent applications.

1.2 Research Agenda

In this thesis, I explore the idea of increasing storage server performance by making storage servers more intelligent. In particular, I explore ways to make storage systems adaptive. Such systems are able to examine their workload, learn what characteristics are important, anticipate what the applications are likely to do in the future, and optimize their behaviors in response.

The essential question I address is whether storage systems can recognize and autonomously adapt to their workloads in a manner that improves the performance they provide their clients.

Some aspects of an idealized adaptive storage server are shown in Figure 1.1. This system is able observe its workload and keep a history of the requests it has seen. This history is used to create models of the workload that can be used to predict what requests or patterns of requests the clients are likely to make in the future. These predictions are used to tune or override the default file system policies and implementation strategies for issues such as caching, file and directory layout, and read-ahead.

Note that my goal is *not* simply to improve system performance by any means necessary. I focus exclusively on transparent changes to the server and do not change the behavior of the clients or the interface that the clients use to communicate with the server. For some of the problems I address, the root of the problem is that a client application uses the storage system in an inefficient or unanticipated manner. In these cases, it is arguable that the most practical way to improve client performance would be to rewrite the application to make better use of the storage system. This solves the problem for only one application (and one storage system) – it does not prevent another application from making the same mistake, nor does it ensure that the application will continue to perform well if the storage system is modified at some time in the future. The goal of my work is to create servers that perform well and can adapt to any application running on their clients, not just well-written ones.

1.3 Contributions

The contributions of this thesis are:

- A new suite of utilities for gathering and analyzing NFS traces. This suite is available in open-source form.[2]
- An analysis of several contemporary NFS workloads, showing that workloads vary in important ways. These variations may offer an opportunity for workload-specific optimizations.

[2] http://www.eecs.harvard.edu/sos/software/

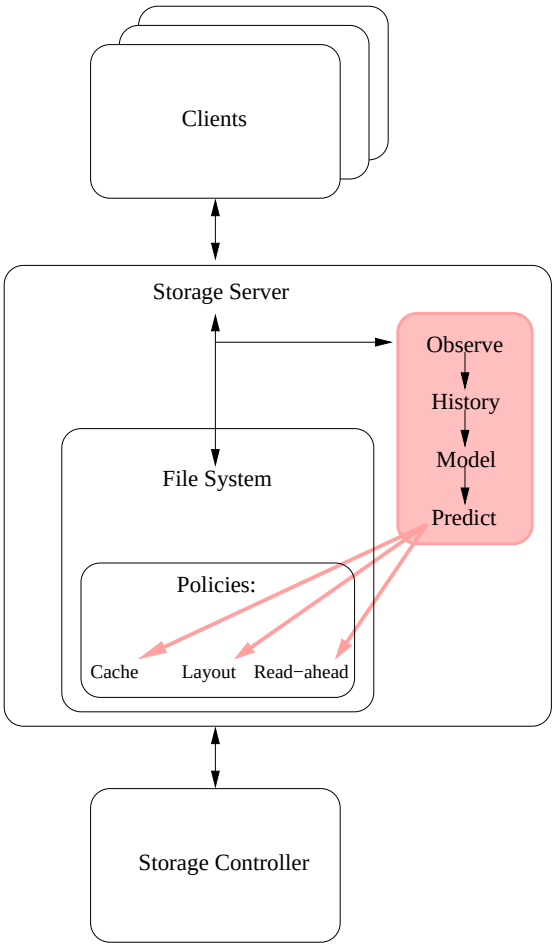


Figure 1.1: An adaptive network storage server.

- Optimizations to the NFS read-ahead heuristic for sequential and non-sequential access patterns with sequential subcomponents. For the latter type of access pattern, end-to-end read performance can be improved by more than 50%.
- The discovery that for most files, the create-time attributes of the file (such as the name of the file) are strongly associated with the operations that will be performed on that file. These associations can be discovered automatically by analysis of the network traffic between an NFS server and its clients.
- A demonstration that these associations can be used to automatically construct models that accurately predict properties such as the eventual size, lifespan, and access patterns of new files.
- A demonstration that these models can be used to improve on-disk locality of reference by using predictions to enhance the file and directory layout heuristics and thereby arrange for the hottest blocks on the disk to be grouped together.

The most general contribution of this thesis is a new set of long-term NFS traces from several different workloads, including research, academic, and mail workloads. I have made these traces available to the research commu-

nity[3] and this body of data is, in and of itself, an important contribution to the field. These data have allowed researchers at several institutions to extend their research and test their theories against new, contemporary traces. Current users of this work include IBM Research (Almaden), Carnegie Mellon University, the University of California at Berkeley, the University of California at Los Angeles, the University of Connecticut, the University of Massachusetts, the University of Michigan at Ann Arbor, the University of Illinois at Urbana-Champaign, the University of Texas at Austin, the University of Pittsburgh, Florida State University, and the State University of New York at Stony Brook. Although these traces have only been available for a relatively short time, they have already been used by researchers investigating issues as divergent as basic file system design [39, 92], comparing different network storage paradigms [63], storage-based intrusion detection [61], the efficiency of versioning systems [79], and new benchmarking techniques [99].

1.4 Dissertation Overview

Chapter 2: In this chapter I introduce the important components of modern networked storage systems and the other issues related to their performance. I also include a brief synopsis of some recent and current schools of thought on how to optimize storage access, including hints and gray-box techniques.

Chapter 3: In this chapter I discuss the challenges of capturing and analyzing traces from contemporary workloads and describe the suite of tools I built to facilitate the capture and analysis of NFS workloads.

This chapter also discusses how traces can be transformed into workload-specific benchmarks.

Chapter 4: I give a brief description of several contemporary NFS workloads in this chapter. I also illustrate, via case studies, how the tools described in Chapter 3 can simplify several analyses of interest to file system researchers and system administrators.

Chapter 5: I explore two new techniques to improve end-to-end read performance by improving the read-ahead heuristics used by the FreeBSD 4.X NFS server. The first of these techniques, called *SlowDown*, is not ultimately successful in improving the performance, but its investigation is enlightening because it exposes many interesting quirks that must be taken into account when benchmarking a system such as NFS. This investigation also reveals a bug in the FreeBSD NFS server implementation, and I show that fixing this bug can increase the performance of heavily loaded servers by 20% for read-dominated workloads.

The second of these techniques, called *Cursor*, is a extension of *SlowDown* that is able to detect sequential subcomponents embedded within access patterns that would otherwise be classified as random. This increases the end-to-end read performance by over 50% for some access patterns.

Chapter 6: In this chapter I show that file names are associated with a number of useful file properties, including expected lifespan, size, and access patterns. I show that these associations can be discovered automatically and used to generate accurate models to classify new files.

Chapter 7: In this chapter I present an experiment that shows how name-based predictions can be used to increase the locality of reference for the disk implementing an NFS file system. I use the workloads described in Chapter 4 to drive the experiment and the benchmark technique described in Chapter 3.

Chapter 8: In this chapter I discuss related work.

Chapter 9: In the final chapter I conclude by summarizing my findings, reflecting on the results, and discussing future work.

[3] http://www.eecs.harvard.edu/sos/traces.html

Chapter 2

An Introduction to Data Storage

To understand the root of the performance challenges faced by modern network storage systems, it is necessary to understand the basic properties and limitations of current storage architectures and the components used to implement these architectures. It is also useful to understand some of the techniques used to overcome these limitations because these techniques form the basis for the new methods described later in this thesis.

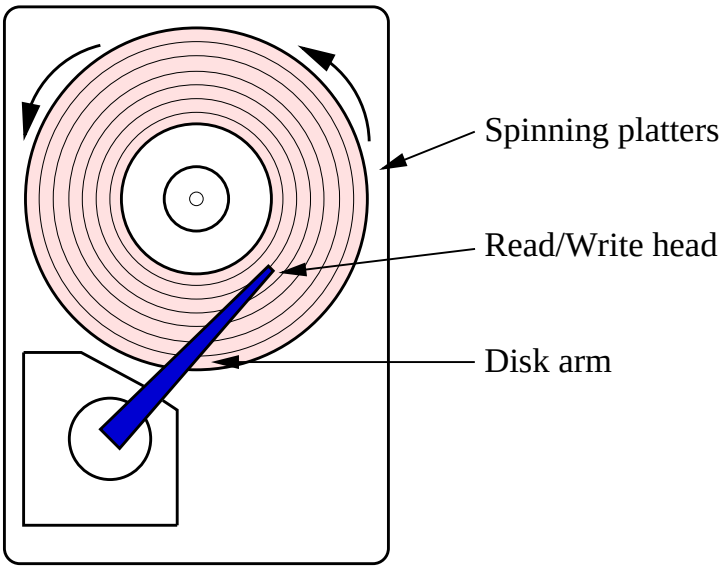
This chapter serves as primer for modern network storage systems. I begin by describing the functionality these systems implement, the basic building blocks of these systems, and how they are arranged to form networked storage systems. I also discuss the protocols used to pass information between these components. I conclude with a survey of the algorithms and related heuristics that are referenced in later chapters.

2.1 Hard Disks

The base of the storage hierarchy in modern computer systems is the hard disk. The key characteristics of hard disks that distinguish them from other parts of the memory hierarchy are persistence and low cost. Persistence refers to the property that once a value is written to the disk, it can be read at any time in the future – even if the disk has been turned off or reset in the meanwhile. Unlike the contents of volatile memory (such as RAM), which are lost if the system loses power or crashes, information on disk is assumed to be persistent.

The property that disks have low cost has a more far-reaching effect on modern system design – gigabyte for gigabyte, hard drives are one of the least expensive methods for storing data available. In fact, in many contexts using disks to provide on-line data storage is only marginally more expensive than storing the same data on off-line storage (such as archival media like magnetic tape, optical disks, etc).

There exist other forms of persistent (or *non-volatile*) memory, such as flash memory, or ordinary memory powered by an uninterruptible power supply. These are used to implement persistent storage in situations where disks are too slow, too awkward, or consume too much power. However, every year disks get smaller and consume less power; the current micro-drives are roughly the same size as a flash memory card and are nearly as rugged. It is plausible that within the next few years devices that have traditionally used non-volatile RAM for reasons of size and power, such as digital cameras, USB keychain drives, and mini-MP3 players, will migrate to inexpensive micro-drives. The speed issue will take more time to overcome – for many tasks, the least expensive flash memory is an order of magnitude faster than the fastest hard drives, and battery-backed RAM is still faster. The primary reason that disks are still used at all is that their cost per gigabyte is roughly 1% the cost of the same amount of non-volatile solid-state memory. Until this changes, hard disks are here to stay.



- One or more *platters*. The platters spin at a constant rate, typically between 5400 RPM and 15000 RPM (and therefore it takes 4-11 milliseconds to complete each rotation).
- The data is recorded magnetically in concentric *tracks* on the platters. Some platters permit data to be recorded on both sides.
- The set of all tracks at a given radius from the center (which may be more than one, if there is more than one platter or the platter allows use of both sides) is called a *cylinder*.
- Each track is divided into fixed-sized *sectors*. A sector typically contains 512 bytes of accessible data. Each sector on the disk has a unique address.
- The disk *arm* moves the *read/write head* from one track to another. If there are several recording surfaces, then each has its own head. Each head is attached to the single disk arm, however.
- The movement from one track to another is called a *seek*. The time required to move the disk arm from one track to another is typically 3-5 milliseconds, but varies depending on how far the head moves.
- When the disk head is not seeking, it can read or write each sector that passes beneath the head.

Figure 2.1: The internal structure of a typical modern disk drive.

2.1.1 Disk Performance Characteristics

The performance characteristics of disks can be defined in terms of bandwidth and latency. Bandwidth is the overall transfer rate with which data can be written to or read from the disk. Bandwidth is typically measured in units of megabytes per second. Latency, in contrast, is the time required to complete a individual operation. As an example, the Postal Service has an enormous bandwidth because it is able to move gigantic amounts of data from one place to another, but unsatisfactory latency for an interactive system because it typically requires at least a day to move even the smallest amount of information. In contrast, the connection between a pair of telephones has a very low bandwidth compared to the Postal Service, but it also has low latency because it can transfer small amounts of data in milliseconds instead of days.

Unfortunately, disks are slow, especially when compared with RAM. As illustrated in Figure 2.1, disks are mechanical devices – their performance characteristics are determined by how fast the platters spin and how quickly the disk arm can move the head from one track to another. While the bandwidth of RAM is measured in gigabytes per second and the latency is measured in nanoseconds, typical modern disk drives have a bandwidth of between 40 and 100 megabytes per second and a typical latency of 5-8 milliseconds for operations that move the head to a new location. To make matters even worse, disk latency depends on a number of complex factors that are difficult to anticipate and vary from one moment to the next and from one location on the disk to another. It may take a different amount of time to complete a particular operation each time it is executed.

At the time of this writing, contemporary disk bandwidth is sufficient for many applications. For those where it is insufficient, techniques such as redundant arrays of inexpensive disks (RAID) may be used [59]. The benefit of RAID in this context is that the operation of reading or writing data can be performed in parallel on several disks simultaneously, thereby increasing the effective bandwidth. This method offers no solution to the latency problem, however. Therefore, the true bottleneck of most storage systems is the latency of the disks used to implement the persistent storage.

To understand why disk latency has not improved (and is unlikely to improve dramatically in the future) and why efforts to reduce the impact of latency on system performance are difficult and frustrating, it is necessary to understand how disks work.

2.1.2 Disk Mechanics

The basic structure of modern disk drives and definitions for related terminology are illustrated in Figure 2.1.

A disk implements the abstraction of random access block-addressable memory. Each sector of the disk has a unique address, and may be accessed individually. The contents of a disk sector change only when explicitly written and retain their value (so that they may be read in the future) even when powered down.

From the perspective of performance, the essential challenge of a disk drive is that the latency of access to different parts of the disk is non-uniform. To make matters even more complex, the latency of access to a given sector is not simply constant but instead depends on the current position of the read/write head relative to the sector.

2.1.3 Zone Effects

Modern disk drives, with few exceptions, use a technique known as zoned constant angular velocity coding (ZCAV) to increase the total disk capacity and average transfer rate [48]. ZCAV can be thought of as an approximation of constant linear density coding, modified to allow the disk to spin at a constant rate and provide an integral number of disk sectors per track. The innermost cylinders of the disk drive contain fewer sectors than the outermost tracks (typically by a factor of 2:3, but for some drives as much as 1:2). The transfer rate between the disk and its internal buffer varies proportionally; in the time it takes to perform a single revolution, the amount of data read from or written to disk can vary by a factor of almost 2 depending on whether the head is positioned at the innermost or outermost track. If the transfer rate between the disk drive and the host memory is greater or equal to the internal transfer rate

of the disk at the innermost track, then the effect of the differences in transfer rate will be visible to the host. For contemporary SCSI and IDE drives and controllers, which have host interface bandwidth exceeding their maximum read/write speeds, this difference is entirely exposed. If the proportion between the number of sectors in the innermost and outermost cylinders is 2:3, then reading a large file from the outermost cylinders will take only two-thirds of the time (and two-thirds of the number of seeks, since each track at the outside of the disk contains more sectors).

This effect has been measured and analyzed and is the basis of some well-studied techniques in file system layout tuning [48, 87]. Beyond papers that explicitly discuss methods for measuring and exploiting disk properties, however, mention of this effect is rare.

The ZCAV effect can skew benchmark results enormously, depending on the number of files created and accessed during the benchmark. If two independent runs of the same benchmark use two different sets of files, then the physical location of these files on disk can have a substantial impact on the benchmark, as demonstrated later in Section 5.3.3. If the effect that the benchmark is attempting to measure is subtle, then it may be completely overwhelmed by the ZCAV effect. It may require a daunting number of runs to statistically separate the effect being measured from the noise introduced by using blocks from different areas of the disk.

2.1.4 The Future of Storage

There are always new storage technologies on the horizon. In low-end devices, the high capacity, low cost, reliability, and time-tested nature of current disk devices makes them hard to supplant. It is possible, however, that in the future disks will be replaced by technologies such as flash memory or MEMS-based devices [88]. However, if disks are replaced by MEMS, then many of the same problems remain; they are physical devices with non-uniform access latencies [70]. If, on the other hand, disks are replaced with flash memory (or a similar media that provides uniform latency of access and high bandwidth) then nearly all of the performance issues that plague disk-based storage will immediately vanish.[1]

2.2 File Systems

Storage systems are used to implement many data storage paradigms. For example, some storage systems masquerade as "raw" partitions that are directly accessed and managed by applications such as databases or used by the operating system for purposes such as swap space or backing store for demand-paged virtual memory [22]. In these applications, the disk is treated as nothing more than an addressable array of sectors, almost as if it were an extension to the main memory of the system.

In this thesis, however, I will focus on the use of a storage systems to implement a *file system*. A file system provides a higher and much more convenient abstraction than an array of sectors. Ordered, arbitrarily-sized sets of sectors can be given names to implement *files*, and these names may be arranged in sets to create *directories* (also known as *folders* on some systems), which are typically implemented in such a way to provide a hierarchical organization.

2.2.1 File System Functionality

File systems typically support a core set of capabilities, including operations to create, rename and delete files and directories, and write data to and read data from a file. Modern file systems also provide additional control over

[1] The lessons learned from disks should not be forgotten, however. In the future, high performance RAM systems will eventually confront some of these issues again, however – due to constraints such as the speed of light, the latency of memory access is bounded by the physical distance that the information must travel from the RAM to the processor. As processors grow faster, it will become worthwhile to take advantage of the fact that RAM that is physically closer to the processor can be accessed more quickly than RAM a few centimeters farther away.

how files are used (for example, by limiting access to a specific set of users). They also provide access to additional information about each file and directory, including values such as the identity of its owner, when it was last accessed, and how many bytes of data it contains. Most file systems also permit the owner of a file to change some or all of these attributes.

For a more comprehensive introduction to file systems and the functionality they provide, consult a modern operating systems textbook such as that by Tanenbaum [85].

There are many storage architectures that are arguably equivalent to file systems, or provide a slightly different set of operations in order to improve support for particular applications. For example, web proxies and web servers can be implemented more efficiently on top of custom file systems rather than general-purpose ones [28, 76]. In this thesis, however, I will focus exclusively on providing general file system functionality.

2.2.2 Typical File System Implementation

The functionality of common file systems is typically implemented in terms of four underlying objects:

Containers: The container can be thought of as the anchor of the file. It contains nearly all of the information *about* the file (but does not contain the data).

On UNIX, containers are implemented as *inodes*. Containers are typically implemented in such a way that they can be referenced by a small identifier. On UNIX, such a reference is called an *inode number*.

Directories: A directory is a set of mappings between names and containers (or references to containers).

Data: The data is an un-typed ordered sequence of bytes.

Index Blocks: The index blocks are used to contain information about the location of other blocks. For example, the container of a file might use a linked list or tree of index blocks to hold the addresses of the blocks that hold the data of the file.

This implementation may seem overly complex, but each component is necessary. It might appear that containers and directories could be combined into a single structure, but this is awkward because a single container may have more than one name. It is also awkward in terms of implementation because the name of each file may be of arbitrary length while it is beneficial for the containers to be of fixed size. Furthermore, the containers are updated frequently, while the names are rarely changed. It has proven simpler to implement file systems with containers and directories as separate objects.

The containers, directories, and index blocks are sometimes referred to as *metadata* because they are data used to implement the file system, not data represented by the file system. This definition is somewhat clumsy because often the metadata (such as information about the size of a file, and when it was last modified) is of interest to applications and therefore made visible. A more pragmatic definition is that the data blocks that contain values written by an application are *not* metadata, and all other blocks are.

2.3 Strategies for Improving Storage System Performance

In this section I outline some of the techniques that have been developed in order to optimize file system performance by minimizing the expected number of disk operations necessary to implement each file system operation and to reduce the expected latency of each operation by reducing the positioning delay.

This is not meant to be an encyclopedic treatment of this subject, but serves principally to introduce and familiarize the reader with techniques that will be used in later chapters.

Most strategies for improving storage system can be described in terms of three primitive strategies:

1. *Avoid using the disk at all.*

   The technique of caching is the best way (short of replacing the entire disk with non-volatile RAM) to avoid using the disk for read operations. Caching is discussed in Section 2.3.1.

2. *Avoid synchronous disk operations.*

   When it is necessary to access the disk, it is best to do so in an asynchronous manner so that the application is not blocked waiting for the disk. Asynchronous operations are discussed in Section 2.3.2.

3. *Arrange the data on the disk in a way that minimizes the cost of reading and/or writing the data.*

   Methods for arranging the data on disk in order to maximize the performance are discussed in Section 2.3.3.

2.3.1 Caching

Caching is the technique of using faster devices (such as RAM) to hold temporary copies of the contents of slower memory devices (such as a disk). In this case, the basic idea is to keep an in-memory copy of each sector of data written to or read from the disk, along with a table giving the address of the in-memory copy of each sector. If a read is requested for data that is already in the cache, then the data can be retrieved from memory rather than from disk. If an in-memory copy does not exist, then the data is read from the disk and an in-memory copy is created (in case the data is accessed again).

Modern file systems depend heavily on caching. Nearly all disk drives also contain a cache, called the *track cache*, that holds a copy of recently written data and the contents of the current track.

The essential problem with caching is that it is not usually practical to store all of the data read from the disk in RAM. At the time of this writing, the cost per gigabyte is roughly one hundred times higher for RAM than for disk.[2] Therefore, the memory available for caching is, on nearly all systems, only a tiny fraction of the size of the disk.

Since not all of the contents of the disk can fit in RAM, the obvious question is what should be cached and what should not. A large number of heuristics have been proposed as a solution. The most commonly used heuristics are variations of the "Least Recently Used" (LRU) metric, which always caches the most recently accessed sectors. If a sector that is not in the cache is accessed, and there is not enough space to cache each sector as it is accessed, the sector in the cache that was least recently used is sacrificed to make space for the new sector. LRU is popular in a number of contexts because it combines conceptual simplicity with good performance across a wide variety of applications. There are some simple and common access patterns that cause LRU to fail – for example, sequentially reading more sectors that will fit in memory will cause the contents of the cache to be completely replaced. If these new sectors are each accessed only once, then keeping them in the cache is a waste of RAM.

Luckily, for most workloads, a cache that is only a small fraction of the disk size is sufficient. Most workloads exhibit a high degree of *locality* – there is a large skew in the probability distribution of sector accesses. In a typical workload, a small number of sectors are the target of most of the accesses and the rest of the sectors are referenced infrequently (if at all). This means that even a small cache can significantly reduce the number of disk accesses. Unfortunately, the property of high locality works in the opposite direction as well. Because the tail of the distribution of sector accesses is long, once the majority of accesses are serviced from cache adding more memory to the cache usually does not help very much.

Caching is such an important technique that it is still an area of active research with important developments each year [8, 45].

[2] Historically, both RAM and disk prices tend to decrease over the long term. The cost ratio between RAM and disk will change, but it appears likely that both the cost ratio and the ratio of information density (bits per cubic volume) will remain in the favor of disks by well over one order of magnitude for the foreseeable future. Furthermore, the effective density of bits per cubic volume for disk is substantially higher than that of RAM.

2.3.2 Asynchronous Operations

When disk operations cannot be avoided, the best way to improve performance is to perform these disk operations asynchronously.

In the case of reading, this is done by a technique called *read-ahead* or *pre-fetching*. The system attempts to identify sectors that it anticipates will be read in the near future and read them into the cache so that they are already there by the time an actual read is requested.

It may seem counterintuitive to take up valuable cache space for data that has not been requested, but recall that this strategy trades the space used by the item that the cache eviction policy thinks is least important to make room for an item that the file system is likely to access soon. If the file system predictions are reasonably accurate, this strategy pays high dividends [77].

In the case of writing, the technique of asynchrony takes the form of delayed writes. The idea behind a delayed write is to postpone the actual disk operation for some period of time with the hope that the data are deleted or overwritten before they are written or that their write can be scheduled according to convenience rather than urgency. For example, if an application does a delayed write of a particular sector and then, a moment later, does another write of the same sector, then it may be that the first write did not reach the disk and now is not necessary at all. Studies have shown that many data sectors are overwritten or deleted within moments of being written, and thus delaying writes for a short period of time (anywhere from a few seconds to a few minutes) can considerably reduce the total number of actual disk write operations.

The drawback of delayed writes is that there are many situations where the data must actually be written to disk for the sake of correctness. One of the important characteristics of disks is that data written to them is persistent in the face of system failure. Data that is scheduled to be written via a delayed write will be lost, even though other data that was written chronologically later might already be written to disk. In the aftermath of a system crash, the contents of the disk may be in a state inconsistent with the order in which the application requested writes (and even worse, it may be self-inconsistent). Therefore systems that use delayed writes also provide the capability to force some data (i.e., a single write, all the outstanding delayed writes for a particular file, or all the outstanding delayed writes for an entire file system) to be written to disk without delay.

The technique of soft-updates provides a way to reduce the risk of inconsistent metadata resulting from delayed writes by ordering writes in such a way that the potential for inconsistencies is reduced or eliminated [30]. For example, creating a file requires creating a container and binding a name to that container. If the binding is written before the container and the system crashes between the two writes, when the file system attempts to recover the directory entry for the name of the file will refer to a container with an undefined value. If the file system ensures that the writes are done in the opposite order, so that the container is always safely written before the directory entry, then this hazard is removed.[3] The file might be lost in the event of a crash, but it will not be in an inconsistent state. For some workloads, metadata updates are a bottleneck and ability to delay them with relative safety provides an enormous performance advantage.

2.3.3 Layout Heuristics

One of the defining characteristics of a file system is the method it uses to choose how to arrange items on the disk. There are many methods for this, but I will discuss only a few key examples.

As mentioned in Chapter 2.2.2, the creation of an object such as a file requires the creation of several on-disk structures: the container, the directory entry, and possibly several index blocks and blocks to contain data. Ideally these structures should be arranged on the disk in such a way that operations that require accessing more than one of

[3] This example is faithful to the idea behind soft updates, but the actual mechanisms used to discover and order the dependencies of writes ware much more complex. In the FreeBSD implementation of the Fast File System approximately half of the lines of code are dedicated to the implementation of soft updates.

them (such as reading all the data blocks in order, or writing a data block and then updating the file container with a new length) can be done with a minimum amount of positioning. Finding an optimal arrangement is usually intractable (even for a given access pattern) because the number of possible layouts is astonishingly large.[4] Therefore a number of heuristics have been developed to guide block allocation.

Placement and Allocation Policies

- *Update-in-Place: FFS*

  The Fast File System (FFS) [43] is the implementation of a set of heuristics that have fared remarkably well during the last twenty years. Many of these ideas have also been used by more recent file systems. The FFS has been updated in a number of ways since its original design, but in this section I focus on the basic heuristics:

  – Instead of using physical disk sectors as the basic unit of storage, contiguous groups of sectors are organized in *blocks* consisting of several sectors. Since the overhead of performing a disk operation is roughly the same no matter how much data is accessed (up to the limit on the transfer size of the device), this means that the overhead per byte is reduced proportional to the block size. In fact, in the original implementation doubling the blocksize more than doubled the system performance.

    Virtually all file systems now use blocks instead of accessing the disk sector-by-sector.[5] In modern systems, a typical block size is 16K, which means that the data for a single file of 16K or less can be read or written in one disk operation.

  – Once a block has been allocated to a file system object, it remains allocated until that object is destroyed (or truncated, in the case of data or directory blocks). When the contents of a block are written, the new value is written over the old value.

  – The disk is divided into virtual super-cylinders named *cylinder groups*. Blocks within a cylinder group are "close" to each other in terms of positioning time. The allocation policy for data and directory blocks is to attempt to place the first few blocks of every file and directory in the same cylinder or cylinder group as the parent container.

    The guiding philosophy for block allocation is to use a block that extends the current file or directory whenever possible or a block within the same cylinder or cylinder group, thereby allowing sequential access or a reasonable approximation of sequential access. At the same time, however, there is a limit to how many blocks may be allocated by any particular file or directory from the same cylinder group. Once that limit has been reached, further allocations are taken from a different cylinder group (if possible). This heuristic helps to ensure that until the disk becomes very full, there are always free blocks within every cylinder group and it is likely that other files within that cylinder group may be extended in a contiguous (or nearly contiguous) manner.

    It follows that very large files will not be allocated contiguously, but they will be allocated in fairly contiguous chunks. Performance for large files is slightly degraded but the property that *most* consecutive blocks are allocated in a contiguous manner improves the performance of a system as a whole.

- *Logging: LFS*

  The premise behind the Log-structured File System (LFS) is that reads are likely to be satisfied from cache, and therefore the most important component of file system performance is write performance. To accomplish this, LFS aggregates writes into large, contiguous regions called *segments* and appends them to a *log*. All writes are placed in the log, including changes to index blocks, containers, and directories as well as data.

[4] The mind cannot grasp these kinds of numbers abstractly; they require physical metaphors. For a 40GB disk, to write out the *count* of possible sector arrangements would require more than 12,000 pages (assuming each page contains 60 rows and 80 columns of decimal digits).

[5] Many modern file systems also use *fragments*, which are sub-block allocation units intended to reduce the amount of space used to represent very short files. I will not discuss fragments explicitly.

Conceptually, reading a log is simple: to find the current value of an object, start at the most recently written log entry and scan backward in time to find the most recently-written value of the object. Fortunately this process is necessary only in the case of disaster recovery. In normal operation, finding an object is done in almost the same way as any other file system. The awkward step in this process is mapping container references (which are called inode numbers in LFS) to physical disk addresses. A separate data structure, called the *ifile*, is used to streamline this process. The *ifile* is also the key to good performance: if there were no way to map inode numbers to physical addresses outside of the context of the log, then every time an inode was updated all of its ancestor directory inodes (defining the path all the way back up to the root) would need to be updated as well.

The difficulty of LFS is that every time a block is written it may be written in a different place – and every version of the block except the most recent is, by definition, out of date.[6] LFS uses a process called *cleaning* to remove out-of-date versions of old blocks. In the simplest form, the cleaner traverses the file system, gathering up all of the up-to-date or *live* blocks in a set of segments and writing them into a new segment and then declaring that the old segments are clean and may be reused.[7] The performance of LFS depends on two things: first, that the cleaning cost is low (or at least low enough so that it does not overwhelm the benefit of using LFS) and, second, that most reads are satisfied from cache. Because LFS does not make an effort to arrange items on disk in a way that is convenient for reading, a workload that cannot satisfy reads from cache will suffer on LFS.

- *Journaling and Other Hybrids*

  Journaling attempts to combine the best of both the FFS and LFS worlds. Most of the disk uses an FFS-style layout, which makes reads and operations such as directory traversal efficient. Recently written data is treated differently – instead of being written directly into this FFS-like arrangement, it is written to a short log. After items have been in the log for a certain period of time without being overwritten (or the disk is otherwise idle) the contents of the log are pushed out, in FIFO order, to the FFS area. Cleaning the log is simple, because as soon as a segment is pushed out to the data area, it is, by definition, ready to be reused.

  Some modern file systems use Journaling for metadata but not for ordinary data to achieve a performance boost similar to that provided by soft updates.

Reorganization and Reallocation

As a file system is used, it may become evident that the layout selected for a given set of blocks is suboptimal. To deal with this situation, some systems detect block access patterns and reorganize the layout of blocks on disk.

Block Reorganization

One technique used to minimize the expected positioning delay is to put the data that are accessed most frequently in the "middle" track of the disk. (In this context, the "middle" track of the disk denotes the track that is halfway between the innermost and outermost tracks, rather than the track that is nearest to the disk's axis of rotation.) The rest of the data are also arranged according to their temperature – hotter data are placed closer to the middle. Thus the coldest data are in the inner and outermost tracks of the disk. This heuristic is called the pipe-organ heuristic because a plot of the relative temperature of each track resembles a configuration often used by the pipes of a pipe organ.

The benefit of this strategy is that it minimizes the expected time required to seek from each requested track to the next, assuming that each track is randomly and independently requested and the probability of requesting a

[6] Some file systems, such as Elephant [67] and WAFL [34], permit the user to access past snapshots of the file system via special directory operations that permit the user to see the "old" versions of files and directories. This is extremely useful for seeing what has changed on a file system or simply to un-do accidental modifications or deletions.

[7] My explanation of cleaning is rather high-level and meant only to provide an intuition for how LFS is implemented. There has been a great deal of interesting work on good cleaning algorithms and their analyses, but it is not essential that the reader understand them.

particular track is simply the sum of the numbers of accesses to the blocks within that track divided by the total number of accesses.

This is fairly intuitive: if the hottest blocks are near the middle of the disk, then the location of the disk head at any instant in time is most likely to be near the middle of the disk (which minimizes the maximum positioning distance for the next seek) – and the middle of the disk is also the location of the next sector to be requested, therefore minimizing the expected seek to the next position.

If all of the tracks contain the same number of blocks, then the proof that the pipe-organ arrangement is optimal for a random workload is reasonably straightforward (although it does require a bit of mathematical manipulation) [81]. Unfortunately, as shown in Section 2.1.3, this assumption does not hold, because there is a considerable variation in the number of blocks per track in the different zones of the disk. The location of the "middle" of the disk and the best arrangement of the data around this "middle" depend upon the probability distribution of block accesses and the distribution of track sizes.[8] As a simple example, imagine a disk whose outermost (and largest) track contains 100 blocks and which has exactly 100 blocks with a non-zero probability of access. In this case, the most efficient layout is to place all of these blocks on the outermost track, thereby eliminating all inter-track seeks. The effect of zoned disks on optimal block arrangement has been explored by Tewari et al. [86].

2.4 Network Storage Systems

2.4.1 NAS and SAN

From the perspective of the client, the most important architectural distinction between different types of network storage systems is the abstraction they export. There are two primary levels of abstraction, which have been given the somewhat confusing names of network-attached storage (NAS) and storage-area networks (SAN). SAN systems export the abstraction of block-addressable storage, similar to that of a disk. NAS systems, in contrast, export a higher level of abstraction, such as a file system.

The techniques for optimizing storage system performance that I explore in later chapters are based on knowledge about the file system that the client is accessing, and therefore these techniques require the NAS abstraction to work in a network storage environment.[9]

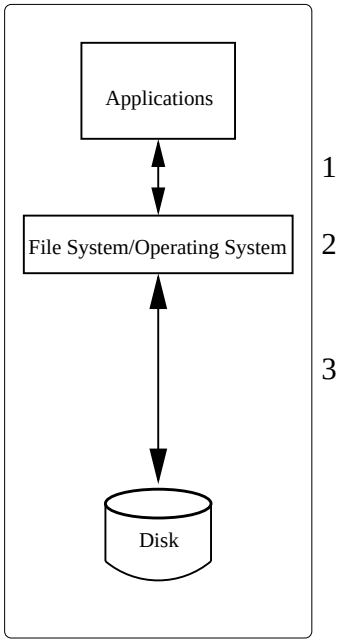
2.4.2 Local File Systems versus NAS

Figure 2.2 shows the abstraction of the storage system that is provided to the application. At the top of the figure, the application requests fairly high-level operations such as creating files, writing data to a file, reading data from a file, or removing files. The file system is responsible for translating these requests into disk read and write operations. For example, if the application requests data from block 10 of file `foo`, it is the responsibility of the file system to determine which disk sectors contain the data for that file block.

Using this model, it is not hard to imagine that the application and the file system can cooperate to arrange the data on disk in such a way to maximize performance. If there is only one application, then the file system can be tuned for the requirements of that application, or the application can supply explicit directions (or hints) to the file system about what it plans to do in the future.

[8] It might also seem that the location of the "middle" of the disk is dependent on the amount of data to be stored on the disk. Instead, I choose to model every disk as always being completely full, although some of the blocks might be infinitely cold. Nevertheless, this property is particularly problematic for real systems because the "middle" of the disk moves dramatically as the amount of data present on the disk increases.

[9] These techniques can also be applied to local file systems or file systems implemented using SAN storage, but in both of these cases the file system functionality is implemented locally.

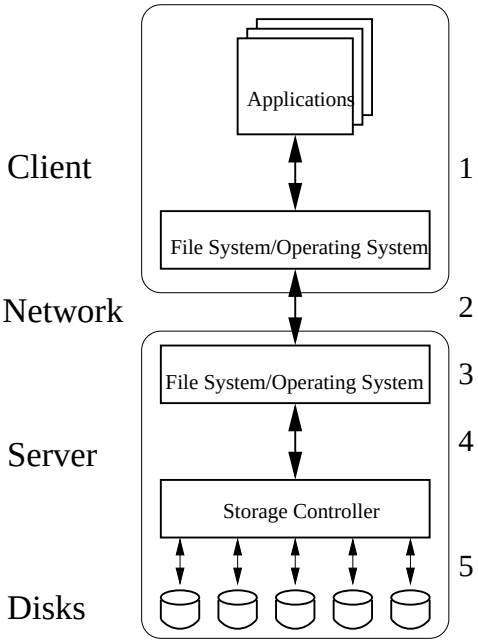


Storage Access:

1. The application requests access to an object in the file system via the operating system in order to perform a read or write operation.
2. The operating system invokes the file system to learn where the object is located on disk.

   The information about where each object is stored may reside on the disk as well, and so this step may cause additional disk operations.
3. The file system uses the lower level of the operating system to request the necessary disk operations.

Figure 2.2: A conceptual model for local file storage.



Storage Access:

1. The application sends a request to its local operating system/file system for access to its storage.
2. The local OS/FS translates the request into a network protocol and forwards it to the server via a network.
3. The server interprets the request as a request for local storage and sends it to its local OS/FS.
4. The local file system translates the request into logical block operations and determines what logical blocks are required to perform the operation and sends the translated block-level operations to the storage controller.
5. The storage controller maps the logical block operations to physical block operations and performs the requested operations. (Depending on the implementation of the storage controller, there may be several more layers required to implement this step.)

Figure 2.3: A conceptual model for NAS.

Unfortunately, the abstract model portrayed in Figure 2.2 is, for complex systems, nothing more than that. The implementation of a networked storage system is quite a bit different. Something closer to reality is shown in Figure 2.3, which shows the block diagram of the architecture of a generalized networked storage server. Like Figure 2.2, at the top of diagram are the client applications, which directly or indirectly create all of the requests to read or write data, and at the bottom of the diagram are the disks, which still serve as the primary repository for data in the system.

Considering only the external aspects of the model (applications at the top, disks at the bottom), these two models might seem to share many properties. This is not true, however, for NAS servers:

- There may be many applications running concurrently. A moderately-sized storage server may handle requests from dozens or hundreds of applications each second. These applications typically have no knowledge of each other, and thus do not cooperate with each other. Instead, they compete for resources.
- There may be many disks, and the arrangement of the data on those disks may be dynamic or unpredictable to the application or file system. For example, reading block 10 of file `foo` might actually require reading sectors from several disks (if the disks implement a RAID-5) or it might only require reading a few sectors from one of several disks (if the disks implement RAID-1).

The more important difference, and the one that will cause much anguish later, is the number of layers of abstraction and virtualization that separate the applications from the disks. In the naive model, there is a fairly straightforward correspondence between the requests made by the application and the activity on the disk. In the more realistic model, this correspondence may be quite complex.

2.4.3 NFS

Users prefer transparent implementations for their storage services and therefore many remote storage servers masquerade as local file systems, at least to some degree. In the UNIX world, the protocol most frequently used to provide this illusion is the Network File System (NFS). In the context of this thesis, NFS refers only to NFS version 2 protocol (NFSv2) [56] and the NFS version 3 protocol (NFSv3) [14]. At the time of this writing, implementations of version 4 of the NFS protocol are beginning to appear, but this version represents a significant departure from the philosophy of versions 2 and 3 [55, 75, 74], and its adoption has not been widespread.

NFS is by no means the only network file system; there are and have been many others.

The implementation of NFS has had a wide influence on the manner in which UNIX file systems are implemented. NFS was one of the first network file systems to be implemented in the kernel, and doing so forced the redesign of some of the internal UNIX interfaces in order to split the file system in such a way that it could be implemented as two pieces communicating via a network. The result is the vnode (virtual node) and VFS (virtual file system) interfaces [38].

One of the interesting goals of the NFS design is to function well in unreliable environments. The protocol is stateless and each request is self-contained, so that the server does not need to keep track of any aspect of the dialog it has with any of its clients. This is particularly useful when the server is unreliable – the server can crash and lose all of its internal state (except for changes to the file system that it claims to have committed to stable storage), reboot, and pick up exactly where it left off without the clients noticing anything except a momentary delay.[10]

[10] It would be interesting to analyze whether this design decision still makes sense now that NFS servers are typically fault-tolerant machines several orders of magnitude less likely to fail than the clients they serve.

2.5 Approaches to Improving Performance

The difficulty in applying the general strategies and performance tricks described earlier in this chapter is that the most effective techniques tend to be those that work especially well for one type of workload – and might actually hurt performance for others. For example, LFS is ideal for update-dominated workloads, while FFS is superior when the workload is read-dominated and accessed sequentially. Journaling works well for more arbitrary patterns as long as there is always enough spare capacity to move items out of the journal – when the journal is full, the asynchronous updates that characterize this strategy become synchronous. Particular layout arrangements, such as immediate files [53] or embedded inodes [29] optimize the case of small files, but at the cost of potentially slowing down operations on large directories or files. There are also issues to be considered with respect to caching and disk scheduling.[11]

At present, it is necessary to understand the characteristics of the workload that the storage system will support before selecting the best set of implementation policies and strategies from those available. Only sophisticated administrators attempt to do this at all, and most users are unaware that such options even exist.[12]

My goal is to look beyond the *status quo* and imagine storage systems that work well for a variety of workloads and can tune their own parameters to control both gross file-system-level and fine file-level behaviors. There has been some work related to this at the level of the storage controller: for example, the AutoRAID system can automatically migrate blocks from one RAID representation to another based on the observed access patterns of these blocks [93]. There is also work on automatic adaptive techniques that optimize block access via movement [42] and replication [1]. The differences between this work and my own is that they are *adaptive* and they focus on block-level storage. I am interested in optimizations at the file level, rather than the block level, my goal is for the file system to anticipate how files will be used *before* they are accessed for the first time.

One of the reasons that this is a considerable challenge is the increasing separation, as measured in terms of the layers of abstraction and indirection, between the applications that drive the workload and the storage arrays that implement the operations necessary to satisfy it. This separation is referred to in the literature as the *information gap* [23]. In the rest of this Chapter, I discuss current approaches to bridging this gap.

2.5.1 Hints

Hints are a general technique and are applicable to other systems beyond storage systems [41]. The idea behind *hints* is, as its name suggests, that the application can provide information about its future requests to the storage system that might be helpful to the storage system. Ordinary systems can use only the past as a basis for deciding how to act in the future. In the case where the application has some additional knowledge about what the future looks like, hints can provide the storage server with a more accurate glimpse into the future.

Examples of hints that an application might send to the storage server include:

1. ”I plan to read this file from beginning to end.”
2. ”I plan to read only the first block of this file.”
3. ”After I read this block, I do not plan to read it again for a long time.”

Given the techniques that I have discussed earlier in this chapter, it is not hard to imagine how the storage system might use these hints: in the first hint, the storage system might immediately begin to pre-fetch blocks from the file,

[11] Disk scheduling is not discussed in this chapter, but will be described in Chapter 5.

[12] And for many users, there are no options to speak of anyway. While there are a variety of file systems with different performance characteristics for operating systems such as Linux, on widely-deployed operating systems, such as Windows and Mac OSX, the default file system configuration is the only viable option.

knowing that they will all be read. In contrast, the second hint might cause the server to disable pre-fetching for this file. The last hint could be used to guide cache eviction policy; if the application believes that this block will *not* be read again in the near future, it should be evicted from the cache before blocks whose hinted use is more immediate.

It is important to remember that, at best, hints contain no relevant information that the storage server will not discover eventually on its own. Hints are not required for correct operation, and they have no effect or relationship to the underlying file system semantics. The value of hints is that they may allow the storage server to anticipate what this information will be and choose the optimal behavior in response. This may be quite valuable – there is ample evidence that hints can be successful in some domains, particularly client-side caching and pre-fetching [12, 16, 52]. Sarkar et al. show that hints are also an efficient and effective way to guide cooperative caching [69].

A major criticism of hints is that they require extra effort on the part of the application programmer because the application must be augmented to provide these hints. This is a non-trivial process, in part because it requires some knowledge about the underlying system. To address part of this problem, there has been work in having compilers generate hints automatically from their analysis of the application source code [12, 52]. This has been quite successful for applications with regular file access patterns, but these techniques have not been demonstrated in wider application domains.

It is important to note that hints might be wrong – if the storage system puts blind trust in the application providing the hints, it is possible for the system to do substantial amounts of extra work as a result. However, it is just as important to note that this is not a weakness specific to hints – any application may generate a workload that (intentionally or not) has a negative impact on the performance of the storage system. A more likely problem is that the hints might simply not be useful, because they assume some knowledge of the underlying storage system architecture.

2.5.2 Gray-Box Techniques

A technique that has received a great deal of attention in the file- and storage-system communities in the last few years is the *gray box* technique [5].

The name *gray box* is based on the name *black box*, which is an abstraction for a device that performs a specific function based on its inputs but reveals no information to an external observer about how that function is implemented. A gray box, in contrast, reveals some information about how the function is computed, or allows an observer to guess some of the details of the actual implementation given some knowledge of the basic implementation strategy and information that can be measured from the system.

Considered as a black box system, disks perform two functions: reading and writing fixed-length blocks of data. When treated as a gray box system, however, disks can reveal a great deal of their implementation via the time required to perform different operations.

As a simple example, imagine that we have an ordinary disk drive whose rotational latency we wish to know. One way to estimate this is by synchronously writing a particular sector of data and then, a short moment later, synchronously re-writing the same sector. After the first write, the disk must wait until the disk has performed a full rotation before the sector passes under the disk head and can be re-written. Therefore, assuming that there are no intervening disk operations, we can estimate the rotational latency of the disk by measuring the elapsed time between the completion of the first and second writes.

A similar method reveals the size of each cylinder on the disk (and observe the zone effects). A simplistic method is to request the disk to write sectors α_i and α_{i+1} for each i less than the disk size, and make note of all values of i where the elapsed time between the writes is unusually large. If sectors α_i and α_{i+1} are on the same cylinder, then they can be written successively (or both writes can be coalesced into a single operation). If they are on different cylinders, however, the disk must perform a seek between the writes, and therefore there will be a noticeable delay between the completion of the two writes.

Note that this technique is meant to serve as simple example of a gray box technique. It is neither perfectly accurate (because it is fooled by remapped bad blocks) nor particularly clever or efficient. For a description of more accurate techniques that have been used to discover the parameters of real systems, see Worthington et al. [96] or

Popovici et al. [62].

Other gray box techniques can be used to discover higher-level information about storage systems, such as how the file system chooses which blocks to allocate for each file and directory. Armed with this knowledge, an application can tailor and augment its I/O requests in such a way that it can control where files and directories are placed on the disk [57] for file systems such as the FFS that use a static policy for allocating blocks [43].

Gray box advocates are now suggesting that it would be a good idea to make current systems such as operating system kernels even more ”gray” in order to expose more of their implementation characteristics [6].

There is no doubt that gray box methods are useful for system that are ”gray” in a useful manner and that behave predictably to changes in their workload. These conditions do not always hold however – and in a network storage system, they are particularly likely *not* to hold. These methods are of little use with storage implementations that employ mutable policies and/or mappings between logical or file block addresses and physical addresses. At the file system level, this include systems such as LFS [65] and WAFL [34] and HyLog [92]. At lower levels, this includes layered storage architectures that hide the details of the implementation through multiple layers of indirection (such as xFS [3]), or adaptive storage architectures such as virtual log-structured disks [21, 91] or self-tuning RAID systems [2, 93].

Gray-box techniques also assume that the application has free reign to do what it likes to the storage system, but in a shared system this is not likely to be the the case. With thousands of applications trying to use gray-box techniques simultaneously, disaster is more likely than success.

My position is that the server should adapt to the workload of each application. The server has the information necessary to weigh the benefits of per-application optimization versus the needs of the system as a whole. It also has more information about overall state of the system.

In short, it does not make sense for the application to bully the server by attempting to craft its workload. This artificial modification to the workload can rob other applications of their share of the server and can confuse the server.

Chapter 3

Gathering and Analyzing NFS Traces

The first step in building an intelligent system is to devise a method to capture as much information as is available to the storage system about the workload. In the parlance of the file system community, a recording of system activity is called a *trace*. In this chapter, I discuss methods for gathering NFS traces and performing basic analyses on them.

3.1 Motivation

The ultimate goal of my research is to increase the performance of a remote storage server by making it more intelligent. Before a system can become intelligent and capable of learning it must be able to observe the context in which this intelligence is to be applied.

The simplest possible strategy for learning would be for the system to explore the space defined by its parameters, observe how its performance varies from point to point, and search for the best position in this space. For this strategy, the only thing that the system needs to observe is how well it is doing. The problem with this simplistic approach is that the space of system parameters is immense, and therefore this approach is not feasible for low-level adaptation. It may be suitable for adjusting some high-level parameters.

The real drawback of this approach, however, is that the system is unaware of anything other than its performance, and is therefore unable to make decisions based on information that might be available *before* the impact on performance can be observed and that the system might learn is related to this impact. As a minimal example, consider a system that can observe the local time in addition to its overall performance characteristics and keep a history of these characteristics over time. It might learn that certain times of the day have different performance characteristics than others and that the best combination of parameters varies from one time to another. From that, it might learn that the best strategy is to preemptively change its parameters at certain times of the day, anticipating changes in the performance characteristics. A system that is oblivious to the time of day, however, would use the same strategy for every time of the day and would never anticipate periodic workload changes.

As a more complex example (and one that more closely mimics the kind of adaptation my work enables), consider a file system that is populated with two types of files: the first type of file is always given a name with a suffix of `wr` while the second always has a suffix of `rd`. The `wr` files are written repeatedly but never read, while the `rd` files are written once and then read repeatedly. A system that can observe file names and per-file access patterns can learn that the set of files is partitioned in this manner and choose a different policy for managing each type. In contrast, a system oblivious to file names and per-file access patterns would never learn to make this distinction and would, by necessity, choose a global policy for managing both types of files. In this situation, because there is no static policy that is optimal for both types of files, the cumulative performance of the system would suffer.

3.2 Types of Storage System Traces

There are several ways to gather traces. The key difference between them is the interface at which the trace is captured. As shown in Figure 2.3, there are many interfaces that storage calls and responses must traverse between the application and the disk, and the traffic across any (or all) of these interfaces may be of interest:

- Application-level: Instrument the applications in order to capture the "high-level" workload generated by the application. The drawback with this method is that the workload generated by the application is often different from that seen by the server. For example, if the application does buffered I/O, then several application-level calls may be combined into one storage system operation. Similarly, if the application makes storage requests that cannot be satisfied with a request to the storage server (for example, the application tries to read 1MB of data in one operation, but the storage server can only perform transfers of 8KB or less).

  It is also often the case that the applications access the storage server via libraries whose interfaces differ remarkably from the interface exported by the storage server. In this case, it is complicated to map the application behavior to storage server calls without examining the implementation of the library as well (and even then, the library may be implemented in different ways on different systems). On the other hand, these traces provide high-level information about what the application is trying to do, and thus may be quite valuable.

- Kernel syscall-level: Capture the application behavior by capturing its calls to the operating system and file system. This removes the problem of finding a mapping from the application libraries to the lower-level calls, and, in many ways, is easier to implement. It is possible to instrument the kernel interface to capture all calls at the `syscall` interface. This has the advantage that by modifying this relatively small and simple piece of the operating system kernel it is possible to capture the calls generated by *all* of the applications in the system instead of only capturing those generated by applications that have been instrumented.

- Internal Kernel/File-level: Inside the client and server operating systems there are additional interfaces. The UNIX kernel uses the `vnode` abstraction as an interface to all file systems, which makes this a convenient place to trace any file system [38]. Other families of operating systems, such as Windows/NT, have a different but similar interface [90]. The obvious drawback of this kind of trace is that it may be non-portable – since these interfaces are not necessarily standard (or even present on all systems), they may not translate well from one system to another.

- Network-level: For a remote storage server, the network packets used to carry the calls to the server and the responses to the client can be captured.

  All of the client request streams converge at the network interfaces of the server, and therefore tracing at these interfaces is advantageous because it is generally much easier to trace one server than to trace an unknown number of clients (and then merge all of these traces together in order to discover the total server workload). In addition, the client/server protocols are well defined and documented (even proprietary protocols such as CIFS have been reverse-engineered). This means that traces at this level are reasonably easy to interpret.

  The drawback is that there are several popular protocols, and they are not all at the same level of abstraction: iSCSI [68] is a block-level abstraction, NFS [56, 14] is an abstraction based on vnodes, and CIFS is a higher level abstraction based on NTFS.

- Block-level: On the server side it is also possible to trace the actual disk requests (on either a physical or virtual layer, above or below the storage controller). Capturing traces here is relatively straightforward, but the traces depend as much on the implementation-specific mapping of file system objects to physical or logical blocks as they do on the application-level workload. Two storage servers managing the same workload may have very different block-level behavior if they use different layout and allocation strategies (for example if one of them uses FFS and the other LFS).

  Although block-level traces have limited use for file system workload analysis, they are quite useful in the analysis and optimization of block-level storage [1, 42] and caching [54, 66].

3.3 Challenges in Gathering File System Traces

Two broad issues make file system tracing challenging: the potential impact on system availability and performance and issues of privacy and confidentiality.

Production servers have, almost by definition, high availability requirements. Installing monitoring software on servers in a production environment is rarely (if ever) a viable option, particularly if the process of installing the software causes the system to be unavailable for any period of time. Furthermore, if there is any suspicion of risk that the monitoring software might cause any decrease in system reliability, the owners and administrators of the system will be even less likely to allow it to be installed. This point is often moot. Because many production environments are based on closed or proprietary software systems, it is not possible to do kernel or application-level monitoring on these systems without some cooperation from the creators of those systems.

Some storage vendors already provide mechanisms that do some or all of the work of gathering traces, and at the very least, most commercial storage manufacturers have monitors built in to their products, which permit them to gather a wide variety of data about their systems. Operating system providers are also developing increasingly sophisticated tracing mechanisms for all aspects of their systems [15, 51, 97]. In these cases, the risk of degrading reliability is minimized, but there is still a potential for impact on the performance of the system (although modern tracing mechanisms are careful to minimize this impact).

Another difficulty in gathering traces from production servers is a combination of social, ethical, and legal issues. Detailed file system traces can reveal sensitive information about the activities of organizations or individual users, and the administrators of these systems have justifiable concerns about protecting the privacy of their users. During my data collection, I contacted several commercial Internet Service Providers (ISPs) in an effort to gather traces from their sites, and although several of the administrators of these ISPs expressed interest, none of them would permit me to gather traces because of privacy concerns.

As a result of these constraints, few kernel or file-system level traces from production systems are publically available and none have complete information. The environments that are the most interesting (such as large public or commercial systems) seem to be those that are the least likely to permit tracing. As a result, much file system research is based on traces gathered from the systems used by file system researchers, rather than a diverse set of workloads.[1] From a scientific perspective, this is quite frustrating because traces are so important to research in this field.

3.4 Anonymized Passive Tracing via NFS

Passive NFS tracing is attractive from a research perspective because NFS is a portable and widespread protocol, used in a broad variety of real-world contexts, and so its analysis is applicable to many interesting real-world problems.

Passive and anonymized NFS tracing solves the problems listed in the previous section without sacrificing the information that I need in order to do my analyses. If I am using a network platform such as Ethernet, this is particularly easy: I can gather all of the network packets going to or coming from the server by using a network tap or by using a network switch to mirror this traffic through another port. Both the clients and the servers are unaware that copies of all their packets are being captured[2] and do not have to cooperate or be modified in any way in order to permit this data to be collected.[3]

[1] In this light, Werner Vogels' file system traces from a variety of Windows/NT systems are particularly impressive – he was able to overcome privacy and confidentiality concerns and gather traces from production systems *and* he did so by making low-level changes to the Windows/NT kernel [90].

[2] In the worst case, interposing a switch with a mirror port does add latency to the network. However, modern Ethernet switches begin retransmitting each packet as soon as they have seen its frame header, and therefore this latency is, in the worst case, on the order of several microseconds (and usually much less). For Ethernet taps, the delay is measured in nanoseconds. In either case, this degree of network latency is hard to detect.

[3] This assumes that the clients have not mounted the NFS file system over an encrypted channel. While this feature is available on many modern

Passive NFS tracing is an old idea, and has been used in many trace studies and file system experiments [9, 10, 20, 33, 50]. The technique of passive tracing on a broadcast network evolved in parallel with broadcast networks such as Ethernet and client/server protocols such as RPC. NFS trace studies began to appear in the literature soon after NFS clients appeared on computer science department networks [33], but they received relatively little attention compared to kernel-based trace studies or studies of other distributed file systems. This changed, however, as NFS became ubiquitous and better tracing tools and methodologies for analyzing NFS traces were devised [9, 50].

To address privacy and confidentiality concerns, I add an anonymization step to my tracing procedure, as described in Section 3.6.1. This step transforms the traces in such a way that user-specific information such as UIDs or filenames is not revealed but preserves the information necessary for many analyses.

Despite the decline in the number of new NFS trace studies in recent years, I believe that the technique of passive NFS tracing in tandem with trace anonymization is more important now than ever, because it may allow researchers to trace systems that would otherwise be completely inaccessible. I hope that the existence of this anonymizer and the value of my findings will convince ISPs to permit tracing, allow these traces to be shared among the research community, and thereby invigorate contemporary file system research.

3.4.1 Implementation Issues

One of the awkward details of the SUNRPC package (the remote procedure call protocol used by NFS) is that response messages do not contain explicit information about the call for which they are a response. Instead, the client sends a (hopefully) unique identifier called an XID to the server along with each call. The server then includes the XID with its response, and the client uses the XID in the response to determine which call corresponds to the response [49].

It follows from this that if I do not observe a call, I cannot decipher the response to the call (because there is little or no information inside the response that reveals the format of the response). It also follows that if I want to be able to decode responses, I need to keep track of all the calls I see (at least for a little while). It is reasonable to assume that if there is no response within several seconds, then it is likely (although possible) that a response will never be seen.

Taking advantage of this fact, some protocol monitors use a circular buffer of calls and search the buffer for the corresponding XID when a response arrives. When the buffer is full, the oldest member is ejected, in the hope that it is too old to see a response. This method is adequate when the message rate is low, which permits the circular buffer to be small and the search to go quickly. For my trace collector, I decided to do something more powerful. I build a hash table of all calls, hashed by XID. When a response arrives, it is fast to find the corresponding call.

The problem with this method is that if the *response* is missing, then the record of the call will sit in the hash table forever, and soon the performance of the hash table will degrade. It is necessary to somehow scan the hash table and cull all calls that are too old to succeed. It is not possible to periodically make a complete pass over the hash table to clean out old calls, however, because this could freeze the system for a moment. Instead, I use a method that makes use of the representation of the hash table – whenever I search the table for a particular XID, I always traverse the complete list of values that hash to the same bucket, and remove all out-of-date calls.

As long as the hash function does a reasonably good job of dispersing the XIDs into different buckets, the expected number of pending calls per bucket is low (typically one or two) and so performing garbage collection on one bucket is fast. Similarly, if the hash function is effective, then no piece of garbage will have to wait very long before being collected. In the worst case, garbage may accumulate in buckets that are rarely accessed, but this does not appear to happen in practice. I monitored the memory use of `nfsdump` over several weeks of running and noticed no long-term increase in size.

versions of NFS, it is apparently rarely used.

3.5 Difficulties of Analyzing NFS Traces

As mentioned in earlier NFS trace studies, the analysis of passive NFS traces presents several challenges, particularly when trying to relate these analyses to those done on kernel-based traces [50]. This subsection provides an overview of the challenges of comparing NFS traces with kernel-based traces:

- Details of the underlying file system are hidden.
- The NFS interface is different than the canonical FS interface.
- Client-side caching skews the observed workload.
- Some NFS calls and responses are lost.
- NFS calls may be reordered *en route*.

Just as the idea of passive NFS tracing is not new, methods for dealing with most of these problems are not new, and have been described at length elsewhere [9, 20, 33, 50], and I will discuss these methods only briefly. The problem of reordered calls, however, is something I have not seen addressed in the literature, and so I will discuss my approach in detail in Chapter 5.

3.5.1 The Underlying File System is Hidden

Much information about the underlying server file system is not revealed by any analysis of the messages between NFS clients and servers. For example, it is impossible to learn much about files or directories that are never accessed via NFS or any information about the actual on-disk layout of the files and directories. It is also impossible to reconstruct the complete file system hierarchy *a priori* – but as shown in earlier studies [9, 50], it is possible to reconstruct the active parts of the hierarchy on-the-fly by learning the relationship between directories and their contents as revealed by `lookup` calls and responses. This method is effective for my traces; after processing several minutes of traces, the probability is high for each file or directory encountered in the trace that its parent directory has already been seen. Unfortunately the files and directories that are most frequently accessed tend to also be those that are cached by the client, and therefore these are often the files that I am least likely to learn about.

Another problem with tracing at this level is that little information about the internal state of the server (such as the state and contents of its cache) is revealed. Because I am interested in characterizing the server workload, which is independent of the internal state of the NFS server, this does not impede my analyses.

3.5.2 The NFS Interface

Analyses that depend on specific details of the file system interface cannot be done on NFS traces. For example, some analyses require knowledge of file `open` and `close` system calls, but neither of these calls exists in NFS. Methods for augmenting a trace with virtual `open` and `close` events are described in related work [9, 50].

3.5.3 The Effect of Client-Side Caching

The effect of client-side caching is an obstacle to inferring the actual client workload from the workload observed by an NFS server. In order to reduce the latency of client operations, NFS allows clients to cache data and metadata in a weakly consistent manner. This means that some client operations are absorbed by the client cache and never reach the server, or are observed only as `getattr` calls when the client checks whether the data in its cache is still valid.

The strategies employed by some clients can also have the effect that some I/O operations seen by the server might not correspond to actual calls by the client application, but instead are the effect of the client-side operating system performing read-ahead in an attempt to warm its cache with data it anticipates will be requested soon.

The effect of client-side caching and methods to infer actual client workload from the server traffic have been studied extensively [9, 10, 20], but still present new challenges in heterogeneous networks because NFS allows considerable flexibility in how caching is implemented and in the degree of consistency provided to different observers of the file system.

One disadvantage of both application- and syscall-level monitoring is that it hides the effect of the client cache. If the client performs caching then this may have a profound effect on the relationship between calls that are observed at this level compared to the calls that reach the server. For example, imagine that the client performs read-caching and is running an application that repeatedly reads a single small file. The first time the file is read, the storage server must provide the data, but it is possible that all of the subsequent reads will be satisfied by the clients' cache and therefore generate no further read calls for the server. (Instead, the server may see calls generated when the client checks that its cached copy is consistent with the copy on the server.)

The effect of the cache can be simulated via several well-known techniques. This has the advantage of allowing the effect of different cache sizes to be quantified.

Because my research is primarily concerned with server workloads, which are strongly shaped by client-side caching, I do not want to remove the effects of client-side caching and therefore make no attempt to control for them. This implies that direct comparisons of operation counts and the volume of data read and written between my NFS traces and traces from local file systems should show that the NFS traces contain relatively more metadata traffic (to validate the cache contents) but less data actually read (if the cache is often valid).

3.5.4 Lost NFS Calls/Responses

One of the most difficult problems in NFS trace analysis is dealing with lost messages. I observed that this is particularly problematic for busy systems – on one system the mirror port I used to collect the traces simply could not keep up with the maximum arrival rate of NFS calls, and therefore had to drop some. By analyzing the call stream for unexpected holes, and by counting the number of call and response messages that had no corresponding response or call, I estimate that during periods of heavy activity, I lost as many as 10% of the packets, and for short bursts the percentage could have been even higher.

This problem is compounded by the fact that it is impossible to decode an NFS response without seeing the call (as discussed in Section 3.4.1), so losing a call effectively results in losing both the call and its response. For a properly-configured network and adequate hardware to perform the tracing, this problem is greatly reduced but not eliminated.

3.5.5 Reordered NFS Calls

Some analyses, such as determining whether a set of accesses are sequential or random, are sensitive to the order of calls. Out-of-order calls occur when NFS calls are delivered to the server in a different order than they were issued by the application. This reordering is largely an artifact of the conventional NFS architecture, in which separate processes, called `nfsiod`s, issue the actual network calls. Although a client's calls are dispatched to the `nfsiod`s in order, the process scheduler determines the order in which the `nfsiod`s run.

To confirm and measure this effect, I performed an experiment using my own clients and server on an isolated network. When the client ran only one `nfsiod`, no call re-orderings occurred, but as additional `nfsiod`s were added, call reordering became more frequent. In the most extreme case as many as 10% of the packets appear to be reordered, and some calls are delayed by as much as 1 second, although no network errors or packet losses were observed. This effect is more common when UDP is used as the RPC transport, but may also occur with TCP. (I will revisit this issue in Chapter 5.)

3.6 New Analysis Tools for NFS Traces

There already exist a number of tools that make it possible to quantify many aspects of NFS traffic, and system administrators have devised ways to use these tools in order to identify and debug common problems [82]. Unfortunately, most of these techniques are limited in their ability to analyze the activities of specific clients and users. Furthermore, they are tailored to solving specific problems and are difficult to use as a way to explore the wealth of information that can be gleaned from NFS traces.

In this section, I describe `nfsdump` and `nfsscan`, a pair of tools that provide a framework for gathering and analyzing NFS traces in a general and extensible manner. `nfsdump` collects NFS traces, while `nfsscan` scans through the traces gathered by `nfsdump` and creates a set of summary tables that can be analyzed by simple scripts or imported into a spreadsheet or other data analysis tool for further processing. In addition to `nfsdump` and `nfsscan`, I describe `ns_timeagg`, `ns_split`, and `ns_quickview`, three applications that simplify manipulation of the tables created by `nfsscan`.

The advantage of decoupling the process of gathering traces from trace analysis is that the same traces can be analyzed several ways and at different time scales. It also means that there is a clear interface between these tools and that they can be used independently. For example, I have written several other programs to analyze the output of `nfsdump` and used them in my own studies of NFS workloads and usage patterns [24, 25].

Another advantage of the separation of the gathering and analysis of traces is that the data may be anonymized before analysis. `nfsdump` includes an anonymizing post-processor that anonymizes the client and server IP addresses, user and group ID numbers, and file names. All of the data presented in this thesis has been anonymized so that they may be made public. In ordinary use by system administrators, however, this anonymization step would be omitted so that specific and potentially useful identifying information about users, groups, client hosts, and file names would be preserved.

3.6.1 `nfsdump`

My tracing system, `nfsdump`, is similar to nfstrace [9] or tcpdump [89] in general philosophy, but captures more information than either, and, unlike tcpdump, only decodes the NFS protocol. Like tcpdump, the output of `nfsdump` is human-readable text. `nfsdump` uses libpcap [36], the same packet-capture library as tcpdump, and has the ability to read and write raw packet files in the same format as tcpdump or any other tool that uses this format.

The most important functional differences between `nfsdump` and earlier tools are that `nfsdump` captures the effective UID and GID of each call, and properly decodes RPC over TCP (even with jumbo frames).

The raw output of `nfsdump` is a time-stamped record for each NFS call and response observed over the network. These records may also be anonymized by passing them through a separate utility that replaces the sensitive information with anonymized forms of the data.

The anonymization process replaces all user identifiers (UIDs), group identifiers (GIDs), and IP addresses in the traces with arbitrary but consistent values. Filenames and paths are anonymized by pathname elements: directories are anonymized individually, so that if two paths have a common prefix path, their anonymized forms will share a common prefix as well. In addition, filename suffixes are anonymized separately from the rest of the filename, so all files that share the same suffix will have anonymized names that end in the anonymized form of that suffix.

The anonymization process is configurable – the mapping for the anonymization of any value can be overridden. For example, in the anonymization of my own data, I disable the anonymization of many common file and directory names (such as `CVS`, `.inbox` and `.pinerc`) or components (such as `lock`). This is justified by noting that these names contain no sensitive information. For example, if everyone on the system stores their incoming email in a file named `.inbox`, I learn nothing about an individual by learning that they own a file named `.inbox`. The names chosen by an individual, however, are hidden.

There is an obvious flaw in this argument, however: if there is a user whose login name is the same as one of the

file names that I do not anonymize, then the name of their home directory (and the name of any eponymously-named files) will contain the un-anonymized form of their login name. Unfortunately, since the strings that I do not anonymize are commonplace names, even anonymizing these names would do little to hide their identity. For example, if some unfortunate user had a login name of `inbox`, the name of their home directory would still be anonymized in the same way as everyone's `inbox` file, and these files are easy to identify.

I also leave un-anonymized the UID and GID of key users and groups (users such as `root` and `daemon`, and groups such as `wheel`) because it is helpful to be able to identify these entities.

I also treat several suffixes and prefixes (such as #, `,v` and `~`) specially in order to preserve the relationship between the anonymized form of filenames containing these suffixes or prefixes and the anonymized form of the filenames without them. It is also possible to configure the anonymizer to omit all filename, UID, GID, and IP information entirely. This makes some analyses impossible, but still provides a great deal of useful information to file system researchers.

My anonymization approach is vulnerable to a known-text attack by an adversary who has detailed knowledge about the system being traced. For example, an attacker who knows exactly when a particular user accessed a particular file can find the user's anonymized UID and GID and the anonymized filename. Similarly, a chosen-text attack on the system while it is being traced can be used to find the anonymized form of any filename that the attacker can create. Both of these attacks require direct access to, or real-time information about, the system being traced. An insider can exploit his or her ability to use the file system to gain knowledge about the other users of the system, but an outsider who does not have this leverage cannot reverse the anonymization process. I do not use hashing or any other deterministic method to do the anonymization, because that would allow an attacker to perform a known-text attack without access to the traced system, and it would also allow direct comparison of filenames and other information between traces gathered at different sites.

3.6.2 `nfsscan`

The first step of my analysis system, `nfsscan`, writes its data in a format designed to be easy to parse and interface with other tools. This is in contrast with most other tracing tools, which often generate output in an irregular, difficult to parse, or undocumented format – for example, tcpdump 3.7.2 prints file offsets for read and write operations in hexadecimal for NFSv3 calls and decimal for NFSv2 calls, but does not print whether the call uses NFSv2 or NFSv3.

The output of `nfsscan` consists of one or more distinct tables (depending on how `nfsscan` is invoked). These tables can be analyzed by simple scripts or imported into a spreadsheet, database, or other data analysis tool for further processing. These tables include the following information:

- The total number of NFS operations and the number of times each NFS operation is called during each analysis period. The default analysis period is five minutes, but a different analysis period may be specified on the command line.
- The average latency of each type of NFS operation.
- A map of the file system hierarchy and information about each file accessed.

  The file system hierarchy is inferred from the results of lookup, create, mkdir, rename, remove, and link calls. Information about each file is gathered from the responses to getattr and other calls.

By default, `nfsscan` only records the counts for the NFS operations that appear most frequently in a typical workload – read, write, lookup, getattr, access, create, and remove. Additional operations or an alternative list of operations may be specified on the command line.

Unless more information is requested, `nfsscan` creates per-server aggregate statistics in a manner similar to nfsstat. However, it can also subdivide the statistics by any combination of client, UID, GID, and file handle, and it can filter the data based on caller host, UID, or GID.

3.6.3 Helper Applications

The most general and powerful method for analyzing the tables created by `nfsscan` is to import these tables into a database or data analysis package. To simplify some of the more common analyses, however, my toolkit includes several helper applications that make it possible to manipulate the output of `nfsscan` and perform useful analyses from the commandline or by using short shell scripts.

By default, `nfsscan` creates a single table containing aggregate operation counts for each five-minute interval of a trace. The user may specify a shorter or longer time interval, but this is usually unnecessary. A helper application named `ns_timeagg` makes it easy to compute aggregates across rows of the table created by `nfsscan`. For example, if the `nfsscan` time interval is five minutes, `ns_timeagg` can create a new table with a time interval of one hour directly from the output of `nfsscan`. It is typically several orders of magnitude faster to compute aggregates from the tables generated by `nfsscan` than it is to re-run `nfsscan` with different parameters. `ns_timeagg` can also aggregate by client host, UID, or GID.

In contrast to `ns_timeagg`, which combines rows from the table, `ns_split` provides a way to filter rows from the table, throwing away rows that do not match given criteria. A combination of `ns_timeagg` and `ns_split` is usually sufficient to isolate the interesting data. It is also easy to write scripts to manipulate the output of `nfsscan`, `ns_timeagg`, and `ns_split` because the tables are generated in a simple text format.

The helper application `ns_quickview` uses `gnuplot` [94] to create plots from tables of operation counts. `ns_quickview` can either save the `gnuplot` script to file, so that it can be edited before execution, or generate the plots immediately. `ns_quickview` also accepts commands to pass directly to `gnuplot`, which allows the user to customize the plots without editing the scripts. All of the plots in Chapter 4 were created by `ns_quickview`, using command-line options to tell `gnuplot` to create encapsulated postscript of the proper size and with the appropriate font.

I anticipate that for most purposes, `nfsscan` with the default parameters will provide enough information to identify general trends in the data. If the user discovers that a specific period of the trace is particularly interesting, he or she can re-run `nfsscan` with different parameters to extract additional per-client, per-user, per-group, per-file, or per-directory information to build a table to investigate specific questions. It is possible to simply create the full set of tables containing all of the information for the entire trace, but this requires a large amount of processing time and memory, and the resulting tables require a considerable amount of disk space.

3.6.4 Building Benchmarks from Traces

One problem faced by any file system researcher is the difficulty of measuring the effect a change to the file system has on system performance. The many benchmarks currently in widespread use are prone to criticism for making unrealistic assumptions, measuring uncharacteristic workloads, or simply being out of date.

The problems I face in my research are even more difficult. In Chapter 6, I demonstrate a system that performs self-adaptation. Performing a short benchmark on such a system does not allow the system to adapt, but simply repeating the same benchmark makes it too easy for the system to adapt. In order to test the system accurately, I must use a workload that faithfully reproduces the workload observed by the system, including the "noise" in the workload that would not appear in a simple statistic-based workload generator.

One approach to estimating the effect of a change to a system is to simulate the entire system. This is attractive in many cases because it is relatively easy to write a simulator for straightforward processes running in simple systems, and these simulations provide immediate and useful feedback about the approximate effect of a modification. Unfortunately, systems such as contemporary storage servers running file systems such as FFS do not meet the criteria of being straightforward, easy-to-model processes. Their performance is affected by subtle but important interactions, particularly in the allocation, layout, and buffering of the blocks containing the system metadata. To accurately measure the effect of proposed changes on a real file system like FFS, I must run the my changes on the real file system, using a real kernel.

This leads immediately to a new problem – in order to use this methodology, I need a real file system. The usual solution to this problem is to recreate the traced file system from a snapshot of its metadata taken at a known time and then begin replaying from that time in the trace [78]. This method works well when snapshots are available, and when a suitable device is available on which to reconstruct. Unfortunately I have neither. There are no publicly-available snapshots of the systems from which my traces were taken, and even if there were, reconstructing them would require at least 500G of disk space and many hours of set-up time per test.

To solve this problem, I have developed a new method of performing a snapshot-less trace replay that uses the trace itself to reconstruct only the subset of the file system necessary to replay a given section of the trace. I call these ”sub-snapshots”.

In essence, my method is to replay the trace twice, building knowledge about the trace and the file system it reflects and creating a new trace that I can use to perform the actual replay. In the rest of this section, I describe each of these passes.

Satisfying the Trace

The first pass reconstructs what it can about the file system hierarchy, primarily by observing the parameters and responses from `lookup`, `getattr`, `create`, `mkdir`, `rename`, `remove`, and `link` calls. The idea of discovering the file system hierarchy by snooping NFS calls is not new and has been in widespread use since the technique was described by Blaze [9]. Unfortunately, this method is not perfect – it is common for some of the information to be absent from the trace because of missed packets or, more commonly, because it is cached on the client during the trace period and thus never visible in the trace. To compensate for this missing data, I keep track of each file or directory accessed whose metadata is incomplete. Directories for which I have not discovered the name are given an arbitrary name. Unnamed files are either given arbitrary names or discarded. Note that when I am using name-based policies and predictions (described in Chapter 6) I cannot simply invent names. At the same time, however, if I want the workload to be realistic, I cannot ignore the contribution of unnamed files. Fortunately, in the traces I have examined, unnamed files contribute less than 5% of the workload (and almost always much less).

Files or directories for which I cannot infer the parent are attached to the root directory, because from my own experiments I have found that such directories are the most likely to be cached on the client. For example, I rarely see lookups for home directories, because home directories are frequently accessed and rarely invalidated.

Stated in different terms, the primary purpose of the first pass through the trace is to discover a sequence of operations that can be interspersed with the trace in order to satisfy all of the operations successfully completed in the trace (and *not* successfully complete any operations that fail during the trace). A corresponding goal is to have this sequence be reasonably close to the minimal set of operations – I do not want to add any operations (such as creating files that do not appear anywhere in the trace) that are unnecessary to satisfy the trace.

Unfortunately, since often some of the necessary information is absent from the trace (usually because of missing call or response messages), some information about the file system must be guessed through less direct means. This is essentially an exercise in causal logic; I am determining what must have happened in the past (perhaps before the start of the trace) in order to permit the sequence of operations in the trace to succeed.

For example, imagine that the trace begins with a successful read operation. In order for this read to succeed, the file being read must already exist. Since there is no record of the file being created before this point in the trace, then it must have been created before the trace began, and therefore I add to the start of the trace a note that I must create this file before beginning the replay. Fortunately, the response from the NFS read will usually contain information about the file being accessed, including its size, mode, owner, and group, so at this point I know many of the necessary details required to create the file – but unfortunately, I do not yet know its name and parent directory. The only information I have about the identity of the file is its filehandle, an opaque identifier. Therefore I store a reference to this file handle with the “create-file” operation, so that if I later learn the name of this file and/or its parent directory, I can add this information.

Similarly, I may discover directories that must exist in order for the trace to succeed, and therefore make a note

to create them before the start of the trace as well.

There are also cases that require inserting new operations between existing operations in the trace. For example, imagine that the trace contains two consecutive operations that are both reads of the same file, and that the response to the first read is success while the response to the second is a failure indicating that the file no longer exists. In order to resolve this contradiction, I must assume that the file was deleted some time between the first operation and the second, and therefore I insert an appropriate delete. The problem with doing this is that there is no way to determine when the delete occurred – all that can be determined as that it took place between these two operations. The current system places it immediately prior to the second operation.[4]

The output of the first pass is a table of pathnames of each file and directory observed in the trace along with a unique identifier for each object, and the size, mode, and other relevant information necessary to reconstruct the object. The purpose of the new identifier is to provide a convenient substitute for the file handles that is independent of the actual implementation of the file system. (File handles usually encode the mount point and inode numbers, and I cannot ensure that I will get the same values when I reconstruct the file system.)

For the workloads I have examined, the number of files actually referenced during a section of the trace is often a tiny fraction of the contents of the file system, and therefore the subsnapshots can be much smaller than the original file system.

Unfortunately there are occasionally processes that scan large parts of the directory hierarchy, referencing many files but actually using very few. Since most of these files are never actually read or written, I could modify my sub-snapshot builder to recognize this case and treat these files differently (only creating a short or empty file, instead of a file the same size as the original). This would permit my to create sub-snapshots for even larger file systems.

Rewriting the Trace

The second pass through the trace replaces all of the file handles in the trace with the unique identifiers that map to each of the files and directories detected (or created) in the first pass and removes references to files for which no information could be inferred.

Based on the table created after the first pass, I then create a file system that matches the rewritten trace and replay the new trace on that file system. The resulting replay trace is repeatable because it will issue the operations in the same order every time it is run. It is also realistic because every operation in the original trace is reflected in the new trace (along with whatever additional operations are necessary to permit these operations to succeed).

Simply replaying the traces does not necessarily faithfully reproduce the original workload however – there are issues of timing and concurrency that I have not addressed. These issues are discussed in other work [99] and in Chapter 7.

[4] It is an open question whether this choice has a measurable effect on the file system. It would be interesting to see if anything can be guessed about the timing of the missing operations by examining the timing of the operations for which there are observations.

Chapter 4

Contemporary NFS Workloads

In this chapter, I discuss the details of the traces I gathered and show the results of several analyses. I also show how my analysis tools can be used by system administrators to learn more about their systems.

4.1 Overview of the Systems

I gathered traces from servers in several different environments, which I call CAMPUS, EECS, and DEAS. In this section I discuss each of these systems and the workloads they support.

4.1.1 CAMPUS

CAMPUS is a collection of machines serving the computing needs for the bulk of the administration, college, and Graduate School of Arts and Sciences of Harvard University. It handles email and web service for the majority of the students, faculty, and administrators, and has approximately 10,000 active user accounts.

This trace was gathered during the Fall of 2001. The data I use for my analyses are taken from the last week of October 2001, because this is when the workload seems the most regular – earlier in the Fall is too close to the start of the semester, and later in the Fall the Thanksgiving holiday and some gaps in the traces (due to problems with the tracing software) make the data less representative.

CAMPUS storage is distributed over three NFS servers hosting a total of fourteen 53GB disk arrays. Each NFS server has several network interfaces, and they are configured, via `gated`, to appear as fourteen virtual hosts, one per disk array. Because all NFS traffic to a particular disk array uses an IP address unique to that disk array, I can monitor traffic to the individual arrays.

Each of the fourteen disk arrays contains the home directories for a subset of the CAMPUS users. Users are given a default quota of 50MB for their home directories. Users are distributed among the disk arrays according to the first letters of their login names. I gathered long-term traces for two arrays and short-term traces for seven others. I computed summary statistics and general usage patterns for all nine of the traced arrays and found them to be similar. I chose to use the array named home02 for my in-depth analysis.

The connection between the CAMPUS clients and servers is a gigabit Ethernet using jumbo (9000 byte) frames. All clients use NFSv3 over TCP.

The central email, WWW, general login, and computer science course servers mount the disk arrays via different networks. My traces capture only the NFS traffic between the email and general login servers and the disk arrays; I do not capture the NFS traffic generated by serving personal home pages or by students working on computer science

assignments. Statistics provided by CAMPUS system administrators indicate that the subnet that carries the email and general-purpose login traffic represents the vast majority of the total CAMPUS traffic. The CAMPUS traces do not include backup activity.

Unlike many other systems, mail spools are not kept in dedicated partitions; users' inboxes are located inside their home directories. However, the activity of the CAMPUS system is so dominated by email that in some sense the file systems used for home directories can be considered to be dedicated email partitions. Most users of the CAMPUS system access mail remotely via a POP or SMTP server from their personal computer or public access kiosks. My traces do not include this POP and SMTP mail delivery activity as such, but do contain records of the NFS traffic this activity generates.

4.1.2 DEAS and DEAS03

The DEAS and DEAS03 traces were taken from the same server. The DEAS trace is described in earlier work [25]. The DEAS trace was gathered during the Fall of 2002. The DEAS03 trace begins in mid-February 2003 and continues until early March. My analyses focus on the DEAS03 traces from the last week of February.

DEAS is a Network Appliance Filer that serves the home directories for the Division of Engineering and Applied Sciences at Harvard University. Neither the DEAS nor DEAS03 traces include backup activity. There is no standard DEAS/DEAS03 client machine, but the typical client is a PC running GNU/Linux or a SUN Sparcstation running Solaris.

The DEAS workload contains email. Unlike CAMPUS, where email is delivered to the users' home directory, email for DEAS03 users is delivered to a mail spool directory on the same file system but outside of the users' home directory. In addition to email, the DEAS03 workload contains a fairly typical research and development workload.

The connection between the DEAS and its clients is 100 Mbit Ethernet using standard frames. Most clients use NFSv3, but some use NFSv2. All NFS communication between DEAS and its clients is done using UDP.

4.1.3 EECS and EECS03

The EECS and EECS03 traces were taken from the same server. The EECS trace is described in earlier work [24]. Like CAMPUS, the EECS trace was gathered during the Fall of 2001, and once again my analyses focus on late October 2001. The EECS03 trace was gathered starting in mid-February and continuing until mid-March 2003.

EECS is a Network Appliance Filer that serves as the primary home directory server for Harvard's computer science faculty and research groups. The workload is a mix of research, software development, and course work. EECS users can directly mount their home directories onto their workstations via NFS, Samba, or other protocols. In the EECS system, all non-NFS file access protocols are routed through an intermediate host, which converts these protocols to NFS but prevents me from directly identifying the source of the resulting activity.

Although there is no standard EECS client, a typical client is a UNIX or Windows NT machine with more than 128MB RAM and local copies of system software and most utilities. The connection between EECS and its clients is a 100 Mbit Ethernet using standard frames. Most of the EECS clients use NFSv3, but many use NFSv2. All EECS clients use UDP to communicate with the server. The EECS server is used primarily for home directories and shared project and data files. Unlike the CAMPUS and DEAS workloads, the EECS traces contain little email traffic because in the EECS system, email is delivered to a separate server. Some users do file their mail or use mail pre-processors to categorize and re-deliver their email to subdirectories to their home directory, however, so there is some mail-related activity.

There are no user quotas on the EECS system. The aggregate disk capacity of the EECS client machines is much larger than the capacity of the EECS server. Much of the research data used in the EECS environment is stored on other servers; the only items stored on the EECS server are home directories, data that need to be shared, or data that users want to have backed up. The traces do not include any backup activity.

All Hours			
	CAMPUS 10/15–10/28/2001	**DEAS03** 02/17–03/02/2003	**EECS03** 02/17–03/02/2003
Total Ops (1000s)	1091.21 (49.61%)	1178.22 (46.71%)	256.80 (62.60%)
Read Ops	64.54% (48.22%)	48.67% (50.92%)	24.19% (73.88%)
Write Ops	21.30% (58.86%)	15.74% (55.31%)	12.35% (123.73%)
Lookup	5.84% (44.41%)	3.35% (161.57%)	27.06% (71.44%)
Getattr	2.29% (60.69%)	29.16% (49.26%)	3.26% (260.19%)
Access	2.90% (51.43%)	1.40% (119.54%)	19.90% (67.56%)
Create	0.29% (44.63%)	0.22% (68.47%)	0.78% (211.04%)
Remove	0.57% (44.39%)	0.33% (66.18%)	0.77% (85.26%)
Read (GB)	4.80 (44.86%)	8.45 (50.26%)	0.99 (107.75%)
Write (GB)	1.76 (58.91%)	1.74 (63.84%)	0.33 (198.95%)
Read/Write (ops)	3.03	3.09	1.96
Read/Write (GB)	2.73	4.84	3.00

Table 4.1: Average hourly operation count and percentage of each operation (for all of the hours).

The coefficient of variation (the standard deviation divided by the mean, expressed as a percentage) is given in parenthesis.

One difficulty in the analysis of EECS is that EECS actually has two network interfaces, serving two different subnets, and my traces capture the traffic from only one of them. This is a potential source of confusion – for example, a file might be changed due to some action performed by a client on the other subnet. If this happens, then the fact that the file has changed may be observed, but the action that made the change will not.

4.2 Basic Workload Characteristics

The most general question of interest is how many operations of each type a server performs over a given period of time. Table 4.1 shows the total operation counts for two consecutive weeks from each trace and the percentage of the total contributed by each of the most common operations.

Table 4.1 illustrates the diversity of workloads that NFS servers must support:

Read/Write Ratio All of the systems have a read/write ratio of more than one. As illustrated in Table 4.2, during the hours of peak activity, EECS03 and CAMPUS have a read/write ratio of approximately 2.5, while DEAS03 has a data read/write ratio of nearly 4.5.

The EECS03 read/write ratio is particularly interesting because it has changed over time: in October 2001, I observed that EECS had a read/write ratio during peak hours of 1.13 [24]. This change is discussed in Section 4.4.7.

Data and Metadata The EECS03 workload has more metadata requests than reads and writes – there are more requests for information *about* files or directories than requests to read and write the contents of files. In contrast, CAMPUS and DEAS03 are dominated by data operations, particularly `read`, although DEAS03 also contains a large number of getattr calls.

Peak Hours (Monday-Friday, 9:00am-6:00pm)						
	CAMPUS 10/15–10/28/2001		**DEAS03** 02/17–03/02/2003		**EECS03** 02/17–03/02/2003	
Total Ops (1000s)	1677.64	(9.48%)	1874.00	(22.36%)	397.86	(39.28%)
Read	63.54%	(8.53%)	50.02%	(24.31%)	18.19%	(63.51%)
Write	22.31%	(16.69%)	16.82%	(28.92%)	12.34%	(86.71%)
Lookup	5.63%	(8.10%)	3.37%	(29.31%)	26.99%	(33.58%)
Getattr	2.36%	(32.64%)	26.60%	(29.41%)	2.98%	(129.90%)
Access	2.96%	(10.81%)	1.31%	(44.85%)	21.48%	(39.76%)
Create	0.28%	(9.30%)	0.27%	(32.31%)	0.64%	(63.82%)
Remove	0.55%	(9.46%)	0.38%	(34.54%)	0.73%	(72.61%)
Read (GB)	6.91	(7.32%)	13.64	(24.12%)	0.88	(102.60%)
Write (GB)	2.84	(16.76%)	3.05	(34.31%)	0.31	(123.59%)
Read/Write (ops)	2.84		2.97		1.47	
Read/Write (GB)	2.43		4.47		2.67	

Table 4.2: Average hourly operation count and percentage of each operation for the peak hours (Monday-Friday, 9:00am – 6:00pm).

The coefficient of variation (the standard deviation divided by the mean, expressed as a percentage) is given in parenthesis.

4.3 Variation over Time

Like Vogels [90], I observe a large variance of load characterization statistics over time. I also observe, however, that much of this variance can be explained by high-level time-related changes in the workload. This correlation has been observed in many trace studies, but its effects are often ignored.

The most notable change in my traces is the difference between peak and off-peak hours, where peak hours are 9am-6pm on weekdays. The difference between Tables 4.1 and 4.2 illustrates this variation. The dramatic decrease in the coefficient of variation[1] when only peak hours are considered indicates that time is a strong predictor of operation counts, amount of data transferred, and the read-write ratio.

Figures 4.1, 4.2, and 4.3 illustrate the change in total hourly operation count over the course of a day or a week. The upper plot in each figure shows a plot of the operation count for five consecutive weekdays, while the lower plot shows the same data but with the plot for each day overlaid on others in order to make the daily pattern more clear.

I examined a range of possibilities for the "peak" hours for CAMPUS and found that using 9am-6pm resulted in the least variance. The coefficient of variation is reduced by a factor of at least 4 for almost all of the CAMPUS statistics when only these peak hours are considered. The same peak hours were also those that resulted in the least variance for EECS, although there is less correlation between the EECS workload and the "regular" work week. In many cases, the EECS03 load spikes from Figure 4.3 are directly attributable to specific causes, such as software builds, large experiments, or data processing, which are often run via `cron` during off-hours.

Understanding the time-varying nature of a workload is essential for any kind of dynamically-optimizing storage system. Systems that experience genuine quiet periods can use them to rearrange data in anticipation of the next period

[1] The coefficient of variation, or COV, is a measure of variance that is normalized by the sample mean in order to provide a dimensionless measure. It is defined to be the standard deviation divided by the mean, and is usually expressed as a percentage.

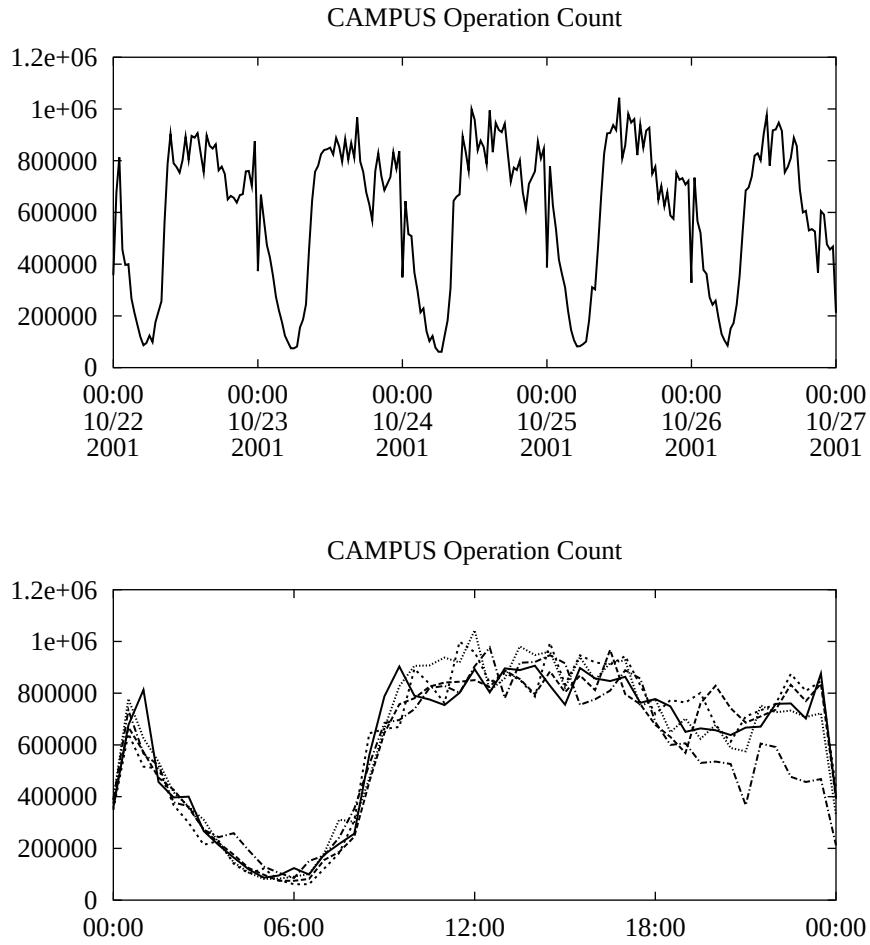


Figure 4.1: Plots of the total operation counts for each 30-minute period of five consecutive weekdays (10/22/01–10/26/01) on CAMPUS.

The upper plot shows the total operation count per half hour period for the entire five days. The plot on the bottom shows the same data, but with the plot for each day superimposed on the others.

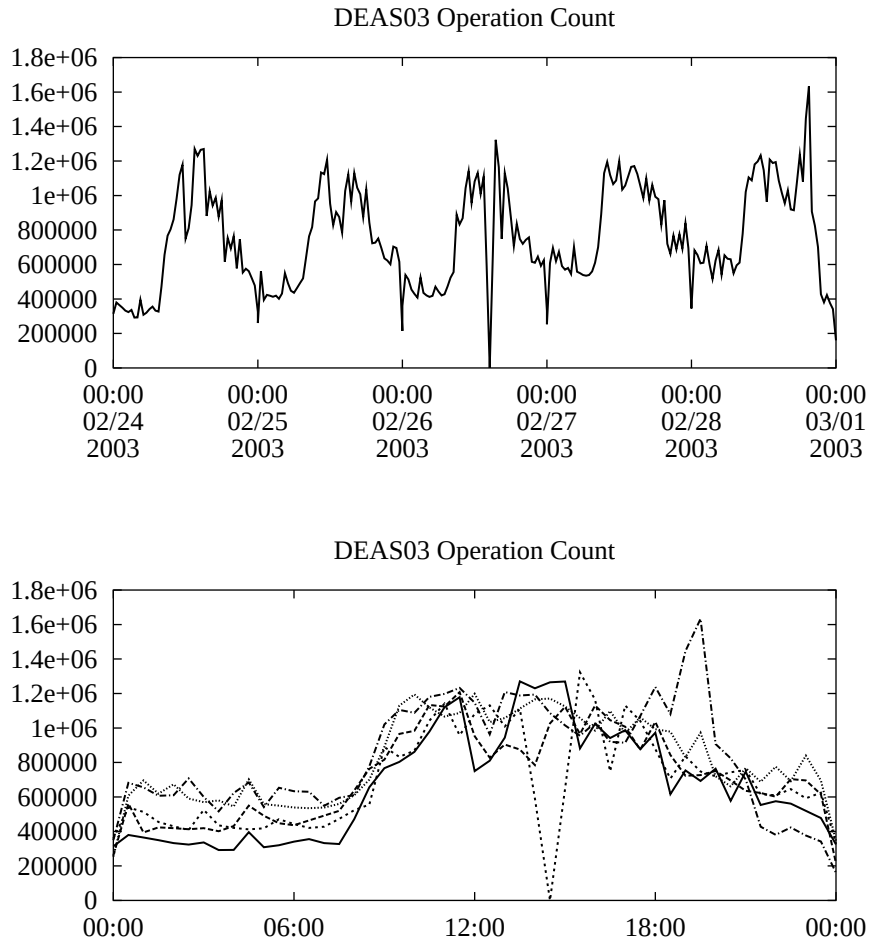


Figure 4.2: Plots of the total operation counts for each 30-minute period of five consecutive weekdays (02/24/03–02/28/03) on DEAS03.

The upper plot shows the total operation count per half hour period for the entire five days. The plot on the bottom shows the same data, but with the plot for each day superimposed on the others.

The apparent gap on the afternoon of 2/26/03 was caused by a temporary problem on the system running `nfsdump`.

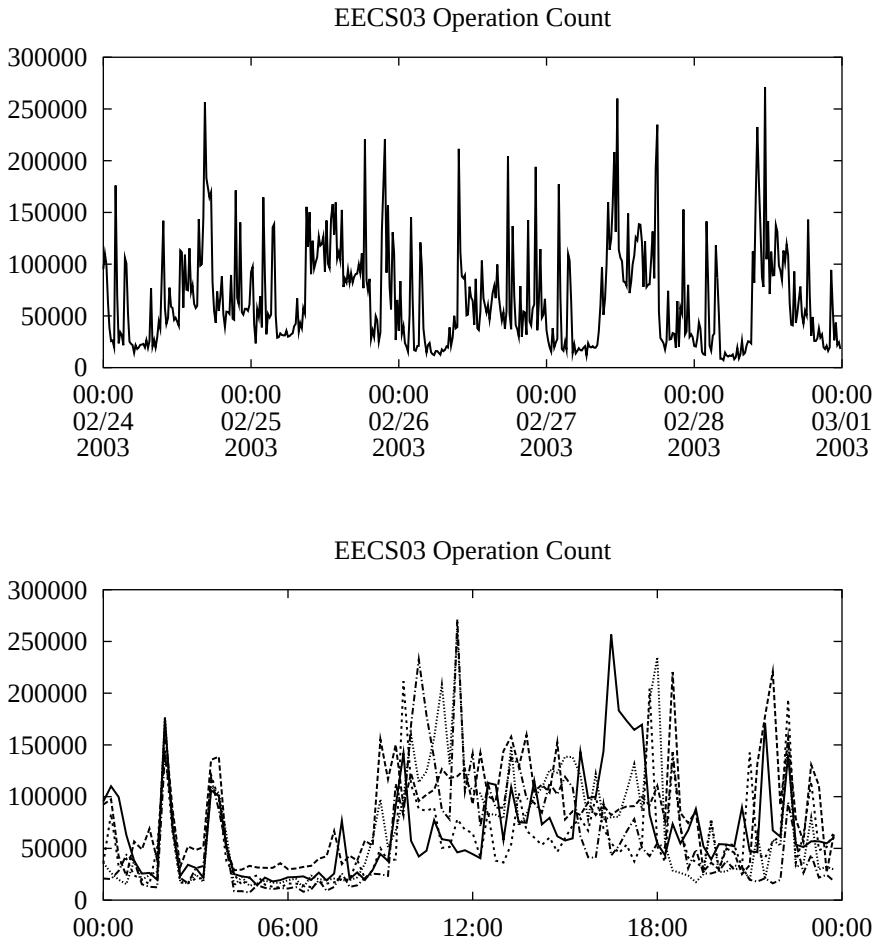


Figure 4.3: Plots of the total operation counts for each 30-minute period of five consecutive weekdays (02/24/03–02/28/03) on EECS03.

The upper plot shows the total operation count per half hour period for the entire five days. The plot on the bottom shows the same data, but with the plot for each day superimposed on the others.

of heavy use. Less radically, if the system is designed to make file layout decisions based upon workload, it is useful to know that there are atypical periods (for example, when backups are running) that might best be ignored, in order to avoid optimizing for an uncommon or non-critical workload.

4.4 Analyses for System Administration

One interesting aspect of the tools I constructed to perform my research is that they also can be used to help solve a wide variety of problems. Administering an NFS environment with hundreds or thousands of clients and users can be a complex task, and when a problem arises there are few tools that can help identify its causes.

In this section, I provide examples of some analyses that are easy to perform with `nfsdump`, `nfsscan`, and related tools. In the example I use five-day traces taken from CAMPUS, DEAS03, and EECS03. Each trace begins at midnight on a Monday and continues until the end of the subsequent Friday. The traces have been processed with `nfsscan`, including per-client and per-user information in addition to the default information.

4.4.1 When is the System Most and Least Busy?

In order to plan maintenance activity or schedule large jobs in such a way to make the best use of an NFS server, it is important to know when the system is busy and whether there are idle periods. I have found that on many systems the load varies in a highly predictable manner according to the time of day and day of week. However, these patterns are not always intuitive – on one system, I found that the system was swamped with requests between 4:00am and 9:00am because several users thought that this would be the time when the system would be most idle, and therefore they each scheduled their largest jobs to run during these hours.

As described in Section 4.3, Figures 4.1, 4.2, and 4.3 show the number of NFS operations per half-hour interval for each trace. These plots are created directly by `ns_quickview`.[2]

CAMPUS and DEAS03 have clear daily rhythms. For all three systems, the operation rate is low in the late evening and early morning, and then climbs rapidly though the morning, reaching a peak at approximately noon, and then staying high throughout the afternoon. The operation rate of DEAS03 decreases rapidly at the end of business hours, but CAMPUS is still busy until almost midnight. Both systems have predictable periods of relatively light activity.

In the plot for the entire week for EECS03, it is difficult to see any particular patterns, but the overlay plot shows that in addition to a general increase of activity during business hours, there are at least three regular daily events that cause load spikes twice in the early morning, and once in the late evening. The regularity of these events suggests that they are probably `cron` jobs scheduled to run in the off-hours.

The overlay plot for CAMPUS shows that the workload is remarkably consistent from one day to the next, particularly in the early morning hours. The workload is lower than average for the weekdays on Friday afternoon and evening. (This is not apparent from the plots as they appear in this paper, because the key that shows which line corresponds to each day has been omitted from these plots. If requested, `ns_quickview` will label each line.)

4.4.2 Which Clients Perform the Most Operations?

Another question is which clients contribute the largest load to the system over time. There are a number of ways to define how much load each client creates (for example, the number of bytes transferred to and from the server) but for this example I will simply assume that the load is simply proportional to the operation count. I can investigate this by

[2] These plots are meant to be as much a demonstration of `ns_quickview` as they are of the data presented.

Client ID	Total Ops	Read	Write	Other	System Use
Total	33639133	7491050	3680255	22467828	
0.0.0.51	19560815	4009983	1489135	14061697	Mail and WWW
0.0.0.130	3106796	38633	21444	3046719	`user1` desktop
0.0.0.53	2847486	460192	580911	1806383	WWW/`user2` desktop
0.0.0.80	1304624	20778	644105	639741	`user3` desktop
0.0.0.84	938404	279071	362768	296565	Cycle server
0.0.0.56	668044	13379	95665	559000	SunRay server
0.0.0.68	614755	128977	99706	386072	Desktop / Cycle server
0.0.0.54	528469	501601	3786	23082	Cycle server
0.0.0.55	519653	492413	3728	23512	Cycle server
0.0.0.57	514069	487531	3705	22833	Cycle server

Table 4.3: Total operation counts for the ten busiest EECS03 clients for the period 2/17-2/21/2003. The client identifiers have been anonymized.

Note that the read/write ratio varies considerably from client to client. For example, the read/write ratio varies from 0.03 (for 0.0.0.80) to 131 (for 0.0.0.57), indicating that these clients are generating very different workloads.

---

using `ns_timeagg` to compute the per-client operation counts. The ten clients with the highest operation counts on EECS03 are listed in Table 4.3.

Once I have identified the busiest clients (or any other clients that appear interesting) I can use `ns_split` to extract the contribution from those clients, use `ns_timeagg` to create a table of the activity of those clients over time, and then use `ns_quickview` to create plots from these tables. Figure 4.4 shows plots generated via `ns_quickview` for the four busiest clients.

Figure 4.4 shows the diversity of workloads contributed by each of the individual clients. Client 0.0.0.51 has a fairly regular pattern that mimics the weekly workload of EECS03 as a whole. This is not surprising given that 0.0.0.51 is the email and WWW server for the department, and therefore the NFS operation count of this machine reflects the number of people using EECS03. Even though mail is not delivered to the EECS03 file system, some activity related to the processing of email (for example, users composing messages or automatic processes filtering spam) does appear in the EECS03 workload.

Clients 0.0.0.130, 0.0.0.53, and 0.0.0.80 are primarily desktop workstations. Client 0.0.0.130 is idle most of the time, but sometimes runs long and fairly NFS-intensive jobs. The workload on client 0.0.0.53 is due primarily to several periodic processes that scan a large number of files and then rebuild the contents of a web site by doing a CVS export to fetch the latest version of every file from a CVS repository [32]. Client 0.0.0.80 is quiet for most of the week, but has an intense period of activity for a few hours.

4.4.3 Which Users Are Busiest?

Finding the busiest users is done in the same manner as finding the busiest clients. The only difference is that the aggregation is per user, instead of per client.

A table of the operation counts for the ten busiest users on EECS03 for the five days 2/17/2003 through 2/21/2003 is shown in Table 4.4. Once again, there is great diversity of workload patterns illustrated in this simple table.

- The busiest "user" is the `www` user. The operation counts for this user are dominated by information requests,

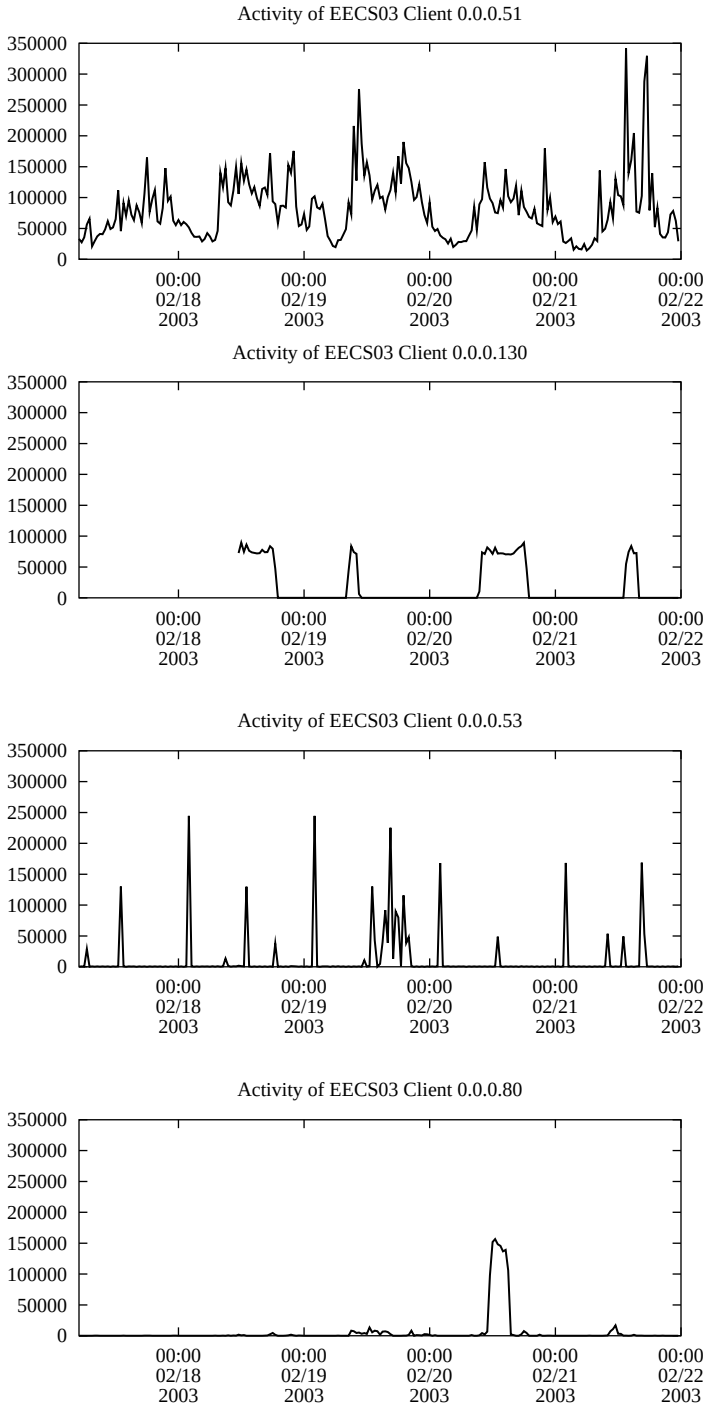


Figure 4.4: Plots of the total operation counts for each 30-minute period of five consecutive weekdays for the four busiest EECS03 clients.

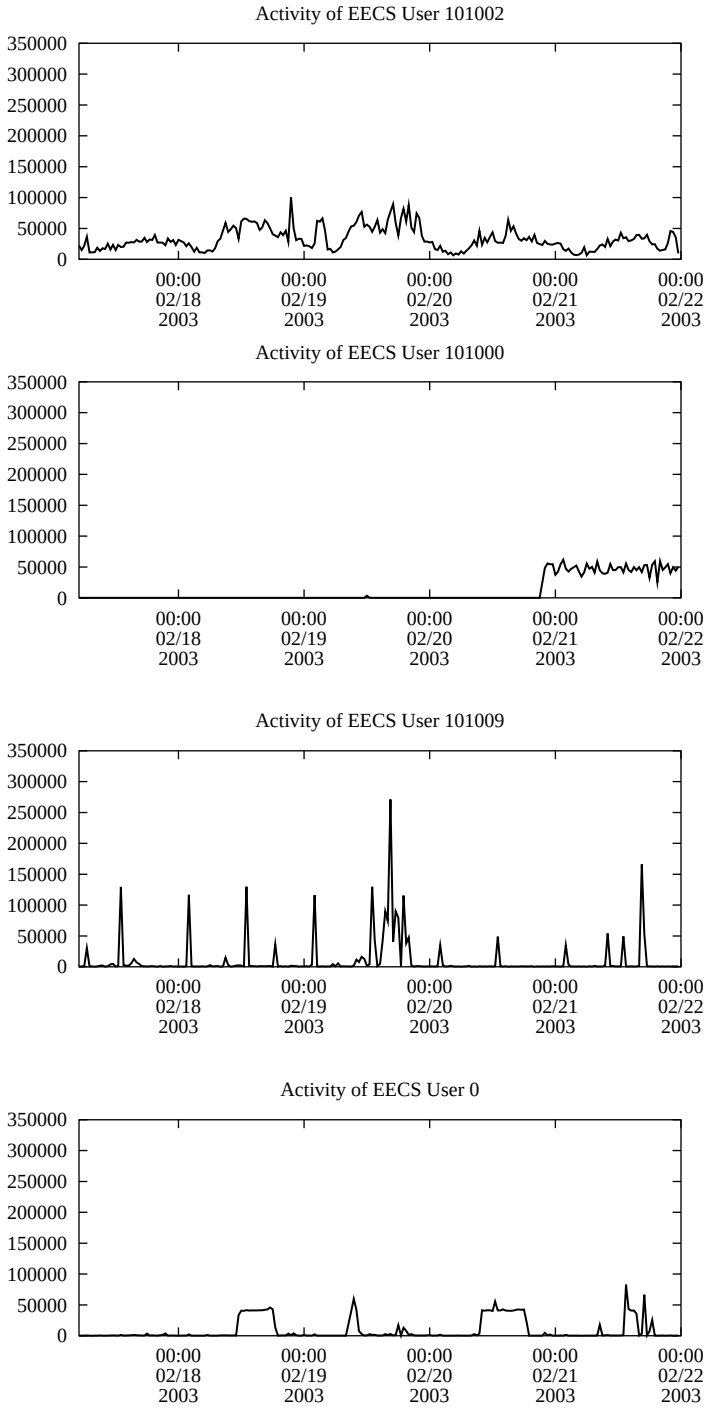


Figure 4.5: Plots of the total operation counts for each 30-minute period of five consecutive weekdays for the four busiest EECS03 users.

User ID	Total Ops	Read	Write	Other	Role
Total	33639133	7491050	3680255	22467828	
101002	7683078	1268777	1024	6413277	`WWW`
101000	2494411	2396508	14917	82986	Grad Student
101009	2461584	384792	496133	1580659	Grad Student (user 2)
0	2020323	16	48	2020259	Superuser (root)
101060	1659601	70247	25455	1563899	Faculty Member (user 1)
101022	1371083	178598	13357	1179128	Faculty Member
101062	1310198	21422	644776	644000	Researcher (user 3)
101035	1131660	96183	1727	1033750	Grad Student
101001	1076060	293780	366905	415375	Grad Student
101034	1008195	138130	817268	5279	Grad Student

Table 4.4: Total operation counts for the ten busiest EECS03 users for the period 2/17-2/21/2003.

Note that the user ID numbers have been anonymized, except for `www` and `root`. Users `user1`, `user2`, and `user3` are the same users referenced in Table 4.3.

Note that the user-level workloads differ considerably. The WWW user workload contains many reads and meta-data operations but very few writes. The workloads for grad students have a read/write ratio less than one, while the faculty members shown here have a much higher read/write ratio. The strange workload of the superuser (user 0), which contains almost not reads or writes, is explored in Section 4.4.6.

---

but also contain some reading and very little writing.

- The second busiest user is a user whose operation count is dominated almost entirely by reading.
- The third busiest user is the user of workstation 0.0.0.53, one of the busiest clients.
- The fourth busiest "user" is `root`, and the operation counts for `root` contain almost no reads and writes. The fact that root accounts for much of the workload is surprising. I investigate the causes of this behavior in Section 4.4.6.
- The fifth busiest user is the user of workstation 0.0.0.130, another one of the busiest clients.

4.4.4 What Files are Busiest?

I can also use `nfsscan` to discover which files and directories are the object of the most operations, and what those operations are. This can be useful to identify hot files or directories, which might benefit from being replicated (if read-only), moved to a local disk (if not shared), or moved to a faster file server.

`nfsscan` can compute per-file operation counts in the same manner as it can create per-user and per-client operation counts. It can also gather additional descriptive information about files, such as their pathname, owner, permissions, and modification times. To save space, particularly when per-user or per-client information is gathered in addition to per-file information, the file description information is stored in a separate table. Both tables can be indexed by file handle, however, so they can be joined if necessary.

4.4.5 What Directories are Busiest?

`nfsscan` can also compute per-directory operation counts. The directory operation count is defined as the sum of the operation counts for all of the files and subdirectories of the directory. This metric is useful for identifying collections of files or directory hierarchies that are the object of many operations even if each individual file is not the object of many. For example, some mail clients store each email message as a separate file in a directory dedicated to this purpose, while others store an entire mailbox (or even a collection of mailboxes) as a single file. In the latter case, it is straightforward to identify and count all of the operations associated with the email of a particular user by keeping track of all operations associated with a single file. In the former case, however, it is much more difficult because the activity is distributed over a directory of files. In this situation, per-directory operation counts are a useful way to aggregate all of this activity into a single table row.

Another example where per-directory operation counts can be helpful is for measuring the total usage of shared directories, such as source code repositories.

Per-directory operation counts can also help identify the busiest subdirectories of a web site. Although it might seem that much of this information can be inferred from the web server logs (which record details about what requests web clients make and how much data the web server sends to its clients), this may lead to inaccurate conclusions. It may be that there are important differences between the traffic from a web server and its clients and the traffic between the web server and the NFS server. For example, if most requests to the web site are for the same pages, these pages may be cached in the web server and therefore these requests will not translate into much NFS activity.

As another example, typical web server logs contain no useful information about what files are accessed each time a dynamically generated web page is visited. As an example, consider a simple web search engine. Search engines typically limit the size of their output to a set number of pages, such as perhaps the twenty "best" matches, and so the size of the data sent to the client is bounded by the amount of space needed to list twenty matches. The total number of disk or file system operations necessary to find the twenty best matches, however, may depend on the total size of the search engine database and the query requested by the client. There are many cases where the total amount of data used to generate such a page might be very different from the amount of data that the web server finally sends to the client.

4.4.6 What is Root Doing on EECS03?

As illustrated in Table 4.4, root was one of the busiest users on EECS03 during the trace period. This is surprising because the EECS03 server is configured to treat root as an untrusted user except from a small number of machines. It is also interesting that nearly all of the NFS operations performed by root during this period are not operations that `nfsscan` ordinarily considers interesting and therefore does not tabulate.

To investigate this mystery, I first used `ns_split` to remove the "trusted" hosts from the table. This made little difference, however, which was a bit troubling because this appears to imply that at least one supposedly untrusted host was able to access the server as root. This, of course, is the kind of situation that haunts the nightmares of NFS system administrators, and so I immediately investigated further.

The plots for the load from client 0.0.0.130 (in Figure 4.4) and the load for the root user (in Figure 4.5) provided a valuable clue. These two plots are very similar in shape (although not in magnitude), suggesting that the activity of this host and the activity of the root account are linked. A quick check using `ns_split` to isolate the contributions of the root user on client 0.0.0.130 confirmed that most of the operations performed by root during the trace period were indeed taking place on that system. Using `ns_split` again to isolate the activity of the user who owns client 0.0.0.130, I observe that at every time scale, plots of the total operation count for this user and for root have an almost identical shape. A further mystery was that the operations performed by the user mostly fell into the category of "normal" operations, unlike virtually all of the operations performed by root.

Now that I knew exactly what host to check and what period of the trace to examine, I re-ran `nfsscan` on a short section of the EECS03 traces, asking it to collect operation counts for all the NFS operations instead of only the default set of "interesting" operations. I discovered that all of the mystery operations were `fsstat` calls. Looking

at the trace itself showed that for every operation invoked by the user, there was one or more corresponding `fsstat` call.

The `fsstat` call requests information about an NFS file system, including how much free space there is and whether there have been any changes to the underlying file system (such as might be caused by a reboot or remount). Because `fsstat` does not access information about specific files or directories, it is permitted to run as root even if root is untrusted, and many NFS client implementations, including the one running on 0.0.0.130, are implemented to treat `fsstat` calls as if they were being performed on behalf of root.

The reason that 0.0.0.130 stands out is that the NFS client used by this host[3] is implemented in a conservative manner and, during the trace period, was also misconfigured, and therefore used `fsstat` incessantly, generating at least one `fsstat` call per ordinary operation. The fact that the primary user of client 0.0.0.130 is also one of the busiest users during this period makes this behavior stand out even more.

Unfortunately for the purpose of my analysis, the user of host 0.0.0.130 noticed that the machine seemed sluggish and requested an upgrade of the OS before I had an opportunity to discover the cause of the excessive `fsstat` calls. However, using my tools I was able to confirm that during later trace periods the number of `fsstat` calls generated by host 0.0.0.130 did return to normal levels.

The behavior exhibited by client 0.0.0.130 raises some administration issues. When confined to a single host, the "extra" `fsstat` calls have no serious implications, but this behavior could become a problem if the EECS administrators had decided to deploy this configuration of this NFS client implementation across the entire department. If *all* of the clients of a server exhibited this behavior, the number of NFS requests would nearly double. On a relatively unloaded server (such as EECS during the EECS03 trace period) this additional load might not cause a problem (or even be noticed), but on a more heavily loaded system, such as DEAS03 or CAMPUS, doubling the number of NFS requests could have an impact on the total system because the additional requests would increase the apparent latency and overall utilization of the network.

4.4.7 What is the Impact of the WWW Server on EECS03?

I noted in section 4.4.3 that the busiest EECS03 user is the WWW server account, and in section 4.4.2 I noted that the busiest client hosts the department web server. I also noted that the EECS03 trace exhibits different load characteristics from an earlier EECS trace and theorized that this might be due to the fact that the earlier trace did not contain accesses to the WWW site. With `nfsscan`, I can test this theory.

Two of the defining characteristics of the EECS traces are a read/write ratio of less than 1, and that requests for metadata outnumber reads and writes significantly. Since the WWW server account is by far the busiest user, and is dominated by reads, a reasonable hypothesis is that the EECS03 workload would resemble the EECS workload if the WWW server traffic were removed. I can test this hypothesis by using `ns_split` to create a new table with all of the contributions from the WWW server account removed.

Removing the WWW server account from the workload does not change the character of the workload very much. The new workload still has a read/write ratio that is much closer to that of EECS03 than EECS. However, I can also note that after the WWW server account is removed, the busiest remaining user (user 10100) also has a notably high read/write ratio. Furthermore, if I use `ns_split` to make a table consisting only of this single user, I can see that the load from this user is distributed over a handful of machines. From my knowledge of the role of each of these machines, and my familiarity with the activities of this user, I know that the workload I am seeing from this user is probably due to this user running several large analyses.

While it is normal for users to run their analyses on these machines, it is arguable that this analysis might be unusual because it does not resemble things I have seen in other traces. I can use `ns_split` again on the original table to compute the operation counts with both the WWW server and this user removed. The resulting operation counts do

[3] The NFS client provided with an early version of MacOS X.

resemble the counts from the earlier EECS traces – the read/write ratio is approximately 1, and requests for metadata (`lookup`, `getattr`, and `access`) dominate requests to read and write data. Therefore, if I can successfully argue that the high operation counts for user 10100 during this period are unusual, then I can claim that the workload of EECS03 is similar to EECS with the addition of the workload contributed by the WWW server.

However, there is an additional wrinkle to the analysis. In the original EECS configuration, the department web pages were hosted on a separate file system, but user home pages and some project home pages were hosted on the EECS NFS server. Therefore, accesses to personal home pages were recorded in the EECS traces, but they had little effect on the total overall workload. Using the per-directory statistics generated by `nfsscan`, I can see that this is no longer true (at least for this trace period). During this trace, project and user home pages contribute the overwhelming portion of the web-oriented traffic. Therefore, I can conclude that moving the departmental home pages back to a separate server would have little impact on the EECS03 workload. Moving user and project home pages to another server, however, would significantly reduce the EECS03 workload.

4.5 Summary

In this chapter I have described the basic characteristics of the workloads illustrated by the traces I have captured and that I will explore in more detail in later chapters of this thesis. The most important finding is that the workloads differ, which implies that workload-specific optimization is a plausible idea.

In Chapter 6 I will explore aspects of workload-specific optimization in more detail, but in the next chapter I will make a brief side excursion and show how read-ahead, a general low-level and relatively workload-independent policy, can be enhanced to recognize useful patterns in the client request stream without any hints from the client. In Chapter 6 I will show how more sophisticated techniques can use trace-based analyses to learn even more and anticipate other aspects of client behavior.

Chapter 5

Improving NFS Read Performance

5.1 Overview

In this chapter I investigate a method to extend the standard FreeBSD NFS server read-ahead heuristic to handle sequential access patterns in the face of modest request reordering. I also introduce a new heuristic that is able to permit the FreeBSD NFS server to recognize sequential patterns within non-sequential access patterns and perform the appropriate read-ahead. The new heuristic dramatically increases the end-to-end read throughput of the system for workloads that include these access patterns.

5.2 Motivation and Background

In an earlier study of NFS traffic, I noted that many NFS requests arrive at the server in a different order than originally intended by the client [24]. In the case of read requests, this means that the sequentiality metric used by FFS is undermined; read-ahead can be disrupted by a small percentage of out-of-order requests, even when the overall access pattern is overwhelmingly sequential. This is a cause for concern because read-ahead is essential to good read performance [77].

The standard read-ahead heuristic used by the FreeBSD 4.x NFS server is to keep a counter of the number of sequential accesses for each file. Every time a read operation is encountered, the starting offset is compared to the ending offset of the previous operation. If the two offsets are identical, then the two accesses are sequential and the counter is incremented.[1] If the accesses are not sequential, then the counter is reset to a small number.[2] This counter is used to determine how much read-ahead to perform; the higher the number, the more read-ahead is requested.

Even though the sequentiality count can drop back to one if an out-of-order request is encountered, a few such requests do not necessarily spell complete disaster for read performance. If the sequentiality count has been high, then read-ahead has been occurring, and therefore the next several reads may also be satisfied from cache (or possibly the disk's track buffer) and thus incur no performance penalty. If these reads are sequential, then the sequentiality score will climb again, and therefore read-ahead will begin again before the cache is empty. The problem occurs when out-of-order requests occur regularly enough that the sequentiality score never has a chance to rise high enough to cause enough read-ahead to allow the heuristic time to recover. In my analysis of NFS request streams I observed that as many as 5-10% of the requests are reordered, which would typically limit the sequentiality score to 10-20.

[1] In the FreeBSD implementation, the counter is limited to a maximum value 127.

[2] If the counter was greater than zero, it is reset to one. If the counter was zero, however, it remains zero.

I have developed two new heuristics to reduce the effect of out-of-order requests:

- The first is a general method, which I call *Reorder*. It is described in an earlier work [24].

  In this method, I attempt to rearrange the requests into sequential order with the constraint that I may not exchange the order of any two requests that are separated by more than time t. The span of time t is called the *reorder window*. If such a rearrangement is possible, then I consider the sequence to be sequential. For my workloads, a t of 5-10 milliseconds is large enough to handle the reordering due to `nfsiod`s.[3]

  Note that the goal is to keep t as small as possible. A very large t (for example, if t is sufficiently long that the entire sequence fits within the reorder window) permits almost any sequence that makes requests from a set of contiguous regions in a file to appear sequential.

  The drawback of the reorder window method is that it is impossible for the the system to determine that a request is *not* sequential until time t has elapsed. If a request sequence is completely sequential, then this will be evident immediately, but identifying completely sequential sequences is not the problem that I am trying to solve. The standard heuristic already accomplishes that.

- The second method, which I call *SlowDown*, is a method that makes use of the existing FFS sequentiality metric and read-ahead code as the basis for its implementation. Section 5.4 provides a detailed description and benchmark of this method.

  Unlike *Reorder*, *SlowDown* does not use a reorder window or depend on time in any way. Like the default method, the only history it keeps is a current position and a sequentiality score. Therefore it can always be evaluated immediately.

The difference between these heuristics is the locality metric I am willing to relax in order to achieve sequentiality. *Reorder* assumes that if the pattern truly is sequential, then reordered requests will arrive within a short period of time of their "correct" order. *SlowDown* assumes that if the pattern is sequential, then the reordered requests will be logically "close" to each other – the reordered requests will be for adjacent or nearby logical blocks, rather than blocks at arbitrary positions in the file.

5.3 Measuring NFS Read Performance

Despite the evidence from my analysis of several long-term NFS traces that *SlowDown* should enhance read performance, the actual benefit of this algorithm proved quite difficult to quantify. In my efforts to measure accurately the impact of my changes to the system, I discovered several other phenomena that interacted with the performance of the disk and file system in ways that had far more impact on the overall performance of the system than my improvements. This section is devoted to discussing these effects and how to control for them.[4]

5.3.1 The Testbed

The server used for all of the benchmarks described in this chapter is a Pentium III system running at 1 GHz, with 256 MB of RAM. The system disk is an IBM DDYS-T36950N S96H SCSI-3 hard drive controlled by an Adaptec 29160 Ultra160 SCSI adapter. The benchmarks are run on two separate disks: a second IBM DDYS-T3690N S96H drive (attached to the Adaptec card), and a Western Digital WD200BB-75CAA0 drive (attached to a VIA 82C686 ATA66 controller on the motherboard). The server has two network interfaces: an Intel PRO/1000 XT Server card running at 1 Gb/s and an 3Com 3c905B-TX Fast Etherlink XL card running at 100 Mb/s.

[3] Jonathan Ledlie implemented this method and constructed the experiment that showed that 5-10 milliseconds are suffi cient for the EECS and CAMPUS traces.

[4] In truth, aspects of this discussion may be more interesting and useful to the reader than the description of my changes to the NFS server.

The clients are Pentium III systems running at 1 GHz, with 1 Gigabyte of RAM, and two network interfaces: an Intel PRO/1000 XT Server card running at 1 Gb/s, and an Intel Pro 10/100B/100+ Ethernet card, running at 100 Mb/s. The 100Mb/s interfaces are for general use, while the 1Gb/s interfaces are used only by the benchmarks.

The gigabit Ethernet cards are connected via a NetGear GSM712 copper gigabit switch. The switch and the Ethernet cards use 802.3x flow control and the standard Ethernet MTU of 1500 bytes. The raw network bandwidth achievable by the server via TCP over the gigabit network is 49 MB/s. This falls far short of the theoretical maximum, but approaches the DMA speed of the PCI bus of the server motherboard, which I measured at 54 MB/s using the `gm_debug` utility provided by Myrinet to test their network cards. (A Myrinet card was installed in the server long enough to run this benchmark, but was not in the system for the other tests.) Only the benchmark machines are attached to the gigabit switch.

All systems under test run FreeBSD 4.6.2. The FreeBSD kernel was configured to remove support for pre-686 CPUs and support for hardware devices not present in my configuration, but I made no other customizations or optimizations beyond what is explicitly described in later sections of the paper.

For all tests, the server runs eight `nfsd`s instead of the default four, and the clients run eight `nfsiod`s instead of the default four. Choosing the optimal number of `nfsd`s and `nfsiod`s is a matter of experimentation. In general, adding daemons allows a higher the level of concurrency, but beyond a certain level the effective level of concurrency decreases due to contention. The optimal numbers vary from workload to workload and from system to system. I found that eight gave the best results for almost every benchmark (including benchmarks of the unmodified system). Therefore I assume that a diligent system administrator would have discovered this fact and increased the numbers accordingly. Since I want my benchmarks to be fair (i.e., not compare an obviously sub-optimal configuration to my experimental system), I chose the best configuration I found.

Note that the clients have more memory than the server. This may seem counterintuitive at first, but is actually not unusual. Many storage servers have a lot of memory, but they also have many clients. Since there tends to be relatively little overlap in the data accessed by each client, the demand on the server cache grows almost linearly with the number of clients [35]. Since in my experimental setup I only have a small number of clients, it is appropriate to use a relatively small amount of server memory.

Some storage servers do provide large amounts of memory, but under the NFS protocol this space is often wasted because of the lack of coordination between the client and server caches in the NFS protocol. It is often the case that the server cache echos the contents of the client cache and almost all cache hits come from the client rather than the server [54, 95].

For this particular benchmark, however, the amount of memory in the client and servers is largely irrelevant (as long as it is large enough to permit the file system to perform read-ahead). The benchmarks, as described in Section 5.3.2, take special care to eliminate the effect of the cache and focus on end-to-end performance of reads from disk.

5.3.2 A Micro-Benchmark for NFS Reads

The aspect of system performance that I wish to measure is the sustained bandwidth of concurrent read operations; the sustained bandwidth I can obtain *from disk* via the file system when several processes concurrently read sequentially through large files. In this section I describe the simple benchmark that I have designed for this purpose. Although it is simple, the results from my benchmark illustrate the complexity of tuning even simple behaviors of the system.

Running the Benchmark

Before running the benchmark, I create a testing directory and populate it with a number of files: one 256 MB file, two 128 MB files, four 64 MB files, eight 32 MB files, sixteen 16 MB files, and thirty-two 8 MB files. I fill every block in these files with non-zero data, to prevent the file system from optimizing them as sparse files.

The benchmark loops through several different numbers of concurrent readers:

For each n (1, 2, 4, 8, 16, 32)

- For each file of size $256/n$ MB, create a reader process to read that file. This process opens the file, reads through it from start to end, and then closes the file. Start all these processes running concurrently.
- Wait until the last reader process has finished. Record the time taken by each reader. The number of MB read divided by the time required for the last reader to finish gives the effective throughput of the file system.

During the first iteration, a single reading process will be created, which will read through the 256 MB file. In the second iteration, two reading processing will concurrently read through different 128 MB files. In the final iteration, 32 reading processes will concurrently read different 8 MB files.

For all of the timed benchmark results shown in this paper, each point represents the average of at least ten separate runs. Unless otherwise mentioned, the standard deviation for each set of runs is less than 5% of the mean (and typically much less).

Defeating the Cache

I wish to benchmark the speed that the file system can pull data from the disk (either explicitly or via read-ahead). To ensure that I measure this (instead of how fast I can access data already in the cache), I must make sure that the data I read are not already in the cache before the benchmark begins.

For my particular setup, every set of files contains 256 MB, and I use a separate set of files for each of the six test cases, so a complete iteration of the benchmark requires reading 1.5 GB. Because my clients have 1 GB of RAM and the server has only 256 MB of RAM and files are not re-read until 1.25 GB (5×256 MB) of other data has been read, none of the data will survive in the cache long enough to have an effect, at least with my current OS. If my clients and servers engaged in cooperative caching, or used a caching mechanism intelligent enough to recognize the cyclic nature of my benchmarks, I would have to take more care to ensure that none of the data are read from cache.

Other techniques for ensuring that the data are flushed from the cache include rebooting each client and server between each iteration of the benchmark, unmounting and remounting the file systems used by the benchmark, and reading large amounts of unrelated data until they fill the cache. I experimented with each of these and found that they made no difference to the benchmark results, so I am confident that caching did not influence my results.

5.3.3 Benchmarking Traps

My initial attempt to measure the effect of changes to the FreeBSD NFS server was frustrating. My experimental algorithms (which will be described in Sections 5.4 and 5.5) are easy to implement, and the diagnostic instrumentation I added to monitor my algorithms confirmed that they were working as intended. However, the results from my benchmarks were confusing and sometimes contradictory. Different runs on the same hardware could give very different results, and the results on different hardware were often puzzling. I anticipated that the effect of my changes would be relatively small, but I did not expect it to be overwhelmed by unexpected effects. Therefore, before proceeding with the benchmarks, I decided that a more interesting course of action would be to investigate and expose the causes of the variation in my benchmark and see if I could design my experiments to control for them.

ZCAV Effects

The impact of the ZCAV effect (described in Section 2.1.3) is illustrated in Figure 5.1, which shows the results of running the same benchmark on different areas of a local disk. For my benchmarks, I divide each of my test disks into four partitions of approximately equal size, numbered 1 through 4. The files in tests `scsi1` and `ide1` are positioned in the outer cylinders of the SCSI and IDE drives, respectively, while the `scsi4` and `ide4` test files are placed in the

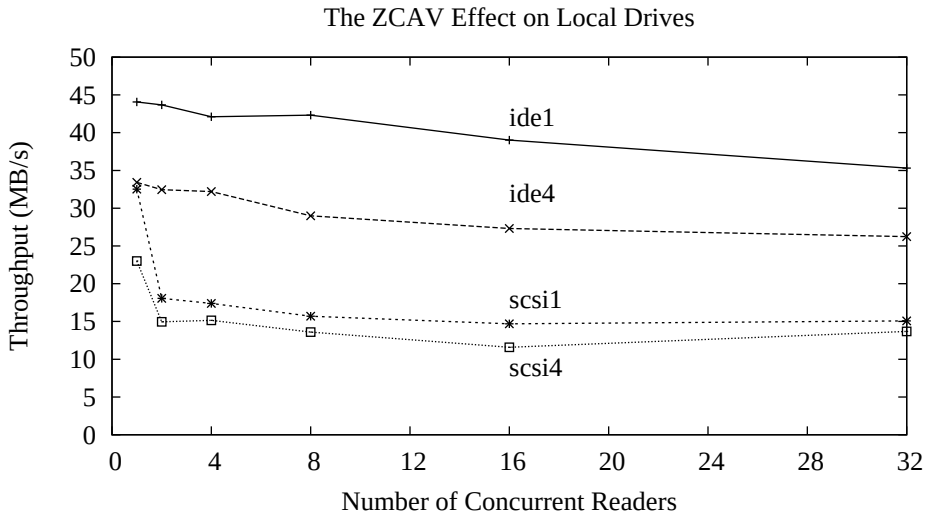


Figure 5.1: The effect of ZCAV on the speed of reading from local drives.

The `scsi1` and `ide1` partitions use the outermost cylinders of the disk, while the `scsi4` and `ide4` use the innermost. As a result, the transfer rates for `scsi1` is higher than `scsi4` and `ide1` is higher than `ide4`.

---

inner cylinders. For both disks, it is clear that there is a relationship between the speed of the benchmark and the zone in which the data are stored. The effect is especially pronounced for the IDE drive. The SCSI drive shows a weaker effect, but, as I show later, this is because there is another effect that obscures the ZCAV effect for simple benchmarks with concurrent readers on the SCSI disk. For a single reader, the effect is large even on the SCSI disk. For both drives the ZCAV effect is more than enough to obscure the impact of any small change to the performance of the file system.

Tagged Queues

One of the features touted by SCSI advocates is the availability of *tagged command queues* (also known as *tagged queues*). This feature permits the host to send several disk operation requests to the disk and let the disk execute them asynchronously and in whatever order it deems appropriate. Modern SCSI disks typically have an internal command queue with as many as 256 entries. Some recent IDE drives support a feature conceptually identical to tagged command queues, but my IDE drive does not.

The fact that the tagged command queue allows the disk to reorder requests is both a boon for ordinary users and a source of headaches for system researchers. With tagged queues enabled, the disk may service its requests in an order different from the one in which they arrive at the disk, and its heuristics for reordering requests may be different from what the system researcher desires (or expects). For example, many disks will reorder or reschedule requests in order to reduce the total power required to position the disk head. Some disks even employ heuristics to reduce the amount of audible noise they generate, because many users would prefer to have a quiet computer than one that utilizes the full positioning speed of the disk. The same model of disk drive may exhibit different performance characteristics depending on the firmware version and on whether it is intended for a desktop or a server.

The SCSI disk in my system supports tagged queues, and the default FreeBSD kernel detects and uses them. I instrumented the FreeBSD kernel to compare the order that disk requests are sent to the disk to the order in which they are serviced, and I found that when tagged queues are disabled, the two orders are the same, but when tagged queues are enabled, the disk does reorder requests. Note that this instrumentation was disabled during my timed benchmarks.

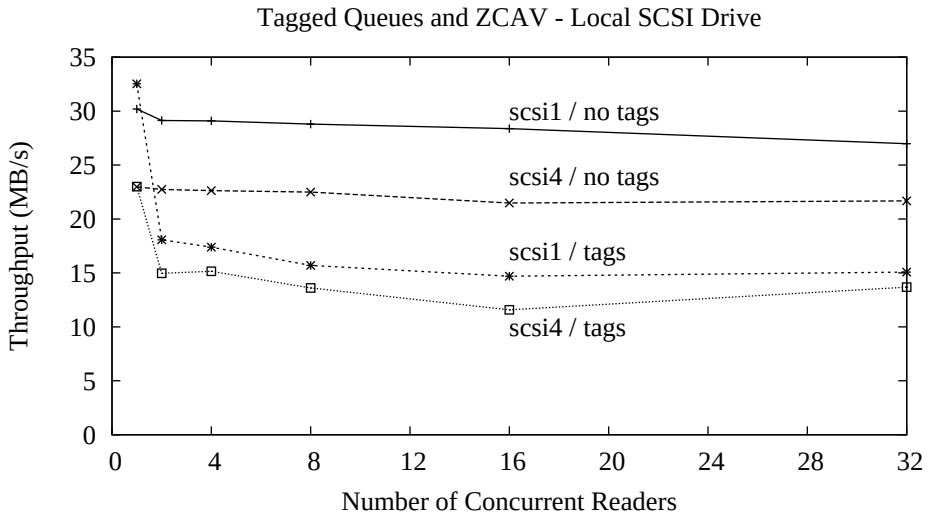


Figure 5.2: The effect on tagged queues and ZCAV on local SCSI drives.

On my test system, disabling tagged queues improves transfer rates substantially for concurrent processes performing long sequential reads.

To explore the interaction between the FreeBSD disk scheduler and the disk's scheduler, I ran a benchmark with the tagged queues disabled. The results are shown in Figure 5.2. For my benchmark, performance is significantly increased when tagged queues are disabled. When tagged queues are enabled, performance for the default configuration has a dramatic spike for the single-reader case, but then quickly falls away for multiple readers. With the tagged command queue disabled, however, the throughput for multiple concurrent readers decreases slowly as the number of readers increases, and is almost equal to the spike for the single reader in the default configuration. For example, the throughput for `scsi1` levels off just above 15 MB/s in the default configuration but barely dips below 27 MB/s when tagged command queues are disabled.

There is no question that tagged command queues are effective in some situations. For my benchmark, however, the kernel disk scheduler makes better use of the disk than the on-disk scheduler.[5] This is undoubtedly due in part to the fact that the geometry the disk advertises to the kernel does, in fact, closely resemble its actual geometry. This is not necessarily the case – for example, it is not the case for many RAID devices or similar systems that use several physical disks or other hardware (perhaps distributed over a network) to implement one logical disk. In a hardware implementation of RAID, an access to a single logical block may require accessing several physical blocks whose addresses are completely hidden from the kernel. In the case of SAN devices or storage devices employing a dynamic or adaptive configuration the situation is even more complex; in such devices the relationship between logical and physical block addresses may be arbitrary, or even change from one moment to the next [93]. For these situations it is plausible that the device, given its better knowledge of the details of the system, could do a better job of scheduling the requests than the kernel, but it certainly depends on both the workload and the device characteristics.

Note that even for ordinary single-spindle disks, there is a small amount of block re-mapping due to bad block substitution, and therefore the disk knows something about the position of each block that the kernel does not. However, such blocks almost always constitute a small fraction of the total number of disk blocks and block remapping is

[5] Kirk McKusick, architect and author of the FFS, notes that he has observed and measured this phenomena in other situations as well, although it apparently has not been documented.

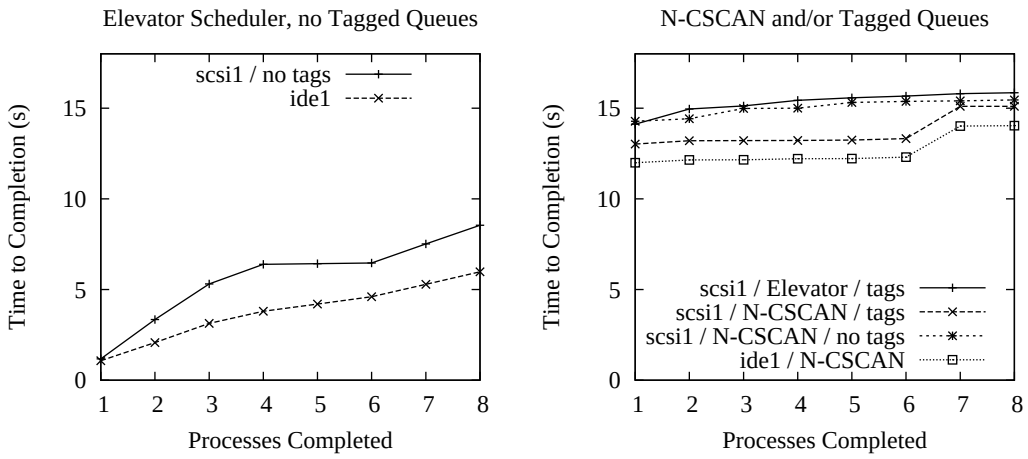


Figure 5.3: The effect of tagged queues and the disk scheduling algorithm on the distribution of average time required to complete a given number of reading processes.

In each run, eight processes are started at the same time, and each reads the same amount of data. Each point on these plots represents the average of 34 runs. Processes run more quickly with the Elevator scan, but the last process takes 6-7 times longer to complete than the first. With the N-CSCAN scheduler, there is less variation in the distribution of running times, but all of the jobs are much slower.

done in such a manner that it has a negligible effect on performance.

Disk Scheduling

The FreeBSD disk scheduling algorithm, implemented in the `bufqdisksort` function, is based on a cyclical variant of the SCAN or elevator scan algorithm, as described in the BSD 4.4 documentation [44]. This algorithm can achieve high sustained throughput and is particularly well suited to the access patterns created by the FFS read-ahead heuristics. Unfortunately, if the CPU can process data faster than the I/O subsystem can deliver it, then this algorithm can create unfair scheduling. In the worst case, imagine that the disk head is positioned at the outermost cylinder, ready to begin a scan inward, and the disk request queue contains two requests: one for a block on the outermost cylinder, requested by process α, and another for a block on the innermost cylinder, requested by process β. The request for the first block is satisfied, and α immediately requests another block in the same cylinder. If α requests a sequence of blocks that are laid out sequentially on disk, and does so faster than the disk can reposition, its requests will continue to be placed in the disk request queue before the request made by β. In the worst case, β may have to wait until α has scanned the entire disk. Somewhat perversely, an optimal file system layout greatly increases the probability of long sequential disk accesses, with a corresponding increase in the probability of unfair scheduling of this kind.

This problem can be reduced by the use of tagged command queues, depending on how the on-disk scheduler is implemented. In my test machine, the on-board disk scheduler of the SCSI disks is in effect more fair than the FreeBSD scheduler. In this example, it will process β's request before much time passes.

The unfairness of the elevator scan algorithm is also somewhat reduced by the natural fragmentation of file systems that occurs over time as files are added, change size, or are deleted. Although FFS does a good job of reducing the impact of fragmentation, this effect is difficult to avoid entirely.

The primary symptom of this problem is a large variation in time required by concurrent readers, and therefore this behavior is easily visible in the variance of the run times of each subprocess in my simple benchmark. Each subprocess starts at the same time, and reads the same amount of data, so intuition suggests that they will all finish

at approximately the same time. This intuition is profoundly wrong when the default scheduler is used. Figure 5.3 illustrates the distribution of the individual process times for runs for the benchmark that runs eight concurrent processes, each reading a different 32 MB file. Note that the cache is flushed after each run. The plot of the time required to complete 1 through 8 processes using the elevator scan scheduler on the `ide1` partition shows that the average time required to complete the first process is 1.04 seconds, while the second finishes in 1.98 seconds, the third in 2.94, and so on until the last job finishes after an average of 5.97 seconds. With tagged queues disabled, a similar distribution holds for the `scsi1` partition, ranging from 1.18 through 8.54 seconds, although the plot for `scsi1` is not as straight as that for `ide1`. The difference between the time required by the fastest and slowest jobs is almost a factor 6 for `ide1`, and even higher for `scsi1`.

N-step CSCAN (N-CSCAN) is a fair variation of the Elevator scheduler that prohibits changes to the schedule for the current scan – in effect, it is always planning the schedule for the *next* scan [22]. The resulting scheduler is fair in the sense that the expected latency of each disk operation is proportional to the length of the request queue at the time the disk begins its next sweep. Only a small patch is needed to change the current FreeBSD disk scheduler to N-CSCAN. I have implemented this change, along with a switch that can be used to select the disk scheduling algorithm at runtime. As illustrated again in Figure 5.3, this dramatically reduces the variation in the run times for each reader process: for both `ide1` and `scsi1`, the difference in elapsed time between the slowest and the fastest readers is less than 20%.

Unfortunately, fairness comes at a high price: although all of the reading processes make progress at nearly the same rate, the overall average throughput achieved is less than half the bandwidth delivered by the unfair elevator algorithm.[6] In fact, for these two cases, the *slowest* reading process for the elevator scan algorithm requires approximately 50% less time to run than the *fastest* reading process using the N-step CSCAN algorithm. For this particular case, it is hard to argue convincingly in favor of fairness. In the most extreme case, however, it is possible to construct a light workload that causes a process to wait for several minutes for the read of a single block to complete. As a rule of thumb, it is unwise to allow a single read to take longer than it takes for a user to call the help desk to complain that their machine has crashed. At some point human factors can make a fair division of file system bandwidth as important as overall throughput.

Also shown in Figure 5.3 is the impact of these disk scheduling algorithms on `scsi1` when the tagged command queue is enabled. As described in Section 5.3.3, the on-disk tagged command queue can override many of the scheduling decisions made by the host disk scheduler. In this measurement, the on-disk scheduling algorithm appears to be fairer than N-step CSCAN (in terms of the difference in elapsed time between the slowest and fastest processes), but even worse in terms of overall throughput.

Although the plots in Figure 5.3 are relatively flat, they still exhibit an interesting quirk – there is a notable jump between the mean run time of the sixth and seventh processes to finish for N-CSCAN for `ide1` and `scsi1` with tagged queues disabled. I did not investigate this phenomenon.

The tradeoffs between throughput, latency, fairness and other factors in disk scheduling algorithms have been well studied and are still the subject of research [13, 71]. Despite this research, choosing the most appropriate algorithm for a particular workload is a complex decision and, apparently, a lost art. I find it disappointing that modern operating systems generally do not acknowledge these tradeoffs by giving their administrators the opportunity to experiment and choose the algorithm most appropriate to their workload.

TCP versus UDP

The remote procedure call (RPC) mechanism which is used by the NFS definition is SUN RPC. Early implementations of SUN RPC used UDP for its transport layer, in part because of the simplicity and efficiency of the UDP protocol. Around the same time as NFS version 3 appeared many vendors began offering TCP-based SUN RPC. Implementing

[6] The global benefit from local unfairness has been noted in other contexts, such as the "shortest connection first" heuristic for servicing web requests [18]. For web services, giving "short" requests a higher priority has little impact on the performance of "long" requests but can significantly improve the overall throughput of the system.

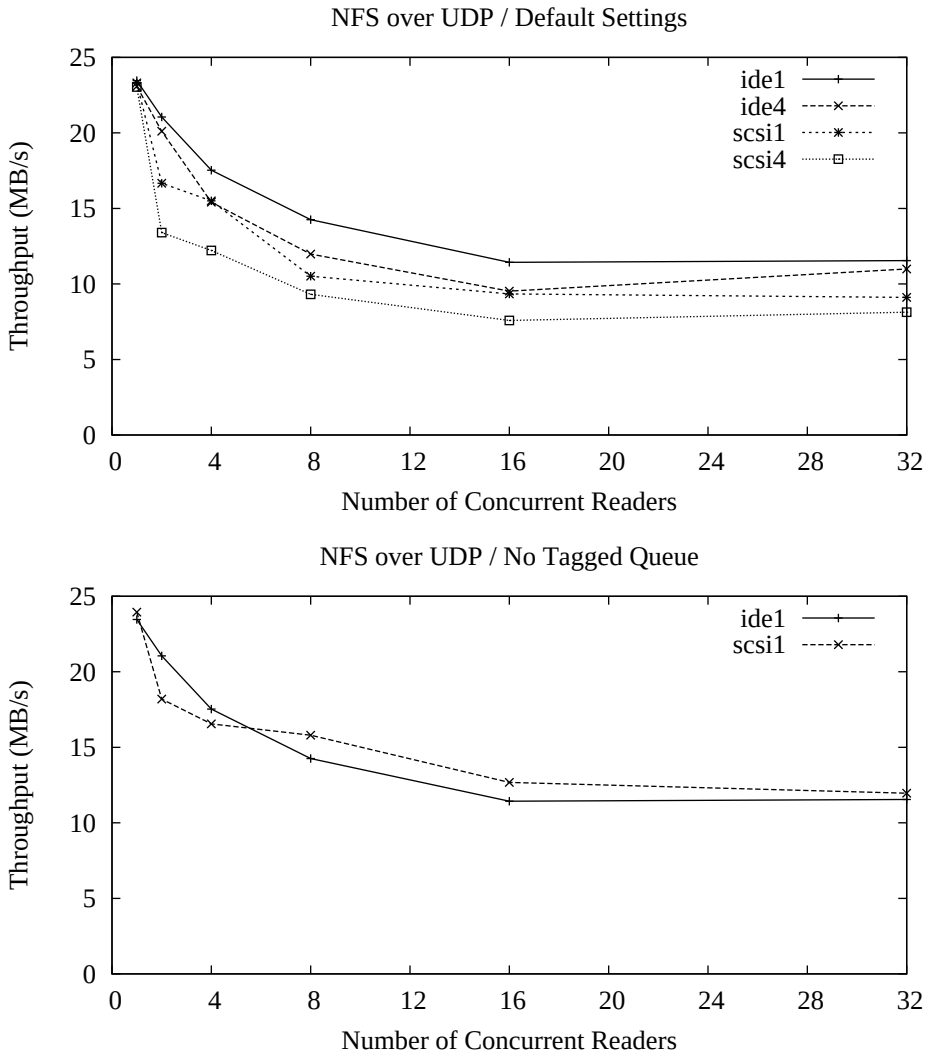


Figure 5.4: The speed of NFS over UDP, with and without tagged queues.

Performance drops quickly as the number of concurrent readers increases. With tagged queues disabled, `scsi1` performance improves relative to `ide1` as the number of concurrent readers increases. Note that the ZCAV effect is still visible, even over the network.

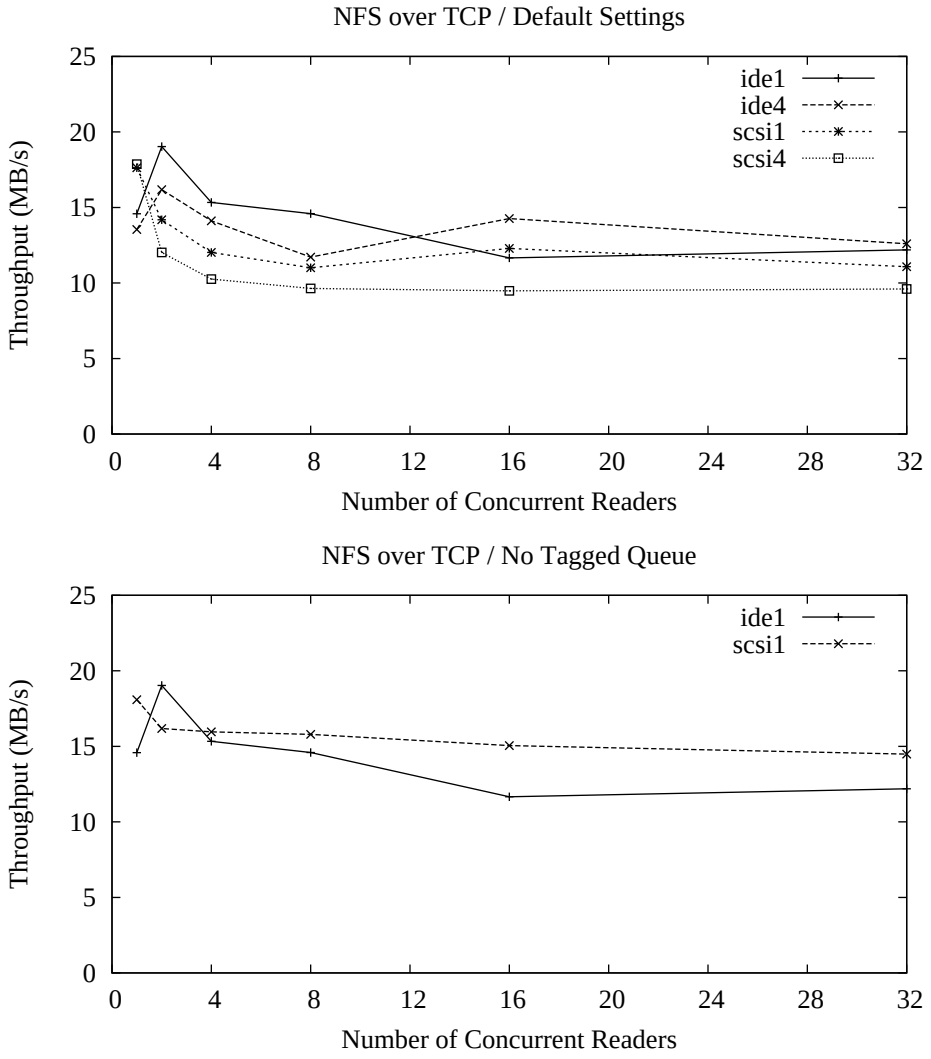


Figure 5.5: The speed of NFS over TCP, with and without tagged queues.

Compared to UDP, the throughput is relatively constant as the number of concurrent readers increases, especially when tagged queues are disabled on the `scsi1`. I do not know why the IDE partitions show a performance spike at two concurrent readers, nor why `ide4` is faster than `ide1` for 16 readers.

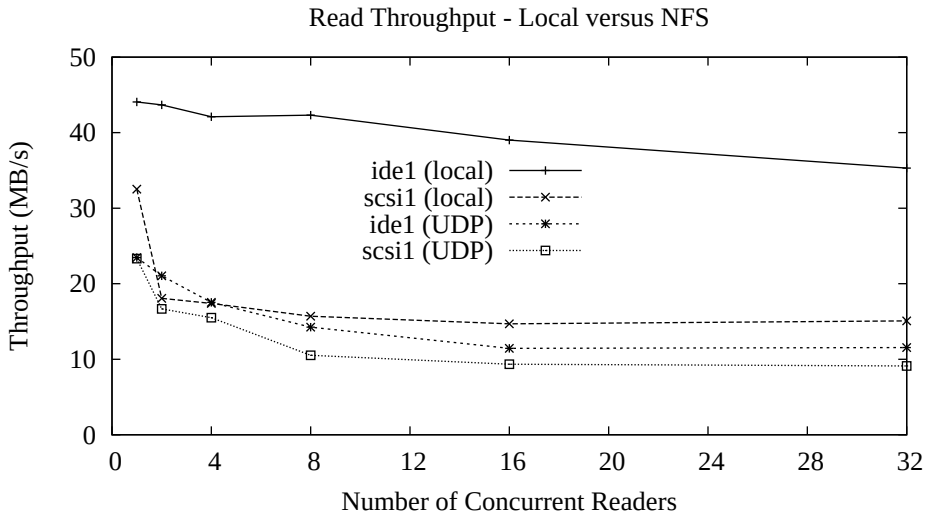


Figure 5.6: Performance of the local file system versus NFS over UDP on a gigabit LAN.

RPC over TCP has advantages in some environments, particularly WAN systems, due to the different characteristics of TCP versus UDP. UDP is a lightweight and connectionless datagram protocol. A UDP datagram may require several lower-level packets (such as Ethernet frames) to transmit, and the loss of any one of these packets will cause the entire datagram to be lost. In contrast, TCP provides a reliable connection-based mechanism for communication and can, in many cases, detect and deal with packet corruption, loss, or reordering more efficiently than UDP. TCP provides mechanisms for intelligent flow control that are appropriate for WANs.

On a wide-area network, or a local network with frequent packet loss or collision, TCP connections can provide better performance than UDP. Modern LANs are nearly always fully switched, and have low packet loss rates, so the worst-case behavior of UDP is rarely observed. However, mixed-speed LANs do experience frequent packet loss at the junctions between fast and slow segments, and in this case the benefits of TCP are also worth considering. In my testbed, I have only a single switch, so I do not observe these effects in my benchmarks.

The RPC transport protocol used by each file system mounted via NFS is chosen when the file system is mounted. The default transport protocol used by `mount_nfs` is UDP. Many system administrators use `amd` instead of `mount_nfs`, however, and `amd` uses a different implementation of the mount protocol. On FreeBSD, NetBSD, and many distributions of GNU/Linux, `amd` uses TCP by default, but on other systems, such as OpenBSD, `amd` uses UDP. This choice can be overridden, but often goes unnoticed.

A comparison of the raw throughput of NFS for large reads over TCP and UDP is given in Figures 5.4 and 5.5. Compared to the performance of the local file system, shown in Figure 5.1, the throughput of NFS is disappointing; for concurrent readers the performance is about half that of the local file system and only a fraction of the potential bandwidth of the gigabit Ethernet. The throughput for small numbers of readers is substantially better for UDP than TCP, but the advantage of UDP is attenuated as the number of concurrent readers increases until it has no advantage over TCP (and in some cases is actually slower). In contrast, the throughput of accesses to the local disk slightly increases as the number of readers increases. I postulate that this is due to a combination of the queuing model used by the disk drive and the tendency of the OS to perform read-ahead when it perceives that the access pattern is sequential, but also to throttle the read-ahead to a fixed limit. When reading a single file, a fixed amount of buffer space is set aside for read-ahead, and only a fixed number of disk requests are made at a time. As the number of open files increases, the total amount of memory set aside for read-ahead increases proportionally (until another fixed limit is reached) and the number of disk accesses queued up for the disk to process also grows. This allows the disk to be kept busier, and

thus the total throughput can increase.

Unlike UDP, the throughput of NFS over TCP roughly parallels the throughput of the local file system, although it is always significantly slower, even over gigabit Ethernet. This leads to the question of why UDP and TCP implementations of NFS have such different performance characteristics as the number of readers increases – and whether it is possible to improve the performance of UDP for multiple readers.

Summary of Benchmark Traps

- *Do not overlook ZCAV effects.*

  The best method to control ZCAV effects is to use the largest disk available and run your benchmark in the smallest possible partition (preferably the outermost partition). This will minimize the ZCAV effect by minimizing the difference in capacity and transfer rate between the longest and shortest tracks used in your benchmark.

- *Know your hardware.*

  Typical desktop workstations or low-end blade servers use PCI implementations that have a peak transfer speed slower than one gigabit per second (and quite a bit slower than typical disk drives). They cannot move data on or off a gigabit Ethernet controller fast enough to sustain gigabit throughput and therefore may be a hidden bottleneck.

- *Check for unexpected variation.*

  The disk scheduler can order I/O in a different order than you expect – and tagged command queues can reorder them yet again. This can cause unexpected effects; watch for them.

- *Know your protocols.*

  Are you using TCP or UDP? Does it make a difference for your test? Would it make a difference in other situations? There may be too many possible configurations to test exhaustively, but it is best to try, at the very least, the most likely possibilities.

5.4 The SlowDown Heuristic

As I mentioned in Section 5.2, the heuristics employed by NFS and FFS begin to break down when used on UDP-based NFS workloads because many NFS client implementations permit requests to be reordered between the time that they are made by client applications and the time they are delivered to the server. This reordering is due most frequently to queuing issues in the client `nfsiod` daemon, which marshals and controls the communication between the client and the server. This reordering can also occur due to network effects, but in my system the re-orderings are attributable to `nfsiod`.

It must be noted that because this problem is due entirely to the implementation of the NFS client, a direct and pragmatic approach would be to fix the client to prevent request reordering. This is contrary to my research agenda, however, which focuses on servers. I am more interested in studying how servers can handle arbitrary and suboptimal client request streams than optimizing clients to generate request streams that are easier for servers to handle.

The fact that NFS requests are reordered means that access patterns that are, in fact, entirely sequential from the perspective of the client may appear, to the server, to contain some element of randomness. When this happens, the default heuristic causes read-ahead to be disabled (or diminished significantly), causing considerable performance degradation for sequential reads.

The frequency at which request reordering takes place increases as the number of concurrent readers, the number of `nfsiod`s, and the total CPU utilization on the client increases. By using a slow client and a fast server on a congested network, I have been able to create systems that reorder more than 10% of their requests for long periods of time, and during my analysis of traces from production systems I have seen similar percentages during periods of

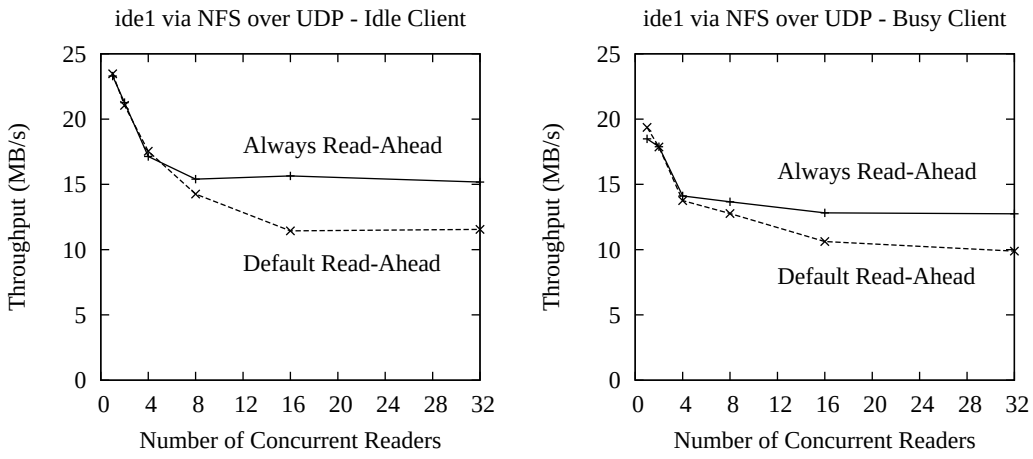


Figure 5.7: A comparison of the NFS throughput for the default read-ahead heuristic and a hard-wired "always do read-ahead" heuristic.

All tests are run using file system `ide1` via NFS over UDP. The idle client is running only the benchmark, while the busy client is also running four infinite loop processes.

---

peak traffic. On my benchmark system, however, I was unable to exceed 6% request reordering on UDP and 2% on TCP on my gigabit network with anything less than drastic measures. This is slightly disappointing to me because lower probabilities of request reordering translate into less potential for improvement via the SlowDown algorithm (described later, in Section 5.4.2), but it is good news for other NFS users because it limits the performance degradation that request re-ordering can have. Nevertheless, for the rest of this section I will explore ideas for overcoming request reordering and measure its effects on my system. If these ideas are helpful on my network, they might prove even more useful to users on less well-mannered networks.

5.4.1 Estimating the Potential for Improvement

Figure 5.7 shows the NFS throughput for the default implementation compared to the throughput when I hardwire the sequentiality metric to always force read-ahead to occur. The difference between the "Always Read-ahead" and "Default Read-ahead" lines shows the potential improvement. In theory, for large sequential reads (such as my benchmarks) the NFS server should detect the sequential access pattern and perform read-ahead. As shown in Figure 5.7, however, for more than four concurrent readers the default and optimal lines diverge. This is due in part to the increased number of packet re-orderings that occur when the number of concurrent readers increases, but it is also due to contention for system resources, as I will discuss in Section 5.4.3.

In my own experiments, I noticed that the frequency of packet reordering increases in tandem with the number of active processes on the client (whether those processes are doing any I/O or not), so Figure 5.7 also shows throughput when the client is running four "infinite loop" processes during the benchmark. Not surprisingly, the throughput of NFS decreases when there is contention for the client CPU, because NFS does have a significant processing overhead. It may seem somewhat counterintuitive, however, that the gap between the "Always Read-ahead" line and the "Default Read-ahead" lines is actually smaller when the CPU is loaded, even though I see more packet reordering.

5.4.2 The *SlowDown* Sequentiality Heuristic

There are many ways that the underlying sequentiality of an access pattern may be measured, such as the metrics developed in my earlier studies of NFS traces [24]. For my preliminary implementation, however, I wish to find a simple heuristic that does well in the expected case, not very badly in the worst case, and requires a minimum of bookkeeping and computational overhead.

My current heuristic is named *SlowDown* and is based on the idea of allowing the sequentiality score to rise in the same manner as the ordinary heuristic, but not cause "slightly" out-of-order requests to immediately set the sequentiality score back to a minimal value and temporarily disable read-ahead, as described in Section 5.2. Unlike the default behavior (where a single out-of-order request can drop the sequentiality score back to one), the *SlowDown* heuristic is resilient to "slightly" out-of-order requests. At the same time, however, it does not waste read-ahead on access patterns that do not have a strongly sequential component. If the access pattern is truly random, it will quickly disable read-ahead. The default metric for computing the heuristic, as implemented in FreeBSD 4.x, is essentially the following: when a new file is accessed, it is given an initial sequentiality metric $seqCount = 1$ (or sometimes a different constant, depending on the context). Whenever the file is accessed, if the current offset *currOffset* is the same as the offset after the last operation (*prevOffset*), then increment *seqCount*. Otherwise, reset *seqCount* to a low value.

The *seqCount* is used by the file system to decide how much read-ahead to perform – the higher *seqCount* rises, the more aggressive the file system becomes. Note that in both algorithms, *seqCount* is never allowed to grow higher than 127, due to the implementation of the lower levels of the operating system.

The *SlowDown* heuristic is nearly identical in concept to the additive-increase and multiplicative-decrease used by TCP/IP to implement congestion control, although its application is very different. The initialization is the same as for the default algorithm, and when *prevOffset* matches *currOffset*, *seqCount* is incremented as before. When *prevOffset* differs from *currOffset*, however, the response of *SlowDown* is different:

- If *currOffset* is within 64k (eight 8k NFS blocks) of *prevOffset* then *seqCount* is unchanged.
- If *currOffset* is more than 64k from *prevOffset*, then divide *seqCount* by 2.

In the first case, I do not know whether the access pattern is becoming random, or whether I am simply seeing jitter in the request order, so I leave *seqCount* alone. In the second case, I want to start to cut back on the amount of read-ahead, and so I reduce *seqCount*, but not necessarily all the way to zero. If the non-sequential trend continues, however, repeatedly dividing *seqCount* in half (using integer division) will quickly chop it down to zero.

It is possible to invent access patterns that cause *SlowDown* to erroneously trigger read-ahead of blocks that will never be accessed. To counter this, more intelligence (requiring more state, and more computation) could be added to the algorithm. However, in my trace analysis I did not encounter any access patterns that would trick *SlowDown* to perform excessive read-ahead. The only goal of *SlowDown* is to help cope with small reorderings in the request stream. An analysis of the values of *seqCount* show that *SlowDown* accomplishes this goal.

The SlowDown heuristic is similar to a read-ahead heuristic used by Margo Seltzer to improve performance on a benchmark that had a quasi-sequential access pattern that triggered too much read-ahead and thereby thrashed the cache [73] (Section 7.7). Both heuristics halve the sequentiality score when a non-sequential request is detected. It is ironic that Seltzer's intent was to disable read-ahead because the default heuristic was too greedy in the face of non-sequential accesses, while my intent is to keep read-ahead alive in spite of re-ordered requests. The default heuristic used by her system was arguably too optimistic, while the default heuristic in FreeBSD 4.x is arguably too pessimistic, and therefore we have met in the middle.

Seltzer's experience also shows that unbounded read-ahead can hurt overall performance. Even though the I/O cost of reading additional blocks from the disk is relatively low, the opportunity cost of storing those blocks in memory may be high. If memory is a scarce resource then blindly performing as much read-ahead as possible may cause total system throughput to decrease as truly important objects are evicted from the cache to make room to hold objects whose importance is purely speculative. The "Always Read-ahead" strategy can backfire in this situation (although this is not the case in the benchmarks I use in this thesis). Therefore it is important to choose a heuristic that finds a

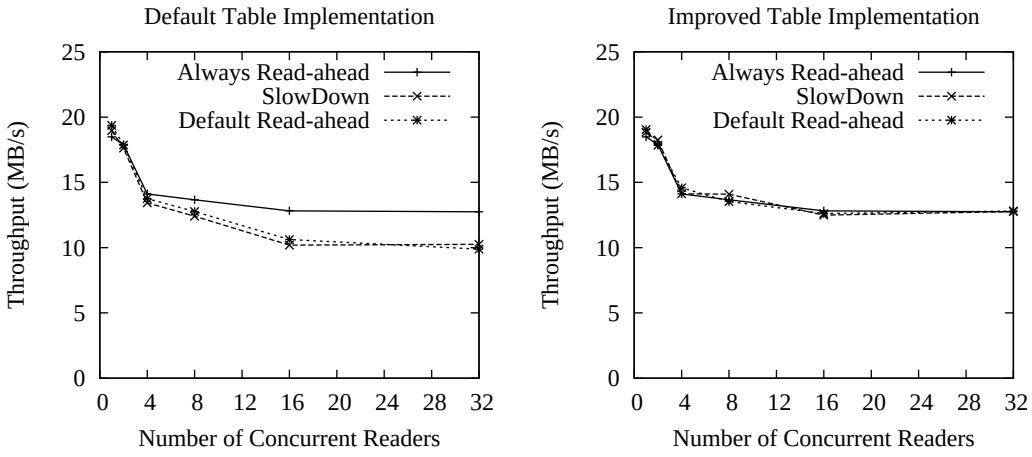


Figure 5.8: A comparison of the NFS throughput for the default read-ahead heuristic, a hard-wired "always do read-ahead" heuristic, and the SlowDown heuristic.

All tests are run using file system `ide1` and NFS over UDP. In all tests, the client is running four infinite loop processes. The plot on the left shows the performance of the system using the default implementation of the *nfsheur* table that holds the per-file sequentiality scores. The plot on the right shows the impact of improving the implementation of the *nfsheur* table. After the table is has been enlarged, it is less likely that *nfsheur* entries are evicted from the table to make room for new entries, and the performance of both SlowDown and the default heuristic are identical to the performance of the hard-wired best case that always performs read-ahead.

---

good balance between maximizing cache utilization (by never performing any read-ahead at all) and minimizing the expected cost of the disk operations (by always performing as much read-ahead as possible).

5.4.3 Benchmarking SlowDown

My offline analysis of SlowDown showed that it was capable of eliminating the effect of re-ordered packets on the sequentiality score of a file. When I benchmarked it against the default heuristic and the "Always Read-ahead" the result was disappointing. Instead of running as fast as "Always Read-ahead", the SlowDown heuristic appeared to offer no advantage over the default. This result is illustrated by the plot on the left of Figure 5.8.

This result was surprising enough to prompt a closer look at the underlying NFS implementation, where I discovered a design flaw in the table used to hold information about each active file. This table, which is named *nfsheur*, was implemented as a hash table, using an open hashing scheme. Unlike ordinary hash tables, the number of probes attempted before giving up a search was limited to a small fraction of the hash table size. Also unlike ordinary hash tables, which fail when an insertion is attempted into a full table, this hash table chooses the least-recently accessed element in the table and ejects it in order to make space for a new element. When this happens, all information about the access pattern of the ejected file is lost, and if that file is accessed again it receives the default sequentiality score.

The net effect of these somewhat arbitrary implementation decisions was that a relatively small number of concurrent file accesses can thrash the table, limiting the ability of the system to keep track of whether or not each file was accessed sequentially. Fixing the implementation of *nfsheur* (by improving the hash function to reduce collisions, increasing the permitted number of probes, and increasing the overall table size) prevents this thrashing, and allows

SlowDown to work as expected.[7] For sequential accesses, SlowDown performs nearly as Always Read-ahead, even when requests are re-ordered. The result of this improvement is illustrated in the plot on the right side of Figure 5.8.

Unfortunately, the plot on the right side of Figure 5.8 also shows that the default heuristic is indistinguishable from Always Read-ahead once the table implementation has been improved! This is explained by the elasticity built in to the read-ahead heuristic, as mentioned in Section 5.2. As long as the sequentiality score is permitted to grow reasonably high between re-ordered requests, it does not matter that it is pruned back to a low value when a re-ordered request arrives.

For this benchmark, it is more important to have have a non-zero sequentiality score in the table than it is to have the right sequentiality score. It is an open question how precise the sequentiality score must be in order to achieve optimal performance; it depends on both the workload and the characteristics of the components of the file system that actually use this score to schedule read-ahead, and the device on which the read-ahead is performed.

5.5 The Cursor Heuristic

Conventional read-ahead heuristics only attempt to detect sequentiality. They cannot take advantage of access patterns that are the composition of more than one sequential access pattern.

The conventional implementation of the sequentiality metric in the FreeBSD implementation of NFS (and in fact, in many implementations of FFS) uses a single descriptor structure to encapsulate all information about the observed read access patterns of a file. This can cause suboptimal read-ahead when there are several readers of the same file or when a single reader reads a file in a regular but non-sequential pattern. For example, imagine a process that traverses a file, reading blocks $0,\ x,\ 1,\ x+1,\ 2,\ x+2,\ 3,\ x+3\ \cdots$. This pattern is the composition of two completely sequential read access patterns ($0,\ 1,\ 2,\ 3, \cdots$ and $x,\ x+1,\ x+2,\ x+3, \cdots$), each of which can benefit from read-ahead. Unfortunately, neither the default sequentiality metric nor *SlowDown* recognizes this pattern. This access pattern will be treated as non-sequential and no read-ahead will be performed. Variations on the pattern are common in engineering and out-of-core workloads, and optimizing them has been the subject of considerable research, although it is usually attacked at the application level or as a virtual memory issue [12, 52].

In the ordinary implementation, the *nfsheur* table contains a single offset and sequentiality count for each file handle. In order to handle such read patterns, I add the concept of *cursors* to the *nfsheur* table. Each active file handle may have several cursors, and each cursor contains its own offset and sequentiality count. When a read occurs, the sequentiality metric searches *nfsheur* for a matching cursor (using the same approximate match as *SlowDown* to match offsets). If it finds a matching cursor, the cursor is updated and its sequentiality count is used to compute the effective *seqCount* for the rest of the operation, using the *SlowDown* heuristic. If there is no cursor matching a given read, then a new cursor is allocated and added to *nfsheur*.

I cannot continue adding cursors to the table indefinitely, and so there must be some policy about what to do when the table grows too large. One possible strategy is to have a pool of cursors shared among all of the files active at any given moment, but this strategy will backfire when a single large file is accessed in a highly random manner – if no two accesses match the same cursor, then the number of cursors used by that file will grow until it is proportional to the size of the file. Another possible strategy is to give each file a pre-allocated set of cursors it may use. This is the strategy I use at this point. The number of cursors per file is fixed when the kernel is compiled. When this limit is exceeded, the least recently used cursor for that file is recycled.

If the access pattern is truly random, then many cursors are created, but their sequentiality counts do not grow and no extra read-ahead is performed. In the worst case, a carefully crafted access pattern can trick the algorithm into maximizing the sequentiality count for a particular cursor just before that cursor dies (and therefore potentially performing read-ahead for many blocks that are never requested), but the cost of reading the extraneous blocks can be amortized over the increased efficiency of reading the blocks that *were* requested (and caused the sequentiality count

[7] My changes to *nfsheur* have been accepted into the 5.x release branch of the FreeBSD distribution.

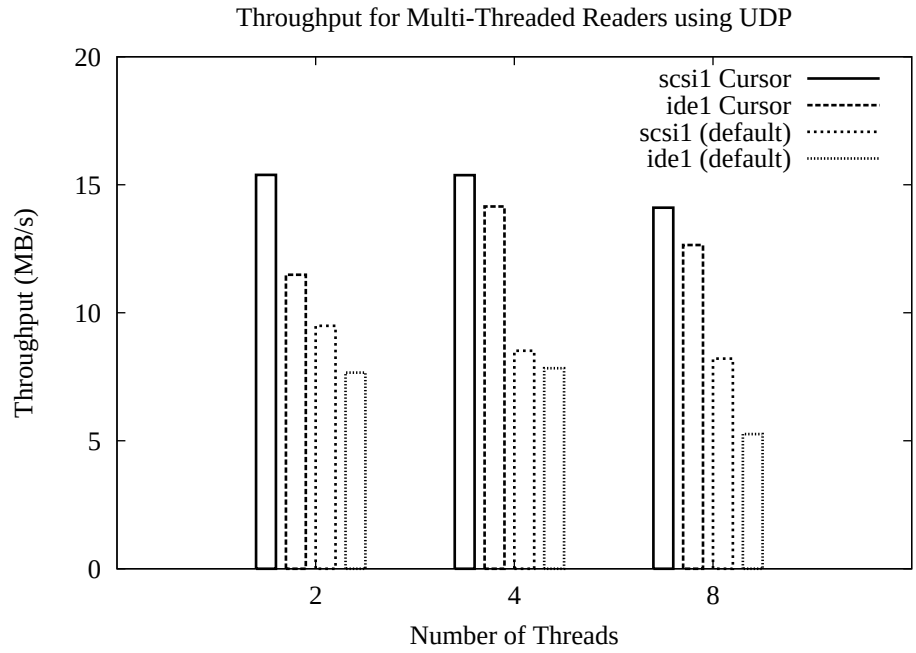


Figure 5.9: Throughput using the default NFS read-ahead compared to the cursor read-ahead, for reading a 256 MB file using 2, 4, and 8 concurrent reading threads.

`scsi1` runs 60-70% faster for all tests when cursors are enabled. `ide1` is only 50% faster for the 2-thread test with cursors enabled, but 140% faster for the 8-thread test.

---

to increase in the first place).

The performance of this method for small numbers of concurrent readers is shown in Figure 5.9 and Table 5.1. For the 2-thread case and a file of length n, there are two concurrent threads, the first reading sequentially offset 0 to offset $n/2$ and the second starting at offset $n/2$ and reading until the end of the file. For the 4-thread case there are four threads, each of which read a different quarter of the file, and so forth. To the ordinary sequentiality or *SlowDown* heuristics, these appear to be completely random access patterns, but my cursor-based algorithm detects the sequential sub-components and induces the proper amount of read-ahead for each cursor. As shown in Table 5.1, the cursor-based method is at least 50% faster than using the default method. In the most extreme case, the cursor-based method is 140% faster for the 8-thread reader on `ide1`.

It is notable that the throughput for `ide1` actually increases from the 2-reader to the 4-thread cases. Due to the manner in which I implemented cursors, each individual cursor has the potential to request as much read-ahead as a single file in the default system. In the 4-thread case, this means that the total amount of read-ahead requested and cached by the system is twice that of the 2-thread case, improving the total throughput. I do not see a similar improvement for `scsi1` because it is already achieving its full potential, the bandwidth illustrated in Figure 5.5.

5.6 Summary

In this chapter I have demonstrated that it is possible to use cursor-based heuristics to substantially increase the read performance of a network storage server for workloads that contain sequential subcomponents. As shown in Figure

File System		s = 2	s = 4	s = 8
`ide1`	UDP/Default	7.66 (0.02)	7.83 (0.02)	5.26 (0.02)
	UDP/Cursor	11.49 (0.29)	14.15 (0.14)	12.66 (0.43)
`scsi1`	UDP/Default	9.49 (0.03)	8.52 (0.04)	8.21 (0.03)
	UDP/Cursor	15.39 (0.20)	15.38 (0.15)	14.12 (0.46)

Table 5.1: Mean throughput (in MB/s) of ten reads of a single 256 MB file using a sequential read by several different threads, comparing the default read-ahead heuristic to the cursor-based heuristic.

The numbers in parenthesis give the standard deviation for each sample. The cache is flushed before each run, and no part of the file is read more than once and therefore caching has no effect on the throughput.

5.9 and Table 5.1, increasing the intelligence of the read-ahead heuristic in a reasonably simple manner can increase the end-to-end performance of certain common access patterns by 50%.

I have also illustrated many benchmarking traps that must be taken into account when measuring the effect of changes to the implementation of network storage servers. These traps can overwhelm or obscure the effect of these changes and lead to confusing or irreproducible results, and isolating the effect of each trap requires non-trivial testing and measurement. For example, in the process of testing my SlowDown heuristic I uncovered (and fixed) a coding error in the implementation of the FreeBSD NFS server that had the potential to significantly impact the throughput of busy a NFS server by as much as 20%. Due to the difficulty in isolating the many factors that influence benchmark results, this error degraded the performance of a widely deployed open-source NFS server for several years without anyone even suspecting it was there.

Chapter 6

Name-Based File Predictions

The most interesting discovery that arose from my analysis of the traces described in Chapter 3 is that the names of files are associated with properties such as file size and lifespan. In this chapter I show how these associations were discovered, demonstrate a method for using this association to construct simple feature-based predictive models capable of accurately classifying new files and discuss the properties of these models.

6.1 A First View of a Larger Pattern

While performing the analyses described in Chapter 3, I noticed patterns between the names given to files and the way those files were used in the CAMPUS traces. On CAMPUS, there is a strong and obvious relationship between patterns within the names of files and the properties of the files [25]. I discovered that almost all of the files that appear in the CAMPUS trace can be accurately categorized simply by examining their names:

- Lock files whose names contain the string "`.lock.`" and end with the name of a host. These files usually live for only a few seconds and do not contain data.
- Mailboxes named `.inbox`, which are immortal (although they are frequently overwritten, and therefore the data within them dies frequently), large, and accessed sequentially.
- Temporary mail composer files whose names begin with `#pico`, which live for a few minutes, have data written to them, but are rarely read. Most of these files are less than 16K in length, although some are considerably larger.
- Dot files (configuration files) such as the users `.login`, `.cshrc`, and `.pinerc`. These files are small, read-only, and immortal.
- Log files such as `.history`, which are short and written when the user logs out.

One trait that all of these files share is that none of their names are chosen by a user – these patterns reflect a set of global policies and many of these names are chosen by hard-coded mechanisms. Therefore it is not hard to believe that the names of these files are associated with the way in which they are used.

CAMPUS may be unusual in the homogeneity of its workload, but other applications choose file names for their users (or constrain the choice of names users can select), and even when given complete freedom to choose file names, most people either invent a consistent naming scheme (or adopt a scheme used by their coworkers) in order to organize their files and thereby make their own lives easier. This raises the question of whether file names can be used as hints about how the files will be used for workloads other than CAMPUS, and if so, how the meaning of these hints can be discovered and used.

6.2 Name-Based Associations

To explore these patterns, I begin with an examination of the distribution of file sizes and lifespans for files observed in the CAMPUS, DEAS, and EECS traces. The results appear in Tables 6.1, 6.2, and 6.3. In these tables, the Name is either a prefix or suffix of a file name (or the entire file name, when the name consists of a string without a prefix or suffix). The special string HOST is substituted when the file prefix or suffix is the name of the originating host. The Count is the number of times a file name with a matching prefix or suffix was accessed during a 24-hour period. The Cost is a simple estimate of the amount of I/O activity actually generated by these accesses, which is computed as the number of accesses plus 0.125 multiplied by the number 8K blocks transferred. This metric depends on both the number of accesses and the amount of data transferred, because both are important; the number of accesses is often to the number of disk operations, while the total size of the transfers is proportional to the disk bandwidth consumed. The Size column shows the file size distribution, shown as a log-scale boxplot (ranging from 0 to 2^{32} bytes). The Lifespan column shows a linear-scale boxplot of the lifespan for the files created within this 24-hour period (ranging from 0 to 24 hours). If fewer than 10 files with the same Name were created during the sample period, a (NA) is shown instead of a boxplot.

The boxplot format used in these tables is described in Figure 6.1. The sizes are given in log-scale, while the lifespan is measured linearly (ranging from 0 seconds to one day). Not shown on these plots are the scales, but because I am looking for patterns rather than specific values, the actual numbers are not as important as their relative sizes. The same scale is used for each plot, however, so any two plots are directly comparable.

The first thing to note about these figures is that files with different features have different distributions. For many of the names, there are notable differences in the mean of the the size and lifespan.

The second thing to note is that many of the distributions have a strong central tendency – the distribution of sizes and lifespans for files with a given suffix is not uniform, but instead is usually clustered around a particular value. In some cases, this clustering is perfect (or nearly so) – for example, all of the `.logout` files on CAMPUS are exactly the same size, as are all of the `.pinerc` files. There is more variation among the sizes of `.login` files, but at least 90% of the `.login` files observed are the same size. Some of the plots also show that sometimes files with different names have similar characteristics – for example, on CAMPUS the distribution of sizes for files named `"sent-mail"` and `"Sent Items"` are remarkably similar.

Even for files that show more variation, there is, in many cases, a relationship between file name suffix and size or lifespan. Although some of the distributions have long tails, many of the boxplots show that the middle 50% of the values are grouped in a small range – particularly for size. These distributions can be used to make predictions about the relative sizes of new files with different suffixes – for example, on DEAS03 95% of the files that have a suffix of `.gif` are smaller than any `.stt` files.

Note that examining the different distributions graphically via boxplots, although fairly intuitive and informative, provides only weak statistical evidence. In collaboration with my colleagues Michael Mesnier and Eno Thereska, I have investigated these associations using more rigorous techniques [26, 46]. Since this is primarily the work of my colleagues, I will only summarize the results: they show, using several measures of association, that file names are strongly associated with many file properties, including lifespan, size, and access patterns. They also demonstrate that other attributes, including the permission mode, the owner, and the group owner of files are strongly associated with these properties as well. In virtually all cases, the confidence level of the association is greater than 99% (and usually even closer to 100%).

6.3 Name-Based Predictions

It is one thing for a human to be able to perceive patterns in trace data, but it is another thing entirely to be able to automate the process of discovering these patterns and building predictive models that can guide decisions about file system policies. In this section, I show that it is possible to construct accurate predictive name-based models for my three workloads.

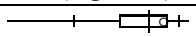
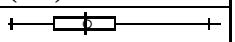
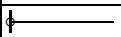
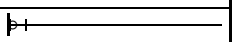
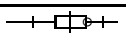
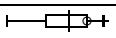
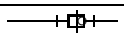
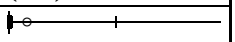
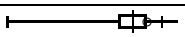
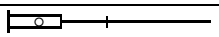
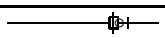
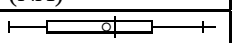
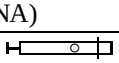
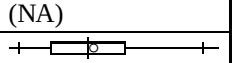
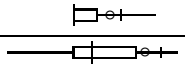
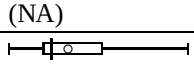
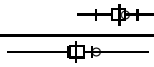
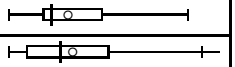
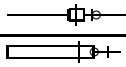
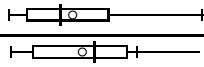
Name	**Count**	**Cost**	**Size** (log-scale)	**Lifespan** (linear)
*.inbox	1222	2011892		(NA)
*.lock	76027	76029		
*.HOST	65383	65383		
sent-mail	553	22651		(NA)
Sent Items	37	14046		(NA)
#pico*	2746	4987		
SENT-MAIL-*	592	2068		
INBOX	12	1731		(NA)
*.doc	1459	1693		
*.history	789	1572		(NA)
*.pinerc	704	1365		
*.login	767	1148		(NA)
*.cshrc	765	1141		(NA)
*.aliases	745	1117		(NA)
*.logout	633	907		(NA)
postponed-msgs	259	710		
saved-messages	160	608		(NA)
*.jpg	306	575		
*.letter	500	541		
*.addressbook	413	482		

Table 6.1: Per-file statistics for simple file name prefix/suffix strings on CAMPUS, sorted by file cost, for files accessed on 10/22/2001.

See Figure 6.1 for an explanation of each column.

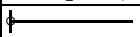
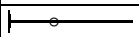
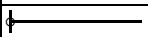
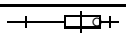
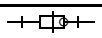
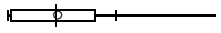
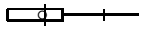
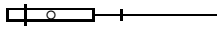
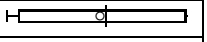
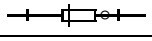
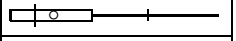
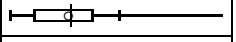
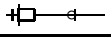
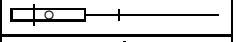
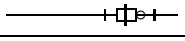
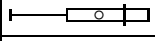
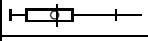
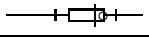
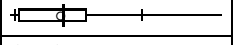
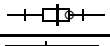
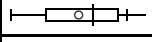
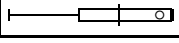
Name	**Count**	**Cost**	**Size** (log-scale)	**Lifespan** (linear)
*.lock	35315	35315		
*.HOST	31900	31924		
sent-mail	57	21481		(NA)
mbox	37	18396		(NA)
*.so	450	17144		
*.pac	11	15912		(NA)
*.odb	95	12907		
*.gif	9008	11771		
*.stt	150	9791		
*.abq	11	8755		(NA)
INBOX	12	5138		
*.dat	340	4794		
*.inp	97	4656		
*.res	106	4492		
*.pdf	472	3232		
*.jpg	1563	2876		
*.html	1492	2124		
*.db	97	2037		(NA)
*.ps	352	1761		
*.x	16	1755		(NA)

Table 6.2: Per-file statistics for simple file name prefix/suffix strings on DEAS, sorted by file cost, for files accessed on 10/21/2002.

See Figure 6.1 for an explanation of each column.

Note that the data for this table comes from the DEAS trace from October, 2002, not the DEAS03 trace from February, 2003.

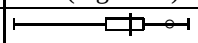
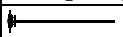
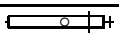
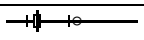
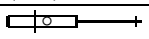
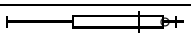
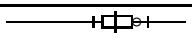
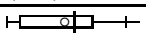
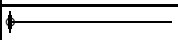
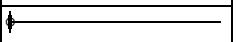
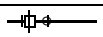
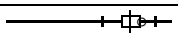
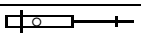
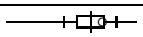
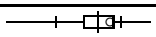
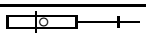
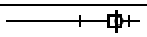
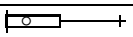
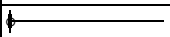
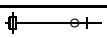
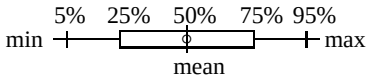
Name	**Count**	**Cost**	**Size** (log-scale)	**Lifespan** (linear)
*.3DIG	712	153219		
perfdb0	73	39235		
perfdb1	73	18477		
RMAIL	38	7804		
core	17	7288		(NA)
*.gz	4976	6126		(NA)
*.gif	5045	5459		
mbox	11	5333		(NA)
*.ps	1145	5224		
Applet_B	5009	5129		
Applet_C	4087	4130		
3DIG	2820	4118		
*.pdf	343	4092		
*.html	2104	3002		
*.jpg	1909	2824		
*.o	1537	2625		
Applet_A	2489	2491		
*.HOST	2230	2326		
*.lock	2223	2304		(NA)
10	21	1769		

Table 6.3: Per-file statistics for simple file name prefix/suffix strings on EECS, sorted by file cost, for files accessed on 10/22/2001.

See Figure 6.1 for an explanation of each column.

Note that the data for this table comes from the EECS trace from October, 2001, not the EECS03 trace from February, 2003.



The horizontal lines extend to the minimum and maximum values in the sample. The vertical lines of the box plot represent, from left to right, 5%, 25% (lower quartile), 50% (median), 75% (upper quartile), and 95% boundaries. The mean value is denoted with a small circle.

In Tables 6.1, 6.2 and 6.3), the Name is either a prefix or suffix of a file name (or the entire file name, when the name consists of a string without a prefix or suffix). The special string HOST is substituted when the file prefix or suffix is the name of the originating host. The Count is the number of times a file name with a matching prefix or suffix was accessed during a 24-hour period. The Cost is a simple estimate of the amount of I/O activity generated by these accesses, which is computed as the number of accesses plus 0.125 multiplied by the number 8K blocks transferred. The Size column shows the file size distribution, shown as a log-scale boxplot (ranging from 0 to 2^{32} bytes). The Lifespan column shows a linear-scale boxplot of the lifespan for the files created within this 24-hour period (ranging from 0 to 24 hours). If fewer than 10 files with the same Name were created during the sample period, a (NA) is shown instead of a boxplot.

Figure 6.1: The notation used in Tables 6.1, 6.2, and 6.3.

One of the constraints of the algorithms I use to construct and use my predictive models is that they must be computationally economical and require low space overhead (both for computing and storing the model). If the model is overly cumbersome to compute or represent, the cost of using it may overshadow any benefit it provides. My goal, therefore, is to develop an efficient algorithm for building reasonably accurate models, rather than striving to achieve accuracy at any cost.

6.3.1 Selecting Features to Use for Predictions

The feature used by my algorithm for classifying file names is the set of *components* of the file names. The components are used to construct simple regular expressions that are tested against future file names. My initial procedure for discovering the components of a file name is quite simple, and is given in Figure 6.2.

This extremely simple partitioning works surprisingly well because many applications construct file names by pasting together components with periods. Unfortunately, some applications do not use periods, but instead build file names by concatenating a random (or opaque) string to a fixed template. For example, older versions of the Netscape browser name all of their cached files `cache`*`X`*`.`*`suf`* (where *X* is an opaque identifier, and *suf* is the suffix of the original file), and the `pine` composer `pico` creates temporary files that begin with the string `#pico`. For the `cache`*`X`* files, I can still learn useful information from the suffix, but I do not benefit from learning that they are part of a browser cache, and hence tend to have different lifetime characteristics than "ordinary" files with the same suffix. The `#pico` files have distinct and interesting properties of their own, but models constructed from these components entirely overlook them. The optional step of adding the first five characters as an additional component addresses this situation.

For example, the file name "prediction.tex" has four components:

- "`^prediction\.tex$`"
- "`^prediction`"

- Let C be the set of components, initially empty. Let S be the file name. Let $E(s)$ be a function to "escape" a string s, protecting any characters in the string that would be interpreted as operators if the string is interpreted as a regular expression.
- Add $E(S)$ with a ^ prepended and $ appended to C.
- Split S at each period, creating set $s_0 \cdots s_n$.
- Let $e_i = E(s_i)$.
- Prepend a ^ to e_0 and append $ to e_n.
- Add each e_i to C.
- (Optional) If S contains more than 5 characters, let p be the first 5 characters of S. Add $E(p)$, with a ^ prepended, to C.

Figure 6.2: The algorithm for finding the set of components in a file name.

- "`tex$`"
- "`^predi`"

6.3.2 Building and Using a Model

The algorithm I use to construct a name-based model is given in Figure 6.3. In a nutshell, a model is a set of file name components that usually appear in the names of files that have a given property but appear infrequently (if at all) in the names of files that do not.

Given this set, the decision rule itself is simple: if the name of a new file contains a component that appears anywhere in the set, then predict that the new file will have the property. Otherwise, predict that the new file will not.

There are several parameters that can be tuned in order to improve the performance of the model. The most important of these is **minfrac**, which controls the probability with which a component is required to appear in the name of a file that does have the property in order for it to be a candidate for the final model. If **minfrac** is set too high, then only components that are perfect predictors survive the culling process – and unfortunately perfect predictors are rare. I have found that a **minfrac** of 0.8 works well for a wide range of properties, although it is not optimal for all. This issue is described in more detail in Section 6.3.5.

Selection of Predicates

Note that properties in this algorithm are described as *predicates*. This is because this algorithm is only able to construct models that predict binary properties – a file either has the property or it does not.

Also note that by design the predicates are chosen ahead of time. I envision that file system designers will choose predicates based on properties of the file system to help them to identify files with properties that they can exploit. A more intelligent system might be able to detect clusters of files in the workload, but my strategy here is not to discover natural clusters that appear in the data, but instead to classify files in terms of the artificial clusters defined by the implementation of the file system.

- Let P be a predicate.
- Let M be a set of tuples $\langle c, n \rangle$, where c is a component and n is a count. M is initially empty. M is used to store the number of times a file with a name containing component c has satisfied predicate P.
- Let A be a set of tuples $\langle c, n \rangle$, where c is a string and n is a count. A is initially empty. A is used to store the number of times each component has been observed.

1. For each file in the training set:
   (a) Let $C = \{c_0 \cdots c_m\}$ be the set of components of the file name (as defined in Section 6.3).
   (b) For each $c_i \in C$: if there exists a tuple $\langle c_i, n \rangle \in A$, then replace it with $\langle c_i, n+1 \rangle$. Otherwise, add $\langle c_i, 1 \rangle$ to A.
   (c) If the file satisfies predicate P, then for each $c_i \in C$: if there exists a tuple $\langle c_i, n \rangle \in M$, then replace it with $\langle c_i, n+1 \rangle$. Otherwise, add $\langle c_i, 1 \rangle$ to M.
2. For each $t_M = \langle c_M, n_M \rangle \in M$, find the corresponding $t_A = \langle c_A, n_A \rangle \in A$.
   If $n_M / n_A < \mathbf{minfrac}$, then remove t_M from M.
   The default **minfrac** is 0.8.
3. (Optional) For each tuple $t = \langle c_t, n_t \rangle \in M$, if $n_t < \mathbf{mincount}$ then discard t.
   The default **mincount** is 5.

Figure 6.3: An algorithm for building a predictive name-based model.

The final set M defines the predictive model. For each new file name, if there exists any $\langle c, n \rangle \in M$ such that the c is a component of the file name, then I predict that P is true for the corresponding file.

As an example, imagine a workload in which all of the files that have a suffix of `alpha` are exactly 900 bytes in length and files with a suffix of `beta` are exactly 1000 bytes in length. A truly intelligent system could discover this fact and use it to generate a model that can perfectly predict the length of new files that have a suffix of `alpha` or `beta`. However, unless there is any value in knowing the *exact* length of the file, such predictions are not useful. On modern file systems, knowing the exact length of a file is less important than knowing how many file system blocks it will require. Therefore the file system designer would be likely to care whether files on this system will require zero, one, or more blocks. Since files of length 900 and length 1000 are represented by the same number of blocks[1] there is no value in knowing whether the exact length will be 900 or 1000 bytes.

In the long run, it may prove useful to have models capable of more precise predictions (and such models may be transformed into classifiers for a fixed set of classes). I leave this area for future work.

6.3.3 Choosing the Training Set

The algorithm given in Figure 6.3 does not explicitly describe how to select which files should be used for training. This has been the subject of much heated discussion among my collaborators. My opinion is that only files that are created during the training period should be included in the training because I am primarily interested in making predictions about files when they are created. The probability that a file will have a given property, for many of the properties of interest, is dependent on whether the file is young or old. Many files on the system are "immortal" – they were created in the distant past, and might never be deleted. These files are perfect for treatment with standard adaptive techniques; if they live for a long time then there is plenty of time to adapt their implementation. My goal is to jump-start the adaptation process by trying to anticipate the properties that the file will have and preemptively adapt to that.

As an example, imagine that there is an application that creates ten files every day, all ending with the same suffix. It always deletes nine of these files within 30 seconds, but never deletes the tenth. At the end of nine days, 90 files have been created and 81 of them have deleted. If I then train on the tenth day, I will observe that ten files with this suffix are created, and that nine of them are deleted. I will also observe that there are nine additional files with the same suffix that are long-lived. Since the number of files with the suffix that are short-lived and long-lived is almost equal, the model is unable to make predictions about the lifespan of new files with the suffix. On the other hand, if I consider only the files that are created on the training day, then my model observes that 90% of the files with the suffix live for a short time and therefore predict that they all do. In my opinion, this is the correct choice.[2] The models I use in this thesis are all built using only files that were created during the training period.

6.3.4 Properties of the Name-Based Predictor

Computational Complexity: Building a Model

Model building is divided into two steps. In the first step, information about the name and simple properties of each file is gleaned by scanning the trace. The second step is building the set of components from the model.

The result of the first step is a table with a row for every file and a column for the name of each file and each property. Every record in the trace is processed exactly once and the amount of work required to compute the corresponding change (if any) to the table is bounded by a constant. Ideally the table would be implemented as a hash table (or similar structure) of sufficient size to provide $O(1)$ expected access and update times, but it is safer to assume

[1] At least, on typical contemporary block-structured storage devices, which have block sizes that are multiple of 256 bytes.

[2] The obvious rebuttal is that being right about 90% of the files is not worth much if the 10% for which I am wrong are long-lived and, over the course of their lifetime, have a much larger cost to the system than the short-lived files. Half of this issue is addressed in the section on error types (Chapter 6.3.4). The other half depends on the existence of an adaptive mechanism by which long-lived files can be re-organized when it becomes clear that this would be beneficial.

a tree-structured table implementation (such as a B-tree) that provides $O(\log n)$ access and update times (where n is the number of rows in the table). Therefore the total cost of preprocessing a trace containing n records is $O(n \log n)$.

Despite the low cost in terms of computational complexity, the implementation is non-trivial. The primary source of difficulty in the implementation arises because the traces are incomplete and therefore some information must be inferred (or guessed). This is exactly the same problem that was introduced in the context of building sub-snapshots (in Section 3.6.4). The problem is ameliorated by the fact that whenever a file is created its name must be specified as part of the creation step, and therefore it is never necessary to guess the name of a file.

The complicated logic needed to reconstruct the information about each file of interest, combined with the large number of records that must be scanned, results in a process that is slow. For systems like CAMPUS or DEAS03 it can take more than two hours of processing time on a moderately fast computer to build the table representing a single day of activity.[3]

The largest inefficiency of my current implementation is that it gathers all of its information about each file directly from my NFS traces, which in an uncompressed form can exceed 15G per server per day. Simply scanning through this volume of data to extract the appropriate 40MB of relevant information is a time-consuming process, but one that is not inherent in my algorithm. I believe that gathering this information at the file system interface (or from reduced NFS traces that include only the information I need, and not a full transcript of every NFS call and response) would greatly reduce this overhead, both in terms of storage space needed and computational overhead.

The good news is that most of this complexity and data are due to the decoupling of the trace-gathering procedure and the file system. If the table-building process was built into the file system (as I imagine it would be, when systems that use this kind of prediction strategy are deployed) then this information can be extracted directly and unambiguously from the file system instead of reconstructed piece-meal. My belief is that the process of maintaining the table can be implemented in a way that requires only negligible overhead.

Another drawback of an decoupled implementation is that it requires a noticeable amount of space to store the information observed during the training period; for a busy day on the busiest server, the log can consume nearly 40 megabytes.

Once the table is complete, however, the rest of the process goes quickly. Building the model from the table is fast, and depends only on the size of the table and the number of components in the file names. The algorithm requires less than 25 seconds on a PIII-800MHz to do the processing for all the files created during a full day of trace data from the DEAS system (approximately 60,000 files created per day). Building a model for all of the files observed during a day of training requires approximately 40 seconds on a PIII-800MHz. In both cases, most of the time is spent parsing the log files (which are stored as text and could be represented in a more efficient manner); after parsing the files, the cost to build a model for each property is approximately five seconds per property, although this depends somewhat on the time required to test whether each file has or does not satisfy the property.

Computational Complexity: Using a Model

The work necessary to check whether a new file has a name that matches a model depends on the number of components in the new file name and the number of components in the model. In the worst case, the total length of the components from a file name is always less than four times the length of the file name itself (and the expected length is closer to two or three, depending on whether or not the five-character heuristic is used): the length of the entire filename, plus the length of all the components separated by periods, plus any interesting punctuation, plus the five-character prefix. With use of a perfect hashing construction (such as a TRIE) we can represent the set of components in each models in a form that can be searched in time proportional to the total length of the strings sought. For my implementation I implement the set as a list and search the list for each component, which means that the time required is, in the worst case, proportional to the size of the set. As long as the sets remain small, more sophisticated methods

[3] My model-building program is currently written in Perl, which is notoriously slow. I believe that it would run in a fraction of the time if I re-coded it in a more effi cient language and took more care to make it run quickly.

are not necessary.

A Discussion of Error Types

One important property of the models generated by this method is that they are one-sided: they are intended to correctly predict if a file will have a particular property, rather than predicting if it will *not*. Many people find this idea somewhat non-intuitive, and therefore a more detailed explanation is warranted. The essential observation is that if the model is imperfect, it will make errors and these errors may attenuate the benefit of the correct predictions.

For a binary prediction, there are only two possible errors: false positive and false negative. A false positive occurs when the model predicts YES but the true value is NO. Similarly, a false negative occurs when the model predicts NO but the true value is YES.

For the purposes for which I assume these models will be used, if the prediction is NO then the default behavior will be used. Only if the prediction is YES will the behavior of the system differ from the default. If this assumption holds, then the behavior of the system will be *no worse* then the default behavior when the prediction is NO, whether or not this prediction is correct. If the prediction is YES and the prediction is correct then the behavior of the system is improved. If it prediction is NO and the prediction is incorrect, however, I must assume that the behavior of the system is worsened by the mis-prediction.[4] Therefore I consider the false positive rate to be as important as the overall error rate – a model with a high false negative rate might make it difficult to properly recognize when an optimization might be possible, but a model with a high false positive rate may easily make the system behave worse.

The one-sided test is a simple way to bias the models away from false positives. In effect, I am assuming that the answer is always NO and only predict YES when there is a pattern present in the filename that is usually present only in the names of other files for which the correct answer is YES.

An examination of the errors from the model shows that predictions are, for most predicates, more likely to be false negatives than false positives. From the perspective of file system designers, this is a valuable property. It means that if I predict that I *should* perform some (potentially costly) modification to the way a file system object is handled, I will rarely be wrong. On the other hand, it means that I may miss some opportunities to make optimizations – but in this case, what I've done is no worse than the default.

Another side effect of ignoring the components of the file names for files that do not satisfy the predicate is that the models are asymmetric with respect to their training. For example, consider a file system containing 100 zero-length files whose names all end in `.zero`, and 100 large files with randomly chosen names. Training for the predicate "this file is zero length" will start with a set containing 100 `.zero` suffixes, and construct a model that predicts that every file that ends in `.zero` will be zero length. Training for the predicate "this file is *not* zero length" will start with a list containing 100 random suffixes. All of these will be culled, because there is no clear pattern, and an empty model will result. Thus the overall accuracy of the model may depend entirely on the sense of the predicate. Note however that in the latter case, the resulting empty model will always answer NO and therefore never incur a false positive.

If overall accuracy is the goal, it is easy enough to create models for both the predicate and its logical negation and test both to see which is more accurate. However, there is a drawback to this approach – the bias against false positives is reversed if the negation is used. It is better to use a more intelligent method of choosing not only the most accurate model, but one whose expected errors are of the least deleterious type.

[4] If the behavior is improved despite the mis-prediction then this implies that the new behavior is better in either case, and therefore it should be used as the default behavior.

Predicate	Description
`lock`	True if the file is created, never written into, and deleted within five seconds of creation.
`write-only`	True if the file is created, and written into, and the read/write ratio for the file is less than 0.2.
`read-only`	True if the file is created, and written into, but the read/write ratio for the file is greater than 20.
`amtime<=0`	True if at the most recent observation the access time for the file is less than or equal to the modification time (the file has been more recently written than read).
`amtime>0`	True if at the most recent observation the access time for the file is greater than the modification time (the file has been read more recently than written).

Table 6.4: Descriptions of the predicates.

6.3.5 The Accuracy of the Models

To test the effectiveness of my algorithm, I generated and tested models for a variety of predicates from the CAMPUS, DEAS03, and EECS03 traces. In Tables 6.5, 6.6 and 6.7 I show the accuracy of the models in predicting a variety of properties for files created during the testing period. The models were trained on traces from a Monday and then tested on the traces from the following four weekdays. Most of the predicates listed in these tables are self-explanatory (for example, `size=zero` means that the size of the file was always zero), but several that are more complicated are explained in Table 6.4.

Up to this point, I have only discussed predictions about a file as whole, but my method can also be used to predict the properties of the names of a file. For example, it is possible to use the same techniques to predict whether a file name will live for less than five seconds before being deleted or renamed. This may be important because the maintenance of directories (which must be updated every time a file is created, deleted, or renamed) can be a significant fraction of the workload. For example, the common protocol for creating lockfiles is to create an empty file with a unique name, then make a link from a known name to the unique name (thereby giving the file two distinct names). If the link succeeds, then the lock has been acquired, and the original name is deleted. This means that the file that implements the lock lives a different length of time than either of the names it has during its existence.

The accuracy of the predictions about the lifespan of file names is shown in Table 6.8. One particularly interesting thing is that in many cases the predictions about names are much more accurate than the predictions about files. For example, on CAMPUS the predictions for whether a file will live for less than one second are much worse than simply guessing based on the observed probability. The predictions about whether a file name will live for less than one second, however, are more accurate than simply guessing based on the most likely outcome.

Note that the models used in Tables 6.5, 6.6, 6.7, and 6.8 are constructed using the default values of the `minfrac` and `mincount` parameters. The default parameters do reasonably well across a wide variety of predicates, but the overall accuracy and in particular the rate of false positives is often dependent on the exact values chosen for these parameters.

Tables 6.9, 6.10, 6.11, 6.12, 6.13, and 6.14 demonstrate the sensitivity of the accuracy of the predictions and the type of error to the threshold probability used by the model builder to select which patterns to use and which patterns to discard. As the threshold rises, the false positive rate tends to decrease and in many cases the overall accuracy increases – at least to a certain point. If `minfrac` is too high, then useful rules may be omitted entirely, and overall accuracy falls precipitously and the number of false negatives soars. Note that the number of false positives never increases when a rule is removed, because by removing a rule I am removing an opportunity to make a positive

CAMPUS File Predictions Trained 10/22/2001, Tested 10/23-10/26/2001						
Predicate	**Correct**	**Occurs**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	87.13%	81.87%	81.87%	29.02%	12.87%	12.73%
`size=zero`	99.78%	94.62%	94.62%	95.93%	0.22%	0.04%
`0<size<=16k`	99.10%	4.69%	95.31%	80.78%	0.90%	0.14%
`0<size<=64k`	98.94%	5.04%	94.96%	78.99%	1.06%	0.14%
`0<size<1M`	98.85%	5.29%	94.71%	78.34%	1.15%	0.12%
`size>16k`	99.55%	0.69%	99.31%	35.33%	0.45%	0.06%
`size>64k`	99.61%	0.34%	99.66%	-13.69%	0.39%	0.11%
`size>1M`	99.93%	0.09%	99.91%	23.77%	0.07%	0.00%
`amtime<=0`	98.61%	4.43%	95.57%	68.68%	1.39%	0.32%
`amtime>0`	99.33%	0.68%	99.32%	2.21%	0.67%	0.02%
`lifespan<=1`	28.71%	76.38%	76.38%	-201.89%	71.29%	1.31%
`lifespan<=5`	85.17%	83.99%	83.99%	7.36%	14.83%	12.78%
`lifespan<=30`	86.87%	86.79%	86.79%	0.61%	13.13%	10.64%
`1<lifespan<=30`	89.60%	10.40%	89.60%	-0.01%	10.40%	0.00%
`lifespan>1`	78.52%	23.62%	76.38%	9.04%	21.48%	0.65%
`lifespan>5`	84.27%	16.01%	83.99%	1.75%	15.73%	0.07%
`write-only`	97.86%	2.60%	97.40%	17.67%	2.14%	0.18%

Table 6.5: The accuracy of predictions for new files on CAMPUS.

The **Occurs** columns shows the percentage of files that actually satisfied the given predicate. The **Guess** column is accuracy rate for a strategy that always predicts that the outcome will be whatever outcome appeared most frequently in the training data. (It will always predict NO if **Occurs** for the training day is less than 50%, or YES otherwise. The **Δ-error** column shows the improvement in accuracy gained by using our models rather than the **Guess** prediction, expressed as a percentage of the **Guess** error rate. The number is 100% if the predictions are perfect, and may be less than zero if the predictions are less accurate than the **Guess** strategy. The **Incorrect** column is simply 100-**Correct**. The **FalsePos** is the percentage of the predictions that were false positives.

DEAS03 File Predictions Trained 02/24/2003, Tested 02/25-02/28/2003						
Predicate	**Correct**	**Occurs**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
lock	89.69%	51.60%	51.60%	78.69%	10.31%	7.67%
size=zero	97.20%	59.33%	59.33%	93.11%	2.80%	0.05%
0<size<=16k	82.58%	34.91%	65.09%	50.10%	17.42%	2.10%
0<size<=64k	83.19%	38.88%	61.12%	56.76%	16.81%	0.37%
0<size<1M	82.44%	40.14%	59.86%	56.25%	17.56%	0.11%
size>16k	94.61%	5.76%	94.24%	6.39%	5.39%	0.25%
size>64k	98.25%	1.79%	98.21%	1.94%	1.75%	0.09%
size>1M	99.48%	0.52%	99.48%	0.00%	0.52%	0.00%
amtime<=0	83.42%	34.00%	66.00%	51.22%	16.58%	2.48%
amtime>0	93.76%	6.51%	93.49%	4.17%	6.24%	0.07%
lifespan<=1	72.14%	67.75%	67.75%	13.62%	27.86%	10.32%
lifespan<=5	73.98%	70.76%	70.76%	11.02%	26.02%	7.75%
lifespan<=30	78.63%	78.63%	78.63%	-0.02%	21.37%	1.50%
1<lifespan<=30	89.16%	10.88%	89.12%	0.34%	10.84%	0.01%
lifespan>1	78.30%	32.25%	67.75%	32.72%	21.70%	4.40%
lifespan>5	81.08%	29.24%	70.76%	35.28%	18.92%	4.54%
write-only	82.93%	36.62%	63.38%	53.39%	17.07%	1.21%

Table 6.6: The accuracy of predictions for new files on DEAS03.

See Table 6.5 for a description of each column.

EECS03 File Predictions Trained 02/24/2003, Tested 02/25-02/28/2003						
Predicate	**Correct**	**Occurs**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
lock	97.38%	26.53%	73.47%	90.14%	2.62%	1.79%
size=zero	97.11%	26.66%	73.34%	89.15%	2.89%	2.04%
0<size<=16k	73.39%	63.45%	63.45%	27.20%	26.61%	4.25%
0<size<=64k	73.99%	70.50%	70.50%	11.85%	26.01%	0.73%
0<size<1M	73.99%	72.88%	72.88%	4.10%	26.01%	0.22%
size>16k	91.07%	9.89%	90.11%	9.77%	8.93%	0.20%
size>64k	97.85%	2.84%	97.16%	24.42%	2.15%	0.23%
size>1M	99.67%	0.46%	99.54%	28.57%	0.33%	0.01%
amtime<=0	77.14%	60.35%	60.35%	42.34%	22.86%	4.16%
amtime>0	89.04%	12.99%	87.01%	15.63%	10.96%	1.46%
lifespan<=1	68.42%	62.70%	62.70%	15.34%	31.58%	1.20%
lifespan<=5	67.91%	64.26%	64.26%	10.22%	32.09%	0.97%
lifespan<=30	66.34%	66.29%	66.29%	0.14%	33.66%	0.75%
1<lifespan<=30	96.40%	3.59%	96.41%	-0.23%	3.60%	0.04%
lifespan>1	66.65%	37.30%	62.70%	10.61%	33.35%	1.13%
lifespan>5	68.03%	35.74%	64.26%	10.53%	31.97%	1.24%
write-only	76.28%	59.38%	59.38%	41.60%	23.72%	4.08%

Table 6.7: The accuracy of predictions for new files on EECS03.

See Table 6.5 for a description of each column.

CAMPUS Name Predictions Trained 10/22/2001, Tested 10/23-10/26/2001						
Predicate	**Correct**	**Occurs**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	95.05%	89.93%	89.93%	50.87%	4.95%	4.82%
`size=zero`	99.79%	94.75%	94.75%	95.90%	0.21%	0.04%
`lifespan<=1`	91.26%	90.10%	90.10%	11.69%	8.74%	6.77%
`lifespan<=5`	93.00%	92.13%	92.13%	11.10%	7.00%	4.89%
`lifespan<=30`	93.17%	93.41%	93.41%	-3.66%	6.83%	4.16%
DEAS03 Name Predictions **Trained 02/24/2003, Tested 02/25-02/28/2003**						
Predicate	**Correct**	**Occurs**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	96.72%	58.74%	58.74%	92.05%	3.28%	0.59%
`size=zero`	97.15%	59.40%	59.40%	92.99%	2.85%	0.04%
`lifespan<=1`	80.75%	77.59%	77.59%	14.10%	19.25%	0.96%
`lifespan<=5`	80.69%	78.43%	78.43%	10.47%	19.31%	0.57%
`lifespan<=30`	79.57%	80.48%	80.48%	-4.65%	20.43%	0.10%
EECS03 Name Predictions **Trained 02/24/2003, Tested 02/25-02/28/2003**						
Predicate	**Correct**	**Occurs**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	97.49%	27.56%	72.44%	90.91%	2.51%	1.40%
`size=zero`	97.47%	27.62%	72.38%	90.84%	2.53%	1.38%
`lifespan<=1`	69.40%	64.46%	64.46%	13.91%	30.60%	0.63%
`lifespan<=5`	68.66%	65.75%	65.75%	8.50%	31.34%	0.38%
`lifespan<=30`	67.23%	68.19%	68.19%	-3.01%	32.77%	0.34%

Table 6.8: The accuracy of predictions about the longevity of new file names on CAMPUS, DEAS03, and EECS03. See Table 6.5 for a description of each column.

prediction. In some cases, the accuracy rate for the models with a high **minfrac** is approximately the same as that for guessing, with the important difference that nearly all of the errors from the predictions are false negatives (instead of a mix of false positives and false negatives).

6.4 Decision Tree-Based Predictions

Shortly after reporting my discovery of the strong relationship between file names and file properties and the possibility of using these relationships to perform predictions about new files [24, 25], Eno Thereska and Michael Mesnier from the Parallel Data Laboratory at Carnegie Mellon University proposed to extend this idea by using additional attributes (such as permission mode, owner, and group) and by using more complex modeling methods via their Attribute-Based Learning Environment (ABLE) [47]. The result has been a fruitful collaborative exploration of this topic [26, 46].

This section presents an overview of the methods used by Mesnier and Thereska in ABLE to develop their models and discusses the differences and trade-offs with respect to the simple name-based method described in Section 6.3.

ABLE uses a modified decision tree model, using a method based on ID3 heuristics [11]. Decision trees are widely used in artificial intelligence and machine learning. A decision tree is an annotated n-ary tree. The internal nodes represent attributes, the edges represent possible values of that attribute, and the leaf nodes represent decisions. Using a decision tree to make a prediction is simple: to find the decision corresponding to a set of attribute values, search the tree for the leaf corresponding to those values.

Despite the simplicity of the tree and the decision-making process that uses it, there are myriad methods for constructing the tree. In this section, I describe the method that we have used in our research and discuss the tradeoffs between our method and other related methods. Most of the work of developing and implementing the heuristics and evaluating the accuracy and other characteristics of the resulting models was done by Michael Mesnier and Eno Thereska at Carnegie Mellon University, and so I will omit a deep discussion of these issues.

The initial algorithm used by Mesnier & Thereska was based on ID3, but the final algorithm differs in several important areas and therefore should be considered distinct.

6.4.1 Constructing a Decision Tree

Imagine that the results of a trace have been tabulated and are shown in Table 6.15. Each row in the table represents one file, and each of the columns on the left represents an attribute. The rightmost column tells whether or not the property of interest holds for this file. In this context, the basic recursive tree construction algorithm is easy to describe and is shown in Figure 6.4.

The process for building a tree from the data shown in Table 6.15 is illustrated in Figure 6.5.

Note that the measure of association required by the recursive step is calculated for the contents of each subtree, not the table as a whole. Therefore it is possible for the same attribute to appear at different levels in the tree (although each path from the root to a leaf must use each attribute exactly once). This possibility is not illustrated in Figure 6.5 because there are only two attributes and therefore after one has been chosen to be the root, there is no remaining choice about what attribute is selected in the next step.

Also note that Table 6.15 is far too small for the χ-squared measure of association (or any other method of association) to give accurate results. Please imagine, for the sake of this example, that it is possible to robustly estimate the association for such a small data set, and also imagine that the `Mode` is more strongly associated with `Write-Only` than is `Suffix`.

CAMPUS File Predictions Trained 10/22/2001, Tested 10/23-10/26/2001						
Predicate	**minfrac**	**Correct**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	0.75	87.13%	81.87%	29.02%	12.87%	12.73%
	0.80	87.13%	81.87%	29.02%	12.87%	12.73%
	0.85	73.74%	81.87%	-44.89%	26.26%	9.79%
	0.90	18.98%	81.87%	-346.97%	81.02%	0.10%
	0.95	18.27%	81.87%	-350.86%	81.73%	0.02%
	0.97	18.20%	81.87%	-351.24%	81.80%	0.01%
	0.99	18.21%	81.87%	-351.23%	81.79%	0.01%
	1.00	18.21%	81.87%	-351.23%	81.79%	0.01%
`size=zero`	0.75	99.78%	94.62%	95.93%	0.22%	0.04%
	0.80	99.78%	94.62%	95.93%	0.22%	0.04%
	0.85	99.78%	94.62%	95.93%	0.22%	0.04%
	0.90	99.78%	94.62%	95.95%	0.22%	0.04%
	0.95	99.78%	94.62%	95.95%	0.22%	0.04%
	0.97	99.78%	94.62%	95.86%	0.22%	0.03%
	0.99	99.78%	94.62%	95.86%	0.22%	0.03%
	1.00	99.78%	94.62%	95.85%	0.22%	0.01%
`0<size<=16k`	0.75	99.10%	95.31%	80.78%	0.90%	0.14%
	0.80	99.10%	95.31%	80.78%	0.90%	0.14%
	0.85	99.10%	95.31%	80.78%	0.90%	0.14%
	0.90	98.71%	95.31%	72.43%	1.29%	0.05%
	0.95	98.69%	95.31%	72.12%	1.31%	0.04%
	0.97	98.63%	95.31%	70.79%	1.37%	0.04%
	0.99	95.78%	95.31%	10.02%	4.22%	0.01%
	1.00	95.78%	95.31%	9.99%	4.22%	0.01%
`lifespan<=5`	0.75	85.09%	83.99%	6.88%	14.91%	12.84%
	0.80	85.17%	83.99%	7.36%	14.83%	12.78%
	0.85	71.73%	83.99%	-76.58%	28.27%	9.87%
	0.90	16.97%	83.99%	-418.60%	83.03%	0.11%
	0.95	16.26%	83.99%	-423.02%	83.74%	0.03%
	0.97	16.20%	83.99%	-423.45%	83.80%	0.02%
	0.99	16.09%	83.99%	-424.11%	83.91%	0.01%
	1.00	16.09%	83.99%	-424.11%	83.91%	0.01%

Table 6.9: The effect of **minfrac** on the prediction accuracy and false positive rate for selected predicates about files on CAMPUS.

DEAS03 File Predictions Trained 02/24/2003, Tested 02/25-02/28/2003						
Predicate	**minfrac**	**Correct**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	0.75	89.69%	51.60%	78.69%	10.31%	7.67%
	0.80	89.69%	51.60%	78.69%	10.31%	7.67%
	0.85	89.69%	51.60%	78.69%	10.31%	7.67%
	0.90	75.18%	51.60%	48.73%	24.82%	0.78%
	0.95	75.05%	51.60%	48.45%	24.95%	0.76%
	0.97	74.77%	51.60%	47.87%	25.23%	0.65%
	0.99	72.90%	51.60%	44.01%	27.10%	0.38%
	1.00	60.33%	51.60%	18.05%	39.67%	0.24%
`size=zero`	0.75	97.20%	59.33%	93.11%	2.80%	0.05%
	0.80	97.20%	59.33%	93.11%	2.80%	0.05%
	0.85	97.20%	59.33%	93.11%	2.80%	0.05%
	0.90	97.20%	59.33%	93.11%	2.80%	0.05%
	0.95	97.20%	59.33%	93.11%	2.80%	0.05%
	0.97	97.20%	59.33%	93.11%	2.80%	0.05%
	0.99	97.20%	59.33%	93.11%	2.80%	0.05%
	1.00	86.08%	59.33%	65.78%	13.92%	0.00%
`0<size<=16k`	0.75	82.58%	65.09%	50.10%	17.42%	2.10%
	0.80	82.58%	65.09%	50.10%	17.42%	2.10%
	0.85	82.30%	65.09%	49.30%	17.70%	2.02%
	0.90	80.07%	65.09%	42.91%	19.93%	0.50%
	0.95	79.06%	65.09%	40.03%	20.94%	0.36%
	0.97	78.58%	65.09%	38.64%	21.42%	0.34%
	0.99	69.17%	65.09%	11.70%	30.83%	0.03%
	1.00	69.17%	65.09%	11.70%	30.83%	0.03%
`lifespan<=5`	0.75	73.98%	70.76%	11.02%	26.02%	7.75%
	0.80	73.98%	70.76%	11.02%	26.02%	7.75%
	0.85	73.98%	70.76%	11.02%	26.02%	7.75%
	0.90	58.99%	70.76%	-40.23%	41.01%	0.80%
	0.95	58.70%	70.76%	-41.23%	41.30%	0.83%
	0.97	56.92%	70.76%	-47.33%	43.08%	0.63%
	0.99	54.52%	70.76%	-55.52%	45.48%	0.35%
	1.00	45.50%	70.76%	-86.36%	54.50%	0.24%

Table 6.10: The effect of **minfrac** on the prediction accuracy and false positive rate for selected predicates about files on DEAS03.

EECS03 File Predictions Trained 02/24/2003, Tested 02/25-02/28/2003						
Predicate	**minfrac**	**Correct**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lock`	0.75	97.03%	73.47%	88.81%	2.97%	2.15%
	0.80	97.38%	73.47%	90.14%	2.62%	1.79%
	0.85	98.87%	73.47%	95.74%	1.13%	0.30%
	0.90	98.75%	73.47%	95.28%	1.25%	0.26%
	0.95	98.86%	73.47%	95.70%	1.14%	0.15%
	0.97	98.90%	73.47%	95.87%	1.10%	0.10%
	0.99	97.64%	73.47%	91.12%	2.36%	0.08%
	1.00	83.43%	73.47%	37.53%	16.57%	0.00%
`size=zero`	0.75	97.11%	73.34%	89.15%	2.89%	2.04%
	0.80	97.11%	73.34%	89.15%	2.89%	2.04%
	0.85	98.94%	73.34%	96.01%	1.06%	0.20%
	0.90	98.81%	73.34%	95.54%	1.19%	0.17%
	0.95	98.92%	73.34%	95.96%	1.08%	0.05%
	0.97	98.97%	73.34%	96.13%	1.03%	0.00%
	0.99	98.97%	73.34%	96.13%	1.03%	0.00%
	1.00	98.97%	73.34%	96.13%	1.03%	0.00%
`0<size<=16k`	0.75	73.17%	63.45%	26.60%	26.83%	4.33%
	0.80	73.39%	63.45%	27.20%	26.61%	4.25%
	0.85	73.81%	63.45%	28.34%	26.19%	4.16%
	0.90	66.71%	63.45%	8.93%	33.29%	0.94%
	0.95	65.55%	63.45%	5.75%	34.45%	0.56%
	0.97	65.86%	63.45%	6.61%	34.14%	0.48%
	0.99	46.25%	63.45%	-47.06%	53.75%	0.02%
	1.00	45.34%	63.45%	-49.55%	54.66%	0.02%
`lifespan<=5`	0.75	67.91%	64.26%	10.22%	32.09%	0.97%
	0.80	67.91%	64.26%	10.22%	32.09%	0.97%
	0.85	67.91%	64.26%	10.22%	32.09%	0.97%
	0.90	67.91%	64.26%	10.20%	32.09%	0.67%
	0.95	67.75%	64.26%	9.75%	32.25%	0.66%
	0.97	67.86%	64.26%	10.06%	32.14%	0.59%
	0.99	66.17%	64.26%	5.33%	33.83%	0.13%
	1.00	49.89%	64.26%	-40.23%	50.11%	0.01%

Table 6.11: The effect of **minfrac** on the prediction accuracy and false positive rate for selected predicates about files on EECS03.

CAMPUS Name Predictions Trained 10/22/2001, Tested 10/23-10/26/2001						
Predicate	**minfrac**	**Correct**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lifespan<=5`	0.75	93.00%	92.13%	11.12%	7.00%	4.89%
	0.80	93.00%	92.13%	11.10%	7.00%	4.89%
	0.85	93.00%	92.13%	11.10%	7.00%	4.89%
	0.90	93.00%	92.13%	11.11%	7.00%	4.89%
	0.95	9.73%	92.13%	-1046.25%	90.27%	0.14%
	0.97	8.56%	92.13%	-1061.19%	91.44%	0.07%
	0.99	8.18%	92.13%	-1065.96%	91.82%	0.03%
	1.00	8.18%	92.13%	-1066.06%	91.82%	0.03%
`lock`	0.75	95.05%	89.93%	50.87%	4.95%	4.82%
	0.80	95.05%	89.93%	50.87%	4.95%	4.82%
	0.85	95.05%	89.93%	50.87%	4.95%	4.82%
	0.90	95.07%	89.93%	51.04%	4.93%	4.80%
	0.95	11.71%	89.93%	-776.63%	88.29%	0.13%
	0.97	10.66%	89.93%	-787.05%	89.34%	0.06%
	0.99	10.39%	89.93%	-789.67%	89.61%	0.03%
	1.00	10.38%	89.93%	-789.76%	89.62%	0.03%

Table 6.12: The effect of **minfrac** on the prediction accuracy and false positive rate for selected predicates about file names on CAMPUS.

DEAS03 Name Predictions Trained 02/24/2003, Tested 02/25-02/28/2003						
Predicate	**minfrac**	**Correct**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lifespan<=5`	0.75	80.69%	78.43%	10.47%	19.31%	0.57%
	0.80	80.69%	78.43%	10.47%	19.31%	0.57%
	0.85	80.69%	78.43%	10.47%	19.31%	0.57%
	0.90	80.69%	78.43%	10.47%	19.31%	0.57%
	0.95	80.69%	78.43%	10.47%	19.31%	0.57%
	0.97	80.69%	78.43%	10.47%	19.31%	0.57%
	0.99	80.69%	78.43%	10.47%	19.31%	0.57%
	1.00	52.69%	78.43%	-119.34%	47.31%	0.23%
`lock`	0.75	96.72%	58.74%	92.05%	3.28%	0.59%
	0.80	96.72%	58.74%	92.05%	3.28%	0.59%
	0.85	96.72%	58.74%	92.05%	3.28%	0.59%
	0.90	96.72%	58.74%	92.05%	3.28%	0.59%
	0.95	96.72%	58.74%	92.05%	3.28%	0.59%
	0.97	96.72%	58.74%	92.05%	3.28%	0.59%
	0.99	96.72%	58.74%	92.05%	3.28%	0.59%
	1.00	70.82%	58.74%	29.28%	29.18%	0.23%

Table 6.13: The effect of **minfrac** on the prediction accuracy and false positive rate for selected predicates about file names on DEAS03.

EECS03 Name Predictions Trained 02/24/2003, Tested 02/25-02/28/2003						
Predicate	**minfrac**	**Correct**	**Guess**	**Δ-error**	**Incorrect**	**FalsePos**
`lifespan<=5`	0.75	68.66%	65.75%	8.50%	31.34%	0.38%
	0.80	68.66%	65.75%	8.50%	31.34%	0.38%
	0.85	68.66%	65.75%	8.50%	31.34%	0.38%
	0.90	68.54%	65.75%	8.13%	31.46%	0.34%
	0.95	68.51%	65.75%	8.05%	31.49%	0.33%
	0.97	68.51%	65.75%	8.05%	31.49%	0.33%
	0.99	67.67%	65.75%	5.60%	32.33%	0.14%
	1.00	66.17%	65.75%	1.21%	33.83%	0.01%
`lock`	0.75	97.50%	72.44%	90.92%	2.50%	1.39%
	0.80	97.49%	72.44%	90.91%	2.51%	1.40%
	0.85	97.49%	72.44%	90.91%	2.51%	1.40%
	0.90	97.72%	72.44%	91.72%	2.28%	1.01%
	0.95	97.99%	72.44%	92.72%	2.01%	0.06%
	0.97	98.04%	72.44%	92.89%	1.96%	0.02%
	0.99	96.78%	72.44%	88.31%	3.22%	0.00%
	1.00	96.78%	72.44%	88.31%	3.22%	0.00%

Table 6.14: The effect of **minfrac** on the prediction accuracy and false positive rate for selected predicates about file names on EECS03.

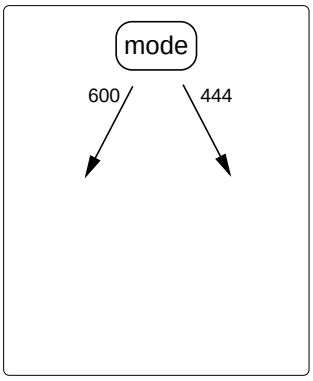
- **Base Case:**

  If there are no unused attributes in the table, then create and return a leaf node labeled YES if the majority of the rows in the table have the property value of YES, NO otherwise. (If the number of YES rows is the same as the number of NO rows then arbitrarily select that latter.)

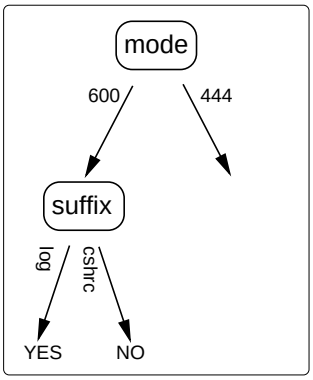
- **Recursive Case:**

  1. Choose the attribute α that has the highest association with the property.
     We have found that the χ-squared measure of association works well, but other metrics are certainly possible.
  2. Mark attribute α as used.
  3. Create a new root node R for the tree.
  4. For each unique value v of attribute α that appears in the table:
     (a) Recursively create a decision tree R_v, using the ABLE algorithm, for all rows of the table such that v is the value of α in that row.
     (b) Attach each R_v to R with an edge labeled v.

Figure 6.4: The algorithm for constructing the ABLE decision tree.

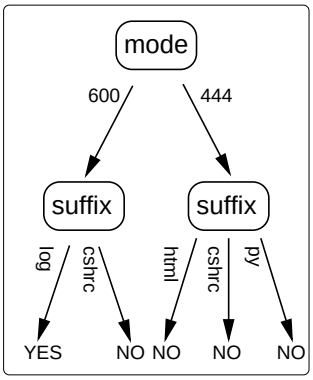


Step 1: since `mode` is the attribute most strongly associated with the outcome, select `mode` as the root attribute. Two unique values of `mode` appear in the training (`600` and `444`), so this tree will have children for each of these possibilities.



Step 2: Build a tree for the subset of the training that includes only files with a `mode` of `600`. `suffix` is the only remaining attribute, so it is the root of this tree. There are only two observed suffixes (`log` and `cshrc`), and the root will have children for each.

Because there are no more attributes to use, the children of the `suffix` node are leafs. The majority of files with a suffix of `log` are write-only, so this leaf is assigned a value of YES. The majority of files with a suffix of `cshrc` are not write-only, so this leaf is assigned a value of NO.



Step 3: Build a tree for the subset of the training that includes only files with a `mode` of `444`. `suffix` is the only remaining attribute, so it is the root of this tree. There are three observed suffixes (`html`, `cshrc`, and `py`), and the root will have children for each.

Figure 6.5: Building an ABLE tree from the data shown in Table 6.15.

Attributes		Property
Mode	**Suffix**	**Write-Only?**
600	cshrc	NO
600	cshrc	NO
600	cshrc	YES
444	cshrc	NO
444	html	NO
600	log	YES
600	log	YES
444	py	NO

Table 6.15: The data used to build the decision tree illustrated in Figures 6.5, 6.6, 6.7.

In this example, only two attributes are considered: **Mode** is the permission mode, and **Suffix** represents the suffix of the file name. The "Write-Only" property holds for files that are written but never read.

6.4.2 Pruning the Decision Tree

In the worst case, this decision tree may have as many leaf nodes as there were files in the training data. Even though the expected case is much smaller (because many files tend to share attributes) the resulting tree is typically large and unwieldy (especially compared with the size of the name-based models).

The most obvious form of pruning is to remove nodes that contribute nothing to the predictive power of the tree. In the final tree from Figure 6.5, the "suffix" node on the right satisfies this property – since all observed files that have a mode of 444 lead to the same outcome, regardless of suffix, there is nothing learned by looking at the suffix. Therefore this node (and its descendants,if any) can be removed and replaced by one of its descendant leafs, as shown in Figure 6.6.

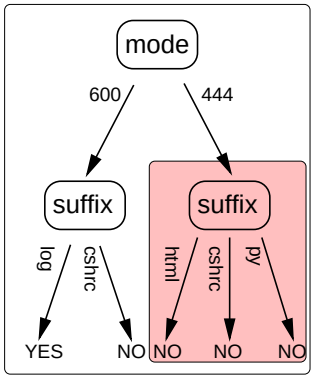
A more aggressive form of pruning is to remove all nodes that have a *purity* above a certain threshold. The *purity* of a node is a measurement of the bias in the observed probabilities that the property is satisfied across all the descendants. A node whose outcome is completely biased in either direction (such as the suffix node in the previous paragraph) is perfectly pure and can be removed without changing the predictions made by the tree. Removing a highly (but not completely) pure node alters the predictions, but it may be worthwhile to do so in order to decrease the size of the tree.

For example, imagine that 99% of the files created during the training period have a property X. Therefore the root node of the decision tree is highly pure: 99% of its descendants (weighted by the number of observations in the trace) have the `Yes` outcome for property X. Therefore, the entire tree could be replaced with a single leaf labeled with `Yes`, and still have a prediction accuracy of 99%.

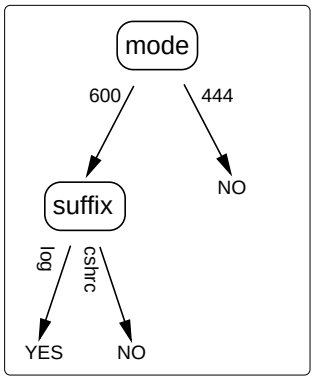
An even more extreme form of pruning is simple to remove nodes of high purity until the tree is reduced to whatever size is required. For very large trees, this may be the only way to reduce the size to something manageable.

Blindly removing nodes from the tree raises the possibility of disaster because purity alone is not necessarily a good indication of how important a node is to accurate predications. The best method to prune a decision a tree is by cross-validation – testing the estimated accuracy of the tree (by testing it against the training data) before and after removing each node and not removing any nodes that cause performance to plummet. This method is expensive, however, and it appears that for our data the results are nearly indistinguishable. We take advantage of this fact to prune the tree in a simple and efficient manner.

As an implementation detail, note that the purity of each node is computable *before* its descendants have been



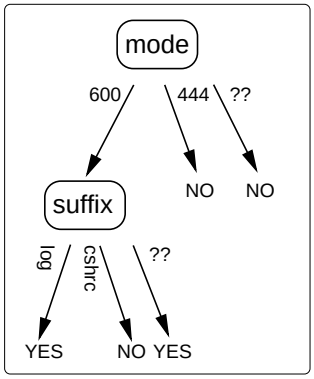
Step 1: Identify pure nodes.



Step 2: Replace pure nodes with the outcome predicted by the majority of its descendants.

Note that the majority is based on the number of observations corresponding to each leaf, not the number of leaves.

Figure 6.6: Pruning the decision tree.



In the original data, the majority of the files did not have the `write-only` property, and so the default action for a mode that was not observed in the training is to predict NO. For a file with a mode of `600` but whose suffix does not appear for any file in the training set whose mode is `600`, the model predicts YES because the majority of the files in the training set that have a mode `600` do have the `write-only` property.

Figure 6.7: Augmenting the decision tree to handle cases that do not appear in the training set.

created. This means that instead of creating the complete decision tree and then pruning, the tree may be preemptively pruned as it is being created (and therefore avoid creating any of the nodes that would eventually be pruned). In practice this is much more efficient than creating and pruning the tree in two separate steps. Our implementation does recognize nodes that are completely pure and not generate their descendents.

6.4.3 Augmenting the Tree

The problem with the decision tree created by the algorithm as it has been described is that it may be too constrained – the tree does not have any way to predict whether the property holds for any files that have attribute values that did not appear in the training. For example, the tree shown in Figure 6.6 is not capable of making a prediction about a file that has a mode of `755`, nor can it make a prediction about file that has a mode of `600` and a suffix of `.c`.

In order to handle these cases, the tree is augmented with additional "default" nodes to handle cases that were not observed in the training. The predicted outcome for each of these nodes is based on the conditional probability of the outcome occurring given the values of the attributes that led to this node. An example of this is shown in Figure 6.7.

6.4.4 The Accuracy of ABLE

Table 6.16 shows that accuracy of the models created by ABLE compared to name-based models described in Section 6.3. For the size predicates, the name-based models actually do better than the ABLE predictor on CAMPUS, but in every other case the ABLE predictor benefits from using information from the additional attributes to make better predictions. Note that the name-based model is particularly bad at predicting which files will live for less than one second on CAMPUS (although it is much better at predicting which files will live for less than one second, as shown in Table 6.5) while ABLE is much better. Even ABLE, however, is a worse predictor for whether a file will live for less than one second than guessing based on the most likely outcome in the training (75.58%).

It is not surprising that more using more attributes gives better models in most cases, and therefore it is not surprising that the ABLE models give better predictions than my name-based models. Although the ABLE models are more accurate, the name-based models have several useful properties:

- The name-based models are typically several orders of magnitude smaller than the ABLE models (a few hundred

	CAMPUS		DEAS03		EECS03	
Predicate	**ABLE**	**NAME**	**ABLE**	**NAME**	**ABLE**	**NAME**
size=0	98.57%	99.66%	98.97%	97.85%	97.06%	96.05%
0<size≤16KB	98.83%	99.02%	95.42%	84.28%	89.96%	77.28%
lftmd≤1s (file)	72.95%	29.28%	88.16%	73.69%	93.60%	62.04%

Table 6.16: A comparison of the accuracy of the ABLE and name-based predictions for several properties.

For the size predicates, the name-based models actually do better than the ABLE predictor on CAMPUS, but in every other case the ABLE predictor is able to use the information from the additional attributes to make better predictions. Note that the name-based model is particularly bad at predicting which files will live for less than one second on CAMPUS (although it is much better at predicting which files will live for less than five seconds, as shown in Table 6.5) while ABLE is much better. Even ABLE, however, is a worse predictor than guessing based on the most likely outcome in the training (75.58%).

Both the ABLE and name-based models are trained with data from an entire day (Monday 10/22/01 for CAMPUS, Monday 02/24/03 for DEAS03 and EECS03) and then tested with the data from the entire following day. Note that the number for the name-based model differ from those reported earlier in Tables 6.5, 6.6 and 6.7 because in this table the training and testing traces are for entire days instead of only the peak hours.

---

bytes versus tens or hundreds of KB).

- Incremental training is easy with the name-based models: adding (or subtracting) training data from the model is simply a matter of incrementing (or decrementing) the occurrence count of the components that appear in the training. For ABLE, the tree must be reconstructed when training is performed. (It is possible to perform incremental training by simply updating the outcome at the leaf nodes, but this does not permit the attributes to be re-ordered in the tree.
- The name-based models have the property that they can easily be biased in such a way to minimize false positives. Decision trees can be biased (most easily by introducing the bias into the measurement of the purity of each subtree) but it is a more complicated exercise.

6.5 Summary

In this chapter I have shown how to construct name-based and attribute-based models that can predict many aspects of a new file's future life, including how much data it will contain, how long it will live, and its access patterns. In the next chapter, I will show how to use these predictions to improve network storage server performance.

Chapter 7

Using File-Based Predictions

In this chapter, I explore the use of name-based predictions to increase the locality of block reference by anticipating which blocks are likely to be hot and grouping them in the same cylinder.

I use two methods to identify hot data blocks. The first method, which I call *HotName*, automatically classifies as hot any file that the models predict will be short-lived and/or zero-length. For this type of file, the overhead of creating and maintaining the inode and name of the file (i.e., the directory entry for the file) can be a large fraction of the cost incurred by the file, and therefore there may be benefit to reducing this overhead. The second method, which I call *HotDir*, predicts which directories are most likely to contain files that have the HotName property. Since these directories are where the names for the HotName files will be entered, there may be benefit from identifying them as well.

The model that I use for HotDir is constructed via a method similar to the name-based predictions, but unfortunately my prototype requires some external logic because it is focused on files and does not currently gather as much information about directories. The heuristic that I use is to treat directories identified as home directories, mail spool directories, and web cache directories as hot directories. (The name-based predictor is capable of identifying directories with names like `mail` and `cache`, but does not currently have any way to identify home directories.)

To test the effect of HotDir and HotName, I have modified the FreeBSD implementation of FFS so that it uses the name-based predictor to predict whether each new directory has the HotDir property and whether each new file has the HotName property. If so, it attempts to allocate the blocks used to implement that file or directory in a small designated area of the disk. My goal is to measure the increase in the number of accesses to this area of the disk when I use policies guided by HotDir and HotName.

7.1 The Benchmark

I use two systems as my testbed. Both have a 1 GHz Pentium III processor, 1 GB of RAM, and run FreeBSD 4.8p3. The experiments use the FreeBSD implementation of FFS with 16KB blocks and soft-updates enabled [30]. I have instrumented the device driver for the disk so that it keeps a count of how many reads and writes are done for each 4KB logical disk block.

I tally counts for 4KB logical blocks instead of 16KB blocks because the counts for 16KB blocks may be misleading. Different clients use a different buffer size in their read and write operations, and therefore one client might write 16KB in a single operation (and therefore tally an operation count of one) while another might perform the same write as two 8KB writes (and therefore tally an operation count of two for the same 16KB block even though the number of physical disk blocks touched by each client is identical). In an ideal experiment, I would measure the operation count of each physical 512-byte disk block. In my current experimental setup, I cannot afford to dedicate the memory required to represent a read and write count for each 512-byte disk block. A 4KB logical block size is a

reasonable compromise.

The test file systems are built on a dedicated 40GB partition of a Western Digital 120GB WE12000BB-16DAA0 disk, using the subsnapshot method described in Chapter 3.6.4. The reason I use a 40GB partition is that this is large enough to hold all of the subsnapshots that are needed, but small enough to reduce the variation in run times that can be attributed to the ZCAV effect (described in Chapter 5.3.3). The size of the subsnapshots ranges from 10GB (for CAMPUS and EECS03) to 37GB (for DEAS03).

The subsnapshots are constructed from traces that run from 10:00am until 4:00pm for four consecutive weekdays (Tuesday through Friday). The models used are trained on a trace for the hours from the preceding Monday. For CAMPUS, the training day is 10/22/2001 and the testing days are 10/23-10/26/2001. For DEAS03 and EECS03, the training day is 03/24/2003 and the testing days are 03/25-03/28/2003.

For each day in the test, the file system is constructed from scratch (starting with a new file system) from the subsnapshot for that day, and then the trace is replayed. The first hour of each trace replay is used only to warm the cache. At the end of the first hour, the block counters are reset, and the test begins in earnest and runs for the final five hours. Each trace is run with the default configuration, HotName, HotDir, and the combination of HotName and HotDir.

7.2 Locality Results

I designate a 4MB region as the target area for hot objects. My evaluation examines the distribution of actual accesses to the disk and compares the percentage that go to the target area to the theoretically maximum number of accesses that would go to the hottest 4MB region given perfect knowledge (i.e., if the hottest 1024 4KB blocks on the disk were allocated in the target region).

As shown in Tables 7.1, 7.2, and 7.3, both HotDir and HotName improve locality compared to the default layout policy, and using both heuristics at the same time is usually an improvement over using either one alone.

The “Default” rows show the number of operations that fall into the designated region when none of the name-based heuristics are enabled. These percentages are often much higher than the laws of chance would suggest. This is due to the way that the default layout policy chooses blocks on the disk. By default, the area used by the designated region contains logical block zero of the disk, which is, by default, allocated for the first object created in the file system, which is the root directory. Using the default layout policies, the descendents of the root directory will be allocated in the same cylinder group. In some cases some of the hot directories are close descendents of the root and may, by virtue of the coincidental location of the designated region, end up in the designated region. Using a different designated region does not change the results significantly for the HotDir and HotName heuristics, but changes the results for the Default heuristic. The expected percentage of operations that fall within a random 4 MB region of the disk, computed by counting the number of distinct blocks that are actually accessed at least once per run is approximately 0.04% for CAMPUS, 0.02% for DEAS03, and 0.03% for EECS03.

The “Optimal” columns show how many operations would fall in the designated area if the blocks were perfectly reorganized so that the 1024 most frequently-accessed blocks were actually located in this area. In nearly all cases, the heuristics come close (and at worst within a factor of three) of achieving this level of optimality for writing.

Write locality is increased more than read locality; this is not surprising because directory contents are read-cached and the activity I am trying to optimize focuses on directory updates. Using both HotDir and HotName, the number of accesses is more than three-quarters that of the hottest possible region on CAMPUS, and within a factor of three on EECS03. On DEAS03 a large fraction of the disk activity is due to a small number of log files that are not recognized by either heuristic, and therefore the hottest possible 4MB region is more than ten times hotter than the designated region.

Remember that the goal of HotDir and HotName is not to reduce the number of disk operations, but only to increase the locality of these operations. The number of disk blocks read and written during each run is the almost the same for every replay of a particular trace, regardless of the combination of layout heuristics used. There is some

CAMPUS						
	Target			Optimal		
Condition	Total	Read	Write	Total	Read	Write
10/23/2001						
Default	0.209%	0.024%	0.630%	2.565%	0.313%	8.130%
HotDir	1.799%	0.077%	5.739%	2.418%	0.312%	7.663%
HotName	0.973%	0.012%	3.169%	2.488%	0.313%	7.882%
HotDir+HotName	2.248%	0.086%	7.183%	2.522%	0.312%	7.966%
10/24/2001						
Default	0.536%	0.023%	1.621%	2.511%	0.298%	7.552%
HotDir	1.608%	0.072%	4.869%	2.365%	0.299%	7.088%
HotName	0.945%	0.005%	2.933%	2.449%	0.298%	7.355%
HotDir+HotName	2.173%	0.085%	6.595%	2.477%	0.289%	7.430%
10/25/2001						
Default	0.446%	0.035%	1.356%	2.338%	0.294%	7.251%
HotDir	1.715%	0.083%	5.337%	2.225%	0.294%	6.883%
HotName	0.900%	0.014%	2.864%	2.282%	0.293%	7.060%
HotDir+HotName	1.699%	0.094%	5.256%	2.326%	0.293%	7.188%
10/26/2001						
Default	0.450%	0.041%	1.479%	2.359%	0.294%	7.997%
HotDir	1.720%	0.076%	5.875%	2.253%	0.294%	7.621%
HotName	0.899%	0.012%	3.137%	2.295%	0.288%	7.779%
HotDir+HotName	2.066%	0.085%	7.062%	2.350%	0.296%	7.929%

Table 7.1: The effect, on CAMPUS, of using the HotDir and HotName predictions on the number of operations that fall within the designated 4 MB region.

The “Target” columns show the percentage of total operations, read operations, and write operations that fall within the target 4MB region of the disk. The “Optimal” columns show the percentage of operations that would fall within this region if the blocks had been arranged optimally to maximize these percentages.

DEAS03						
	Target			Optimal		
Condition	Total	Read	Write	Total	Read	Write
02/25/2003						
Default	0.285%	0.007%	0.444%	29.676%	0.490%	46.513%
HotDir	1.069%	0.034%	1.659%	29.666%	0.494%	46.495%
HotName	0.547%	0.001%	0.858%	29.661%	0.488%	46.510%
HotDir+HotName	2.143%	0.033%	3.345%	29.644%	0.491%	46.479%
02/26/2003						
Default	3.175%	0.001%	4.833%	30.963%	0.460%	47.073%
HotDir	0.796%	0.026%	1.200%	30.925%	0.463%	47.034%
HotName	0.467%	0.020%	0.701%	30.942%	0.461%	47.055%
HotDir+HotName	3.983%	0.023%	6.054%	30.923%	0.472%	47.032%
02/27/2003						
Default	1.681%	0.006%	2.925%	26.089%	0.489%	45.421%
HotDir	1.167%	0.030%	2.010%	26.110%	0.487%	45.424%
HotName	0.510%	0.013%	0.880%	26.079%	0.487%	45.410%
HotDir+HotName	2.857%	0.029%	4.956%	26.083%	0.486%	45.390%
02/28/2003						
Default	1.271%	0.003%	2.099%	26.906%	0.512%	44.403%
HotDir	0.778%	0.033%	1.264%	26.900%	0.512%	44.390%
HotName	0.416%	0.009%	0.682%	26.926%	0.511%	44.410%
HotDir+HotName	2.235%	0.034%	3.673%	26.891%	0.511%	44.372%

Table 7.2: The effect, on DEAS03, of using the HotDir and HotName predictions on the number of operations that fall within the designated 4 MB region.

EECS03						
	Target			Optimal		
Condition	Total	Read	Write	Total	Read	Write
02/25/2003						
Default	0.059%	0.002%	0.094%	18.705%	0.851%	30.141%
HotDir	8.112%	0.164%	13.029%	18.701%	0.847%	30.108%
HotName	1.922%	0.010%	3.106%	18.724%	0.851%	30.164%
HotDir+HotName	8.361%	0.162%	13.433%	18.685%	0.848%	30.084%
02/26/2003						
Default	0.202%	0.001%	0.330%	21.706%	1.182%	35.363%
HotDir	5.566%	0.068%	9.082%	21.801%	1.176%	35.497%
HotName	2.653%	0.007%	4.349%	21.746%	1.177%	35.448%
HotDir+HotName	6.098%	0.062%	9.962%	21.741%	1.171%	35.410%
02/27/2003						
Default	0.135%	0.021%	0.222%	28.816%	11.805%	35.430%
HotDir	8.622%	0.127%	14.293%	21.281%	0.784%	35.328%
HotName	2.280%	0.009%	3.802%	21.359%	0.778%	35.515%
HotDir+HotName	8.296%	0.117%	13.744%	21.252%	0.776%	35.248%
02/28/2003						
Default	0.057%	0.003%	0.082%	21.310%	1.054%	30.876%
HotDir	8.204%	0.140%	11.875%	21.303%	1.052%	30.868%
HotName	2.369%	0.006%	3.445%	21.260%	1.057%	30.819%
HotDir+HotName	9.152%	0.137%	13.251%	21.246%	1.056%	30.771%

Table 7.3: The effect, on EECS03, of using the HotDir and HotName predictions on the number of operations that fall within the designated 4 MB region.

Host	Date	HotName	HotDir	HotDir/ HotName	NVRAM
CAMPUS	10/23/01	2.10%	6.44%	7.71%	10.99%
	10/24/01	-0.12%	4.43%	5.70%	10.96%
	10/25/01	-1.06%	3.97%	4.79%	9.82%
	10/26/01	-1.61%	3.73%	5.00%	10.62%
DEAS03	02/25/03	1.28%	3.11%	1.47%	4.16%
	02/26/03	0.31%	3.54%	3.92%	7.03%
	02/27/03	1.00%	2.69%	2.59%	5.40%
	02/28/03	-1.14%	0.71%	0.65%	3.10%
EECS03	02/25/03	1.44%	3.37%	3.20%	10.16%
	02/26/03	0.95%	5.82%	6.15%	11.48%
	02/27/03	1.61%	-1.00%	-1.63%	5.28%
	02/28/03	0.91%	4.03%	3.40%	10.45%

Table 7.4: Percentage improvement (compared to the default layout policy) in elapsed time for each benchmark.

The NVRAM column shows the improvement in elapsed time when HotDir and HotName are used together and the designated 4 MB region of the disk is implemented by 4 MB of RAM.

---

variation (less than 1%, and often less than 0.1%) but I believe that this can be explained by nondeterministic effects such as the timing of the synch daemon and similar effects.

One interesting observation is the difference between the read/write ratios observed at the disk and the read/write ratios observed at the NFS level. As reported in Table 4.2, the read/write ratio at the NFS layer is 2.43 for CAMPUS, 4.47 for DEAS03, and 2.67 for EECS03 during peak hours (which are the hours used by the trace replay experiments). The read/write ratio at the disk level, however, is roughly 2.2 for CAMPUS and 0.6 for DEAS03 and EECS03. It appears that the cache is absorbing most of the reads for DEAS03 and EECS03, but is not as effective at lowering the read/write ratio on CAMPUS.[1]

The heuristics do not have as much of an impact on reading, but this is not surprising because their intent is to optimize the writing of metadata. The optimal columns show that the locality of reference for reading is simply not as high as for writing – there may be some blocks that are read frequently, but they are handled by the cache and do not cause disk activity.

7.3 Timing Results

Table 7.4 shows the average percent change in elapsed time necessary to run the benchmark for each of HotName, HotDir, and HotDir with HotName, relative to the default file system. Interestingly, each workload benefits differently from different combinations of HotDir and HotName:

- CAMPUS does best when both HotDir and HotName are used, but does show improvement when HotDir is used by itself.

[1] It is dangerous to generalize these results and make inferences about the behavior of the original systems, because the Network Appliance Filers that implement EECS03 and DEAS03 use a completely different file system than FreeBSD and reportedly use a different caching strategy.

- DEAS03 usually does best when HotDir is used alone. Using HotName alone sometimes hurts performance.
- EECS03 is more variable: on most days HotDir appears to help, but on other days it reduces performance. HotName appears to always have a benefit.

7.4 Discussion

The differences in elapsed run times shown in Table 7.4 must be interpreted with some care. Unlike the original workloads, which exhibited some degree of concurrency (i.e. the server might have a queue of several outstanding requests at any moment) my benchmark performs each operation serially, issuing the next operation only after the current operation has finished. This reduces the opportunities that the disk scheduler has to reorder and optimize the request stream. This may have a substantial effect on the overall throughput.

Similarly, there no "think time" built into the benchmark – the operations are executed as quickly as possible (which is very useful because it allows me to replay the traces in faster than real time). Lack of idle time may also have an effect on the overall behavior.

Chapter 8

Related Work

8.1 Name-Based Heuristics

Todd Anderson et al describe a hint-based mechanism for controlling the persistence guarantees of files and mention that legacy applications can use file names as hints [4]. In addition to adding hints to their applications, the user may give the storage system a table of file name classes and the desired degree of persistence for each class. For example, the user might specify that `.o` files, which are created directly from other files (for example, `.c` and `.h` files) via a relatively inexpensive operation, require less persistence than the files used to create them. Files that require little or no persistence can be cached in memory and lazily written to disk at a convenient time (perhaps never) instead of written immediately.

Although they share some characteristics, Anderson's work differs from mine in several important ways: first, it is the responsibility of the application to provide the name-based hints – the storage system does not generate these hints automatically, nor does it provide any guidance to the application-level hint author. The reason for this is the second difference, which is that in Anderson's system the hints may be used to change the semantics of the underlying storage system in order to increase performance and depart from the standard semantics requires permission from the application. The goal of my work is to increase performance in a manner that preserves the standard semantics and is otherwise completely transparent to the application.

Names have also been used to guide high-level allocation policies for specialized domains such as web servers. For example, the system architect might choose to use several storage servers with different characteristics to implement the storage requirements of a large web site. One strategy is to put all of the small files (such as `.html` files) on a server optimized for small files, and distribute the larger files (such as `.jpg` files) across several other servers optimized for such files.[1] Again, this strategy is hard-coded by the application architect – this storage system has no special intelligence.

8.2 Gathering Traces

Matthew Blaze introduced several techniques for inferring the client workload from an NFS trace and implemented these techniques in rpcspy and nfstrace [9]. Andrew Moore gives a survey of these and other techniques and contrasts the analysis of file system workloads via passive network monitoring and direct kernel-level traces [50]. This work shows that passive NFS tracing reveals useful and accurate information about many aspects of an NFS workload.

[1] This example strategy has been used by the web site of the New York Times. This was brought to my attention by my colleague Jonathan Ledlie, who used to work for the New York Times.

There already exist many tools for gathering NFS traces or summary information about NFS activity. Basic tools like nfsstat(1M), iostat(1M), and nfswatch [19] provide aggregate statistics about NFS traffic. For many diagnostic purposes, these statistics suffice. Specialized tools such as nfstrace and the NFS Logging System [83] focus entirely on NFS. General-purpose network protocol monitors, such as tcpdump [89], ethereal [17], rpcspy [9], and snoop [84] can be used to analyze NFS traffic and other system activity. Most of these tools contain code to decode and print detailed information about NFS calls and responses, and therefore general-purpose tools like `tcpdump` have replaced special-purpose NFS monitors in many contexts.

In terms of general capability, my system most closely resembles nfswatch and the NFS logging/nfslogd system provided with Solaris. In comparison to nfswatch, `nfsdump` and `nfsscan` are more flexible because they runs on more platforms and capture more information – `nfsscan` can gather any combination of per-user, per-group, per-client, per-file, and per-directory information, while nfswatch gathers either per-user or per-client information.

In contrast to the NFS logging system and nfslogd, my system is completely passive, can run on a separate host instead of running on the server, and is portable across a range of platforms instead of being tied to Solaris. A shortcoming of `nfsdump`/`nfsscan` is that it does not attempt to reconstruct an application-level trace of the NFS activity in the same manner as nfslogd, but this capability could be added as a postprocess to the output of `nfsdump` (much as nfslogd postprocesses the records created by the NFS Logging System).

8.3 Benchmarks

As trace-based studies identify changing trends in workloads and uncover new problems for researchers to solve, it is benchmarks that are used to measure our progress. Benchmarks can be divided into two general categories: micro-benchmarks, which measure a particular aspect of file system performance (such as the time required to create a new file, scan a directory for a nonexistent file, or read one disk block from a file) and macro-benchmarks, which measure the performance of the file system at the level of an application or possibly an entire workload. Micro-benchmarks are well-suited to investigating the effects of changes to the file system, because they allow isolation of these effects to the syscall layer (or below) and therefore are useful to the research community. Macro-benchmarks, on the other hand, provide a more realistic view of what the users of the system running a particular workload will actually experience. Hybrid approaches, such as hBench [98], attempt to combine the best features of micro- and macro-benchmarks by characterizing a workload as an aggregate of simple operations, each of which can be represented by a micro-benchmark. This approach appears to work well, as long as there is a way to characterize the workload in terms of these micro-benchmarks.

In our work, I am primarily interested in workload- or macro-benchmarks, so I will not dwell on the myriad micro-benchmarks that have been developed. There are several file system macro-benchmarks in widespread use: the venerable Andrew Benchmark [35] has been a staple of file system research papers for nearly twenty years, but seems to be losing its popularity. This is probably for the best, because the development workload it represents is one that I believe is no longer very relevant – most users are unconcerned with how long it requires to compile programs. Previous analyses of our NFS traces imply that most users are primarily concerned with email [24].

The PostMark benchmark [37] attempts to simulate email, but actually only simulates a particular kind of mail workload – one with many small files. In the traces I examine, the component of the workload due to email is quite different. The methodology proposed by Elprin et al. [27] to generate email workloads by sending scripts of commands through a real IMAP server produces a more realistic workload, and can be made site-specific by creating new scripts and using a production IMAP server.

The SPEC SFS (System File Server) benchmark is an attempt to create a general-purpose benchmark for NFS servers [80]. SFS attempts to recreate a typical workload, based on a survey of traces. Unfortunately, the result does not resemble any NFS workload that we have observed. Furthermore, I question whether a typical workload actually exists – each NFS trace I have examined has had unique characteristics.

File system benchmarks have evolved in parallel with advances in file system design and implementation, but they usually lag file system innovation by many years. There are two primary reasons for this. First, designing and

constructing useful and accurate general-purpose benchmarks (and convincing the research community to use them) is a difficult task. The second reason follows directly from the scientific process: in order to compare two systems, it is best to run the same experiment on each. Since members of the research community cannot run today's benchmarks on the computing hardware from a decade ago, we instead run benchmarks that are a decade old on our latest hardware.

One of the goals of my research is to reduce these obstacles: first, our system constructs an accurate and repeatable benchmark directly from a trace without any user intervention. Second, the resulting benchmarks appear to be scalable: as our systems improve to the point where our current workloads are trivial, we can scale our benchmarks to the point where they will stress any system.[2]

8.4 Traces

Trace-based analyses have guided and motivated contemporary file system design for the past two decades. The original analysis of the 4.2BSD file system [58] motivated many of the design decisions of the log-structured file system (LFS) [65]. The revisitation of the original BSD study [7] confirmed the community's earlier results and further drove file system design and evaluation towards the support of distributed file systems as well as computer science engineering and research workloads. In the late 1990's, our repertoire of trace-based studies was expanded to include the increasingly dominant desktop systems of Microsoft [64, 90] and new workloads such as WWW servers [64]. It is clear from the literature of trace-based studies that there are many interesting and important workloads to consider when designing a file system, and that new workloads emerge as new applications and uses for file systems appear. I believe that as the community of computer users has expanded and evolved there has been a fundamental change in the workloads seen by file servers, and that the research community must find ways to observe and measure these new workloads.

As mentioned in Chapter 3.4, passive NFS tracing is a well-known technique and has been used as the basis of many studies and experiments [9, 10, 20, 33, 50]

The effect of client-side caching and methods to infer actual client workload from the server traffic have been studied extensively [9, 10, 20], but still present new challenges in heterogeneous networks because NFS allows considerable flexibility in how caching is implemented and in the degree of consistency provided to different observers of the file system.

8.5 File System Optimization

As CPU and memory bandwidth continue to grow at a much faster rate than disk access speed, disk I/O has become an increasingly important concern to system designers. Although there are constant improvements in disk latency and other aspects of I/O hardware architectures, it is increasingly the case that good I/O performance depends on accurate predictions of future operations that allow the system to anticipate and prepare for requests before they occur. This is particularly true for read requests, where there has been extensive work in prefetching (sometimes called pre-caching), but it is also true that file systems can benefit from knowledge about the size and future access patterns of a file when writing.

A long history of benchmarking has shown the effectiveness of the heuristics made popular by the Fast File System (FFS) [43], including aggressive read-ahead and clustering [72]. FFS also employs the heuristic of grouping files belonging to the same directory in same area on disk, in the expectation that files in the same directory are likely to be accessed together. This idea, along with the concept of *immediate files* (files that are short enough that it makes sense to embed them in the same disk page as their metadata [53]) has been extended to create variants of FFS

[2] Ningning Zhu, at the State University of New York at Stony Brook, is working on ways to extend the simple type of benchmark I use in my work into a more general benchmarking technique. One of her contributions is the ability to generate arbitrarily heavy trace-based workloads, by compressing time and/or cloning and duplicating client workloads to mimic the workload that would be expected if the number of clients increased.

such as C-FFS [29], which co-locates inode and directory information to improve directory lookups and uses explicit grouping to allow groups of small files in the same directory to be accessed entirely sequentially. These heuristics, while effective in many cases, are limited because they are static and do not adapt to variations in workload that violate the assumptions of their designers.

To overcome this limitation, there has been research in dynamically learning the inter-file access patterns among groups of files, both for general-purpose file systems [31, 40] and for highly tuned special-purpose file systems for applications like web servers (such as the Hummingbird file system [76]).

There has also been considerable work in developing and exploiting predictive models at the disk-block level, instead of the file system level. These can be done at the hardware level (i.e., AutoRAID [93]) or under the control of the operating system [1] or via a hybrid of block and file level optimization [81].

Every widespread heuristic approach suffers from at least one of the following problems: first, if the heuristics are wrong, they may cause performance to degrade, and second, if the heuristics are dynamic, they may take considerable time, computation, and storage space to adapt to the current workload (and if the workload varies over time, the adaptation might never converge).

One partial solution to the problem of inappropriate or incomplete file system heuristics is for the application using the files to supply *hints* to the file system about their anticipated access patterns. These hints can be extremely successful, especially when combined with the techniques of prefetching and selective caching [16, 60]. The drawback with this approach is that it requires that applications be modified to provide hints. There has been work in having the application compiler automatically generate hints, but success in this area has been largely confined to scientific workloads with highly regular access patterns [52].

Chapter 9

Conclusions

9.1 Lessons Learned

In this section I reflect on some of the things I learned during the course of performing the research described in this thesis.

- *Look at the data.*

  Rich data sets are like uncharted worlds. Don't just race through them in order to complete some errand; take some time and explore.

  The research questions that motivated me to build `nfsdump` were quite a bit different than the research questions that I eventually decided to pursue and are addressed in this thesis. I didn't know ahead of time that NFS calls were being re-ordered on the client, but this discovery (for which Jonathan Ledlie deserves most of the credit) led to the invention of *SlowDown* and *Cursor*. Benchmarking *SlowDown* turned out to be much more complicated than I had anticipated – when I looked at the benchmark results, there were things I didn't understand. Rather than simply using the benchmark results, I decided to investigate more carefully. The results of my exploration form the majority of Chapter 5.

  Similarly, the fact that file names are strongly associated with file properties was something that I noticed in data that I had gathered for other purposes. The hypothesis that this association exists did not simply pop into my head one day – it was in the data all along, waiting for me to find it.

- *Never throw away data.*

  An analyst's nightmare is to be cornered by a question that can only be answered with a variable that is not in the data set – so when in doubt, save every variable possible. This is doubly true for data sets such as the NFS traces, which were very hard to get and might never be repeated.

  Many of the analyses I eventually performed on the trace data would have been impossible if I had written `nfsdump` in such a way that it did the minimum required for the experiments I had planned in my original research proposal. (For example, my original plan did not include looking at names in any way, so I could easily have discarded them.) Retaining as much information as possible turned out to be quite useful – both to me and to other researchers who have used my traces in a variety of different ways.

  I must admit that `nfsdump` does not record *everything* from the trace; there were things that were difficult or awkward to capture and for which I could think of no use, and so I made the decision to omit them. Now, of course, other researchers are asking for them, so perhaps the next version of `nfsdump` will support them as well (if I can find a way to record this information without breaking backward compatibility).

- *Organize the data for future reference.*

I knew that I was going to look at the output of `nfsdump` in a lot of different ways, so I tried to make it easy to handle. The output if `nfsdump` is plaintext, easy for a human to read, and easy to parse. My goal was to make it possible to write programs that do simple analyses with just a few lines of Perl or awk. In this regard, the format was effective, but there were trade-offs.

The output format of `nfsdump` is large and awkward to process in complex ways. Furthermore, the goal of being able to do analysis in a scripting language like Perl may not have been justified – it's easy to write and maintain short programs in Perl, but long programs are notoriously problematic. Perl is not a particularly helpful language for building complicated systems; it has little in the way of types (and is philosophically opposed to notions such as type safety). I spent a lot more time debugging my code than I planned, and this may have overwhelmed the benefit of using a high-level language in the first place.

A second problem is that Perl is slow. I didn't anticipate how large the data sets would become, and therefore I didn't anticipate that processing speed would become an important issue. For many of my experiments, however, processing time has become a dominant cost; I would like to have tried other analyses, or re-run my analyses on longer sets of data, but it takes too much time and RAM.

My colleagues from CMU, Michael Mesnier and Eno Thereska (and their advisor Gregory Ganger) translated the traces into a binary format and wrote their tools in C/C++. They are able to perform analyses in hours that might take days with my tools.

9.2 Future Directions

9.2.1 Identifying Problems in the File System Interface

As I mentioned in the Introduction, the guiding philosophy of the work I have presented here is that the server should be intelligent enough to deal with strange or sub-optimal client behavior. My goal is to improve performance by changing the server in ways to which the client is oblivious. This is a reasonable research goal. It is, however, not the only (and arguably not the most effective) way to increase storage server performance. Another method is to widen the interface in order to allow the applications running on the clients to pass extra information to the server.

On the surface, this idea appears to be completely contrary to the idea of improving storage performance *without* changing the storage server interface, but the two ideas can share a great deal in terms of methodology (if not philosophy). The first task in widening the interface is to discover the narrow passages in the existing interface – the places where adding extra information might be worth the corresponding cost of complicating the interface. One way to find these points is to consider the problem from the perspective of an intelligent server: what information is beneficial to the server, and what information can the server infer without any help from the application?

As an example, in Chapter 5 I showed that the Cursor heuristic can detect some kinds of read access patterns and perform appropriate read-ahead without any help from the client. Because there is little or no benefit for the application to tell the server that it is going to use this kind of access pattern there is no need to provide an interface with which to do so.

In some cases, the changes to the interface that would be necessary to support an application are so sweeping that the result would hardly resemble a file system. In the traces there are frequent examples of abuse or misuse of the file system or NFS interface in order to perform operations for which the file system interface is ill-suited. Some examples of this include:

- Use of the file system as a lock manager.
- Use of flat-files or directories to implement associative arrays. Mail spools can be treated as a special case of associative arrays – each mail message is a single entry, accessed independently, within an array of entries.
- Using files as a mechanism for interprocess communication. This is particularly problematic for NFS, for two reasons:

- NFS uses a weak cache consistency model that requires clients to poll the server in order to find out whether a file has changed. Since polling is a cause of overhead, some clients elect to poll the server infrequently (especially for metadata). This permits inconsistencies to exist between the clients – two applications might view the same directory and observe different contents.

  To avoid the potential of inconsistency, locking is required. If these locks are implemented via the file system, then this brings us back to an earlier example.
- NFS performs file-level caching instead of block-level caching. This means that a change of any part of a file (or even a write that overwrites a block of the file with the same value) causes the entire file to be invalidated from the client. Re-reading the file again wastes server and client time as well as network capacity.

Seen in this light, the entire workload of CAMPUS is arguably an abuse of NFS.

It is an open question whether it is better to make the storage system interface so flexible and amorphous that it can support both the ordinary file system semantics and the kinds of operations that would support the CAMPUS workload or whether it is better to treat these cases separately. There is already a body of work that shows that mail workloads can be implemented more efficiently outside of the file system [27]. It is my belief that the storage system interface should be kept as narrow as possible in order to make them easy to define, implement, and test, and should only be changed if the benefit is great.

9.2.2 Improved Predictions

All of the name- and attribute-based predictive models discussed in this thesis are based on techniques that can be applied to anonymized data. As discussed in Section 3.6.1, the anonymization step by necessity removes information about file names. Enough information remains to permit reasonably accurate models, but it might be possible to build considerably more accurate models if the complete information was available.

In the envisioned application of these models, the models are constructed and used inside a storage server and therefore there are no extra concerns about privacy or anonymity because the information used to create these models is already present in the system. Therefore future systems might be able to use more sophisticated methods to extract information from the true file names and therefore build and use more accurate models.

Determining how much information is actually expressed in file names would be an interesting topic (in addition to discovering whether better predictions would also provide better performance).

9.3 End Note

One of the disappointments of my research to this point is that I have not found a simple example, based on real workloads, to prove that name-based adaptations can improve performance to a substantial degree. However, I do not think that this is terribly surprising – it is hard to find proof that any new general optimization technique will improve the performance of real workloads. The problem is that application coders already make an effort to write the applications in a manner that works well with the underlying storage systems, and storage system designers have gone to great effort to build storage systems that give good performance for general workloads. The application and storage system designers have therefore been spiraling in on a relatively small number of points in the design space and trying to build systems that work well at those points.

The real opportunity for adaptive systems to prove themselves will come with the advent of new applications. Just as the exploding popularity of the world wide web prompted many new research initiatives to build systems that can handle the enormous workload and scaling issues required to build global-scale systems, I believe that the next new application that requires large-scale network storage will benefit from this adaptation.

The current generation of optimized WWW services was largely constructed by hand, using unproven and

untested ideas and depending to a large extent on the ingenuity of human engineers.[1] When we can build adaptive systems, however, the next generation of network services will be optimized by the servers themselves. There is much work left to do.

[1] Or, in many cases, the ability to throw money and hardware at the problem.

Bibliography

[1] Sedat Akyurek and Kenneth Salem, "Adaptive block rearrangement," *Computer Systems*, volume 13, number 2, pages 89–121, 1995.

[2] Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez, and John Wilkes, "Selecting RAID levels for disk arrays," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 189–201, January 2002.

[3] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Yang, "Serverless Network File Systems," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 109–126, December 1995.

[4] Todd A. Anderson and James Griffioen, "An application-aware data storage model," in *Proceedings of the USENIX Annual Technical Conference*, pages 309–322, June 1999.

[5] Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, "Information and control in Gray-Box systems," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, vol. 35, 5 of *ACM SIGOPS Operating Systems Review*, (New York), pages 43–56, ACM Press, Oct. 21–24 2001.

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici, "Transforming Policies into Mechanisms with Infokernel," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 90–105, October 2003.

[7] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, "Measurements of a Distributed File System," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[8] Sorav Bansal and Dharmendra S. Modha, "CAR: Clock with Adaptive Replacement," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, March 2004.

[9] Matthew A. Blaze, "NFS Tracing by Passive Network Monitoring," in *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, January 1992.

[10] Matthew A. Blaze, *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[11] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone, *Classification and Regression Trees*. Chapman and Hall, 1984.

[12] Angela Demke Brown and Todd C. Mowry, "Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently," in *Proceedings of the USENIX Symposium on Operating Design and Implementation OSDI*, pages 31–44, October 2000.

[13] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz, "Disk Scheduling with Quality of Service Guarantees," in *IEEE International Conference on Multimedia Computing and Systems*, vol. 2, pages 400–405, June 1999.

[14] Brent Callaghan, Brian Pawlowski, and Peter Staubach, "NFS Version 3 Protocol Specification." `http://www.ietf.org/rfc/rfc1813.txt`, June 1995.

[15] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[16] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Transactions on Computer Systems*, volume 14, number 4, pages 311–343, 1996.

[17] Gerald Combs, "ethereal." `http://www.ethereal.com/`.

[18] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter, "Connection Scheduling in Web Servers," Technical Report 1999-003, Boston University, 1999.

[19] Dave Curry and Jeff Mogul, "nfswatch." `http://freeware.sgi.com/cd-3/relnotes/nfswatch.html`.

[20] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, 1994.

[21] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh, "The logical disk: a new approach to improving file systems," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, 1994.

[22] Harvey M. Deitel, *Operating Systems, 2nd Edition*. Addison-Wesley Publishing Company, 1990.

[23] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "Bridging the Information Gap in Storage Protocol Stacks," in *Proceedings of the USENIX Annual Technical Conference*, pages 171–190, June 2002.

[24] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer, "Passive NFS Tracing of Email and Research Workloads," in *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216, March 2003.

[25] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer, "The Utility of File Names," Technical Report TR-05-03, Harvard University Division of Engineering and Applied Sciences, 2003.

[26] Daniel Ellard, Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Margo Seltzer, "Attribute-Based Prediction of File Properties," Technical Report TR-14-03, Harvard Computer Science Technical Report, December 2003.

[27] Nicholas Elprin and Bryan Parno, "An Analysis of Database-Driven Mail Servers," in *Proceedings of the 17th Large Installation Systems Administration Conference (LISA)*, pages 15–22, October 2003.

[28] Eran Gabber and Elizabeth Shriver, "Let's put NetApp and CacheFlow out of business!," in *Proceedings of the ACM SIGOPS European Workshop*, pages 85–90, September 2000.

[29] Gregory R. Ganger and M. Frans Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," in *Proceedings of the USENIX Annual Technical Conference*, pages 1–17, 1997.

[30] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt, "Soft Updates: a Solution to the Metadata Update Problem in File Systems," *ACM Transactions on Computer Systems*, volume 18, number 2, pages 127–153, 2000.

[31] J. Griffioen and R. Appleton, "Improving file system performance via predictive caching," *Parallel and Distributed Computing Systems*, pages 165–170, September 1995.

[32] Dick Grune, Brian Berliner, and Jeff Polk, "Concurrent Versions System." `http://www.cvshome.org/`.

[33] Riccardo Gusella, "The Analysis of Diskless Workstation Traffic on an Ethernet," Technical Report 379, University of California at Berkeley, 1987.

[34] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," in *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, 17–21 1994.

[35] J. Howard, M. Kazar, S. Menees, S. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed System," *ACM Transactions on Computer Systems*, volume 6, pages 51–81, February 1988.

[36] Van Jacobson, Craig Leres, and Steven McCanne, "libpcap." `http://sourceforge.net/projects/libpcap/`.

[37] J. Katcher, "PostMark: A New File System Benchmark," Technical Report TR-3022, Network Appliance, October 1997.

[38] Steve R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," in *Proceedings of the USENIX Summer Technical Conference*, pages 238–247, 1986.

[39] Andrew J. Klosterman and Gregory R. Ganger, "Cuckoo: Layered clustering for NFS," Technical Report CMU-CS-02-183, Carnegie Mellon University School of Computer Science, 2002.

[40] Thomas M. Kroeger and Darrell Long, "The Case for Efficient File Access Pattern Modelling," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 14–19, IEEE, March 1999.

[41] Butler W. Lampson, "Hints for Computer System Design," *ACM Operating Systems Review*, volume 15(5), pages 33–48, October 1983.

[42] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou, "C-Miner: Mining Block Correlations in Storage Systems," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 173–186, April 2004.

[43] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, "A Fast File System for UNIX," *Computer Systems*, volume 2, number 3, pages 181–197, 1984.

[44] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[45] Nimrod Megiddo and Dharmendra S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, March 2003.

[46] Michael Mesnier, Eno Thereska, Daniel Ellard, Gregory R. Ganger, and Margo Seltzer, "File classification in self-* storage systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 44–51, May 2004.

[47] Michael Mesnier, Eno Thereska, and Gregory R. Ganger, "ABLE: An Attribute-Based Learning Environment." `http://www.pdl.cmu.edu/ABLE/index.html`.

[48] Rodney Van Meter, "Observing the Effects of Multi-Zone Disks," in *Proceedings of the Usenix Technical Conference*, pages 19–30, January 1997.

[49] Sun Microsystems, "NFS: Network File System Protocol Specification." `http://www.ietf.org/rfc/rfc1057.txt`, June 1988.

[50] Andrew W. Moore, "Operating System and File System Monitoring: a Comparison of Passive Network Monitoring with Full Kernel Instrumentation Techniques," Master's thesis, Monash University, 1995.

[51] Richard Moore, "A Universal Dynamic Trace for Linux and other Operating Systems," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 297–308, June 2001.

[52] Todd C. Mowry, Angela K. Demke, and Orran Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 3–17, 1996.

[53] S. Mullender and A. Tanenbaum, "Immediate files," *Software – Practice and Experience*, volume 4, pages 365–368, April 1984.

[54] D. Muntz and P. Honeyman, "Multi-level Caching in Distributed File Systems -or- Your cache ain't nuthin' but trash," in *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.

[55] "The NFSv4 Project." `http://www.nfsv4.org/`.

[56] Bill Nowicki, "NFS: Network File System Protocol Specification." `http://www.ietf.org/rfc/rfc1094.txt`, March 1989.

[57] James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "Controlling your PLACE in the File System with Gray-Box Techniques," in *Proceedings of the USENIX Annual Technical Conference*, pages 311–324, June 2003.

[58] 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," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–24, Dec. 1985.

[59] David A. Patterson, Garth Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 109–116, June 1988.

[60] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka, "Informed prefetching and caching," in *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pages 79–95, December 1995.

[61] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger, "Storage-based Intrusion Detection: Watching storage activity for suspicious behavior," in *Proceedings of the USENIX Security Symposium*, pages 137–152, August 2003.

[62] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "Robust, Portable I/O Scheduling with the Disk Mimic," in *Proceedings of the USENIX Annual Technical Conference*, pages 297–310, June 2003.

[63] Peter Radkov, Li Yin, Pawan Goyal, Prasenjit Sarkar, and Prashant Shenoy, "A Performance Comparison of NFS and iSCSI for IP-Networked Storage," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 101–114, March 2004.

[64] Drew Roselli, Jacob Lorch, and Thomas Anderson, "A Comparison of File System Workloads," in *Proceedings of the USENIX Annual Technical Conference*, pages 41–54, 2000.

[65] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, volume 10, number 1, pages 26–52, 1992.

[66] Chris Ruemmler and John Wilkes, "UNIX disk access patterns," in *Proceedings of the USENIX Annual Technical Conference*, pages 405–420, Winter 1993.

[67] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir, "Deciding when to forget in the Elephant file system," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–123, 1999.

[68] P. Sarkar and K. Voruganti, "IP Storage: The Challenge Ahead," in *Proceedings of the IEEE Symposium on Mass Storage Systems*, pages 35–42, April 2002.

[69] Prasenjit Sarkar and John H. Hartman, "Hint-based cooperative caching," *ACM Transactions on Computer Systems*, volume 18, number 4, pages 387–419, 2000.

[70] Steven W. Schlosser and Gregory R. Ganger, "MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 87–100, March 2004.

[71] Margo Seltzer, Peter Chen, and John Ousterhout, "Disk Scheduling Revisited," in *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, 1990.

[72] Margo Seltzer and Keith Smith, "File system logging versus clustering: A performance comparison," in *Proceedings of the USENIX Annual Technical Conference*, pages 249–264, June 1995.

[73] Margo I. Seltzer, *File System Performance and Transaction Support*. PhD thesis, University of California, Berkeley, College of Engineering, Electrical Engineering and Computer Science Department, Computer Science Division, January 1992.

[74] Spencer Shepler, Carl Beame, Brent Callaghan, Mike Eisler, David Noveck, David Robinson, and Robert Thurlow, "The NFS Version 4 Protocol." `http://www.ietf.org/rfc/rcf3530.txt`, May 2000.

[75] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck, "The NFS Version 4 Protocol." `http://www.ietf.org/rfc/rfc3010.txt`, May 2000.

[76] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein, "Storage management for web proxies," in *Proceedings of USENIX Annual Technical Conference*, pages 203–216, June 2001.

[77] Elizabeth Shriver, Christopher Small, and Keith A. Smith, "Why Does File System Prefetching Work?," in *Proceedings of the USENIX Annual Technical Conference*, pages 71–84, June 1999.

[78] Keith A. Smith and Margo I. Seltzer, "File System Aging - Increasing the Relevance of File System Benchmarks," in *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, June 1997.

[79] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger, "Metadata Efficiency in Versioning File Systems," in *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 43–58, March 2003.

[80] "SPEC SFS (System File Server) Benchmark." `http://www.spec.org/osg/sfs97r1/`, 1997.

[81] Carl Hudson Staelin, "High performance file system design," Technical Report TR-347-91, Princeton University, 1991.

[82] Hal Stern, Mike Eisler, and Ricardo Labiaga, *Managing NFS and NIS, 2nd Edition*. O'Reilly & Associates, 2001.

[83] Sun Microsystems, Inc, "nfslogd(1M)." Solaris 8 Reference Manual Collection.

[84] Sun Microsystems, Inc, "snoop(1)." Solaris 8 Reference Manual Collection.

[85] Andrew S. Tanenbaum, *Modern Operating Systems, 2nd Edition*. Prentice Hall, 2001.

[86] Renu Tewari, Richard King, Dilip Kandlur, and Daniel M. Dias, "Placement of Multimedia Blocks on Zoned Disks," in *Multimedia Computing and Networking*, January 1996.

[87] Peter Triantafillou, Stavros Christodoulakis, and Costas Georgiadis, "A Comprehensive Analytical Performance Model for Disk Devices Under Random Workloads," *Knowledge and Data Engineering*, volume 14, number 1, pages 140–155, 2002.

[88] Mustafa Uysal, Arif Merchant, and Guillermo A. Alvarez, "Using MEMS-based storage in disk arrays," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, March 2003.

[89] Van Jacobson and Craig Leres and Steven McCanne, "tcpdump implementation." `http://www.tcpdump.org/`.

[90] Werner Vogels, "File System Usage in Windows NT," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 93–109, Dec. 1999.

[91] Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson, "Virtual log based file systems for a programmable disk," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–43, 1999.

[92] Wenguang Wang, Yanping Zhao, and Rick Bunt, "HyLog: A High Performance Approach to Managing Disk Layout," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 144–158, March 2004.

[93] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan, "The HP AutoRAID Hierarchical Storage System," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 90–106, IEEE Computer Society Press and Wiley, 2001.

[94] Thomas Williams and Colin Kelley, "gnuplot." `http://www.gnuplot.info`.

[95] Theodore M. Wong and John Wilkes, "My cache or yours? Making storage more exclusive," in *Proceedings of the USENIX Annual Technical Conference*, pages 161–175, June 2002.

[96] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes, "On-line extraction of SCSI disk drive parameters," Technical Report CSE-TR-323-96, Carnegie Mellon University School of Computer Science, December 1996.

[97] Karim Yaghmour and Michel R. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging," in *Proceedings of the USENIX Annual Technical Conference*, pages 13–26, June 2000.

[98] Xiaolan Zhang, *Application-Specific Benchmarking*. PhD thesis, Harvard University, May 2001.

[99] Ningning Zhu, Jiawu Chen, Tzi-cker Chiueh, and Daniel Ellard, "An NFS Trace Player for File System Evaluation," Technical Report TR-16-03, Harvard University DEAS, 2003.