Serverless Network File Systems

by

Michael Donald Dahlin

B.S. (Rice University) 1991
M.S. (University of California at Berkeley) 1993

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David A. Patterson, Chair
Professor Thomas E. Anderson
Professor Pamela Matson

1995

The dissertation of Michael Donald Dahlin is approved:

Chair	Date
	Date
	Date

University of California at Berkeley

1995

Serverless Network File Systems

Abstract

Serverless Network File Systems

by

Michael Donald Dahlin

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David A. Patterson, Chair

This thesis presents the design of a *serverless network file system*, a file system that distributes its functionality among cooperating, networked machines to eliminate the central file server bottleneck found in current network file systems.

In a serverless system, any machine can cache, store, or control any block of data. This location independence provides better performance and scalability than traditional file system architectures. Further, because any machine in the system can assume the responsibilities of a failed component, the serverless design can also provide higher availability.

This dissertation details the design of three serverless subsystems, each of which distributes a specific piece of functionality that a traditional system would implement as part of a central server. I describe and evaluate *cooperative caching*, a way to coordinate client caches to replace the central server's cache. I evaluate different *distributed disk storage* architectures and present several improvements on previous log-structured, redundant storage systems. Finally, I present the design of a *distributed management* architecture that splits the control of the file system among managers while maintaining a seamless single system image.

Together, these pieces form a a serverless file system that eliminates central bottlenecks. I describe the integration of these components in the context of xFS, a prototype serverless file system implementation. Initial performance measurements illustrate the promise of the serverless approach for providing scalable file systems.

Professor David A. Patterson

Table of Contents

List of Figures

List of Tables

Acknowledgments

One of the things I've looked forward to most about writing my dissertation has been the chance to write this acknowledgments section. So many people have contributed to my enjoyment and success as a graduate student, but in our hectic day-to-day lives, I seldom have the chance to thank them. Without the people listed here (and others I'm sure I've forgotten) I would not be here. Thank you all for five terrific years!

I have been fortunate to work with Dave Patterson and Tom Anderson acting as my co-advisors. Although each is a terrific mentor in his own right, a happy set of circumstances has let me benefit from their complementary strengths and philosophies. Not only have they offered invaluable technical guidance, criticism, and inspiration, but they have also helped me develop as a writer, teacher, and professional engineer. Both are also well aware that everyone needs a life outside of the lab — an important quality in an advisor.

Pam Matson has been kind enough to act as the third member of my committee and as an informal advisor for my Global Climate Change minor. I have enjoyed working with her and benefitted greatly from her guidance as I tried to get a toe-hold for my understanding of global change.

My office-mates, Jeanna Neefe, Drew Roselli, and Randy Wang, have contributed to the development of many of the ideas in this thesis. We have worked together on the xFS project for several years and will continue to develop the ideas discussed in this dissertation in the future. It is great to work in a group where everyone criticizes each other's work (quite energetically at times), and everyone accepts criticism in the productive spirit in which it is offered. xFS is a difficult project, and it would be impossible without this constant, critical feedback.

The other members of NOW project also provided much advice and inspiration during my work on my thesis. I could always find someone to bounce an idea off of or run a rough paper draft

past, and it has been fun to work with a group of about 30 smart people working on a wide range of interesting projects.

Katherine Crabtree, Bob Miller, and Terry Lessard-Smith have protected me from the bureaucracy of a 30,000 student university. Their ability to make paperwork do the right thing and to help avoid bureaucratic pitfalls is nothing short of brilliant.

The dissertation has benefitted greatly from Thomas Phillipi's efforts. His careful proofreading and editorial suggestions have made this process much more bearable.

I must also thank all of my fellow students who have made graduate school such a pleasure. The Happy Hour/Alexis Monday/Bago crew has been a constant source of dearly needed distractions, and Wednesday basketball has been a blast.

My debt to my parents and family is immeasurable. They have offered me their unconditional support in all of my endeavors. Leading by example, they have instilled in me the value of hard work and a job well done, and their confidence in me has given me the confidence to succeed in graduate school.

Finally, my fiance, Marla, deserves much of the credit for this dissertation. Her love and support have been unfailing, and she has always been understanding of the evenings and weekends spent in the lab.

This research was funded by a National Science Foundation graduate fellowship, a National Science Foundation grant (CDA 0401156), a Advanced Research Projects Agency grant (N00600-93-C-2481, F30602-95-C-0014), and California MICRO. Corporate support for different aspects of the xFS and NOW projects came from the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett Packard, IBM, Siemens Corporation, Sun Microsystems, and Xerox Corporation.

1 Introduction

The evolution of computer processing has followed an apparently inexorable trend as the platform of choice for demanding applications has moved from centralized, shared "mainframe" computers to collections of commodity, "personal" computers. This evolution has been described variously as the "attack of the killer micros" (a description popularized by Eugene Brooks), "many, little defeating few, big" [Gray, 1995], and, as Figure 1-1 illustrates, an inverted food chain in which the little fish eat the big ones.

No matter what name it is given, this trend arises from the incredible economies of scale for microcomputers, which ship in volumes measured in tens of millions of units each year. These high volumes enable research and development expenditures that cause microprocessor performance to improve by 60% per year while microcomputer prices hold steady or even decline over time. These trends make microcomputers the most cost-effective computing platform for many applications.

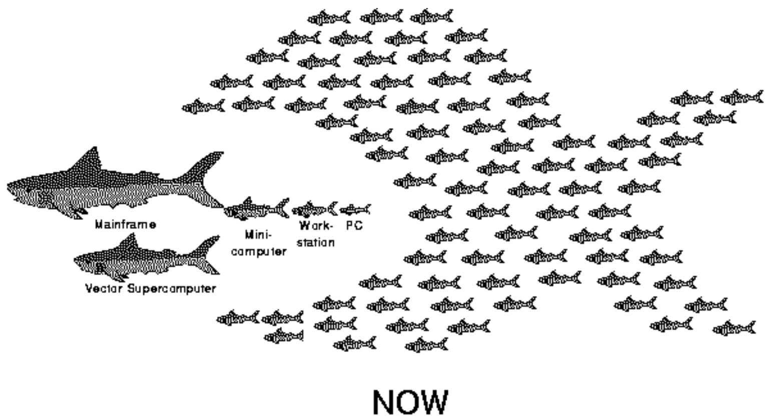


FIGURE 1-1. **"Attack of the killer micros."** This drawing illustrates the "attack of the killer micros" in the form of an inverted food chain, where little fish eat the big ones. The same forces that make killer micros attractive for compute-intensive applications make them attractive way to provide file service. This illustration was assembled by Drew Roselli based on an earlier version by Dave Patterson.

Recent advances in local area networks (LANs) expands the range of applications for which killer micros can be used. Next-generation LANs such as Autonet [Schroeder et al., 1991], ATM [ATM Forum, 1993], switched Ethernet, and Myrinet [Boden et al., 1995] provide aggregate network bandwidth that scales with the number of machines connected to the network, and new, low-latency network interfaces [von Eicken et al., 1992, Martin, 1994, Basu et al., 1995] allow machines connected by these networks to cooperate closely. These network technologies allow groups of microcomputers to work together to solve problems that are too large for a single one to handle, further extending the realm of the killer micros. For instance, massively-parallel computers such as the Thinking Machines CM-5, IBM SP-2, Cray T3D, and Intel Paragon are based on microprocessors connected by fast networks and are replacing vector supercomputers such as the Cray Y-MP for many applications.

Although compute-intensive applications have been quick to benefit from the personal computer revolution, file systems have been slower to take advantage of the "killer micros." Even when many users each run their programs on microcomputers, they typically store their file data on a single, centralized *file server*. This central server is both a performance and reliability bottleneck: because all file system requests go through the server, the single server limits the system's throughput, and a server failure halts file system activity. Current file systems attempt to address this problem by using high-end, special-purpose central server machines that resemble mainframes in their cost and complexity.

My thesis is that file systems built around a new architecture called *serverless network file systems* can exploit the performance and cost/performance advantages of microcomputers that are connected by fast LANs. A serverless network file system's architecture can be described with the phrase "anything, anywhere" because this approach distributes all aspects of file service — including data storage, data caching, and control processing — across potentially all of the machines in the system. I hypothesize that this approach can realize three primary benefits compared to a central server architecture.

• **Improved Performance**

A serverless system outperforms a centralized system by eliminating the central bottleneck and utilizing the aggregate resources of a large number of machines. For instance, a single client can read or write data in parallel to a large number of disks and thereby achieve a peak disk bandwidth limited only by the bandwidth between the client and the network. Similarly, when many clients actively access disk storage, the system's disks work together to provide aggregate disk bandwidth approaching the sum of the bandwidths of all of the disks. Because fast networks make it faster to access remote RAM than local disk, the serverless system can also provide better performance than a central server by exploiting the combined memory caches of all of the machines; this large *cooperative cache* reduces the number of disk accesses compared to the individual cache of a typical central server. Finally, the serverless system distributes control processing across many machines so that control bottlenecks do not limit data throughput.

In addition to eliminating bottlenecks, the location independence of the serverless approach improves performance by improving locality and balancing load. For instance, because any machine in the system can control any file, a serverless system can co-locate the control and metadata processing of a file with the client using that file's data, thereby reducing latency and load due to communication between the client and file's manager. Similarly, if a disk, processor, or network link is a hot spot, a serverless system can change the distribution of files to machines to balance load more evenly.

• **Improved Availability**

A serverless system provides high availability because when one machine fails, the remaining machines can take over its responsibilities. Two aspects of the serverless design are crucial to this goal. First, the distributed storage system uses redundant disk storage [Patterson et al., 1988] and log-based storage [Rosenblum and Ousterhout, 1992] to ensure that higher levels of the system can access all necessary file system state even when some machines have failed. Second, the system can use location independence to delegate the duties of failed machines to the remaining machines.

In a central server architecture, in contrast, the central server is a critical resource; when a central server fails, file system clients can access only data that they already have cached until it is repaired. Several central server systems have attempted to use server replication to increase availability [Walker et al., 1983, Kazar, 1989, Popek et al., 1990, Liskov et al., 1991, Kistler and Satyanarayanan, 1992, Birrell et al., 1993]. However, replication increases the cost and complexity of central servers, and it can also increase the latency of writes since the system must replicate data at multiple servers.

• **Improved Scalability**

A key design goal is to eliminate all centralized bottlenecks from the file system — effectively treating the file system as a parallel program. A serverless system's performance and availability improve as more CPUs, DRAM, or disks are added to provide more resources for the system to exploit. This architecture should work well for systems with tens or hundreds of machines; it was not designed to scale beyond about one thousand machines because such large systems are unlikely to provide the uniformly fast networks and mutual trust among machines assumed by the design.

In addition to these benefits, because the serverless architecture relies on commodity microcomputers and LANs, it should achieve these goals in a cost-effective manner compared to central-server approaches and allow file systems to take advantage of the "killer micro" revolution.

To support my thesis, I make four principal contributions in this dissertation.

1. I demonstrate how cooperative caching, using the cache memory already present in desktop machines as a global file cache, outperforms traditional central server caching.
2. I evaluate a range of distributed disk storage approaches and conclude that log-structured redundant network storage [Hartman and Ousterhout, 1995] is the most promising approach; I extend this design to provide scalable, distributed disk storage.
3. I describe how to distribute file system control across many machines through distributed file management and quantify its benefits compared to central server management.
4. I bring these elements together in the design of a prototype serverless network file system and demonstrate that the prototype achieves good performance and scalability.

1.1. Design Environment and Applicability

I have developed the serverless system described here in the context of the Berkeley Network of Workstations (NOW) project [Anderson et al., 1995]. This environment has four characteristics that affect the design. First, the building block for this system is a commodity workstation or personal computer (PC) that contains a high-performance processor, a significant amount of memory, and a disk. Second, these computers are connected by a high-performance, switched local area network. Third, these machines are administered as a unit, so all machines are considered to be equally secure. Finally, a NOW can be used for a wide range of workloads from traditional, office/engineering programs to batch and parallel jobs.

Although I describe the serverless design in a context where the building blocks are complete workstations that act as peers, other configurations are possible. For instance, it may be desirable to configure some machines with many more disks than a typical desktop machine to amortize the cost of the machine's processor and memory over many inexpensive disks. Also, while the serverless system can be made entirely of desktop workstation peers suppling file service to one another, another approach would be to run the serverless file system on a number of computers in a machine room and then use this serverless "core" of machines to supply file system services to "fringe" client machines on desktops; this approach could be used when desktop machines do not meet the high-performance network assumptions or security assumptions made in the serverless design.

My design also assumes that networks are fast compared to disk; Chapter 2 discusses this assumption. Emerging high-bandwidth, switched local area networks allow machines to cooperate closely to provide seamless, high-performance file service. Although my description and initial implementation assume that machines communicate via explicit network messages, the benefits of the serverless design would also apply to systems that use a shared-memory abstraction to provide high-performance communication among nodes that each behave like a workstation [Kubiatowicz and Agarwal, 1993, Blumrich et al., 1994, Kuskin et al., 1994].

To get the full benefits of the serverless design, machines must trust one another so that file service can be distributed among machines. As I've already mentioned, dividing machines up into "core" trusted machines and "fringe" less-trusted machines may allow the serverless design to be used in environments that are less homogeneous than a NOW. Chapter 6 explores security issues in detail.

The serverless design presented here should work well for a wide range of workloads. Most of the simulation results presented in this thesis use office/engineering trace workloads that include graduate students, staff, and faculty's day-to-day activities such as reading mail, editing files, and simulating computer systems. These workloads present a number of challenges to file system designers: in particular they require systems to handle not just large files, but also small files efficiently. The system was also designed with other NOW workloads in mind; while I did not have access to trace workloads for the parallel and batch workloads that may become more common in a NOW, the serverless system eliminates central bottlenecks and supports data sharing to make it work well in such an environment. Further, while I do not consider multimedia workloads explicitly, the scalable system presented here should be of interest to file system designers who wish to support those workloads.

1.2. Overview of the Dissertation

The body of this thesis consists of six chapters. The first motivates taking a new approach to file system design. The remaining chapters present the serverless design and evaluate that approach by describing how to distribute each of the three main pieces of a file server (cache, disk, and control), the security implications of this new approach, and how the different pieces of the design fit together in a prototype serverless file system.

In Chapter 2, I review key technology trends that affect file system design. I explore the changing trade-offs made possible as different technologies evolve at different rates over time and conclude that disk, processor, memory, and network technologies are improving in ways that motivate distributing file system services across a network. I also examine how workload trends motivate scalable file systems.

Chapter 3 describes cooperative caching. Cooperative caching replaces central server caching by coordinating the contents of clients' caches and allowing clients to satisfy other clients' read requests. In addition to being more scalable than central server caching, cooperative caching outperforms central server caching by reducing the percentage of requests that go to disk. For instance, I simulated both cooperative caching and traditional central server caching under an office/engineering workload and found that cooperative caching improved file system read performance by 30% to 150%.

Chapter 4 explores how to distribute disk storage across multiple machines. It begins by exploring a range of options and concludes that redundant, log-based striping similar to that used in Zebra [Hartman and Ousterhout, 1995] has the most promise. It then builds on the Zebra design to make it scale to large numbers of disks by eliminating its bottlenecks.

I then describe how to distribute the remaining file system functionality by spreading control to distributed managers. Chapter 5 presents an approach that adapts multiprocessor cache consistency designs [Lenoski et al., 1990, Chaiken et al., 1991] to the rigors of file system use. In particular, I describe how to adapt the data structures of log structured file systems (LFS) [Rosenblum and Ousterhout, 1992] for distributed management and how to allow the system to reconfigure management duties to continue operation in the presence of failures; I also explore policies for assigning files to managers to improve locality.

While the serverless design generally assumes that machines are equally secure, not all environments meet this assumption. Chapter 6 describes how the serverless design can be modified to work in environments where not all machines trust one another. Unfortunately, reducing trust makes it harder for machines to use one another's resources to provide file service, but restricted versions of the serverless protocol may still provide benefits for some machines.

Chapter 7 integrates the pieces of the design by describing xFS, a prototype serverless network file system. I first detail how cooperative caching, distributed disk storage, and distributed management work together to provide integrated file service. I then present initial performance results for xFS running on up to 32 SPARCStation 10's and 20's. These results illustrate the promise of the serverless approach, but they also point out several limitations of the current xFS implementation.

Finally, Chapter 8 summarizes the key conclusions of this dissertation, points out some of the lessons I have learned during this project, and discusses areas that would benefit from further investigation.

2 Trends in Technology

This chapter describes several key trends in technology that will influence the design of file systems for the next decade. Section 2.1 outlines the basic trends to hardware performance that underlie file system design. These trends affect both user demands on file systems and trade-offs in their design. Section 2.2 considers how technologies will drive more demanding file system workloads that will demand scalable file systems. Section 2.3 outlines how opportunities raised by these low-level technology trends impact specific aspects of the serverless design to provide that scalable file service. Finally, to put the serverless approach in context, Section 2.4 discusses other technology trends that affect file systems but that are not explicitly addressed in the serverless design.

2.1. Trends in Technology

Table 2-1 summarizes the current performance and expected rates of improvement for hardware technologies that influence file system design. The data in this table provide a basis for four

Hardware Parameter	1995 Baseline	Yearly Improvement Rate
Disk Latency	12 ms	10%/year
Disk Bandwidth	6-9 MB/s	20%/year
Disk Cost/Capacity	$0.24/MB	100%/year
Network Latency	1 ms	20%/year
Network Bandwidth	20 MB/s	45%/year
Processor Performance	100 SPECInt92	55%/year
Memory Bandwidth	30-70 MB/s	40%/year
Memory Cost/Capacity	$31/MB	45%/year

TABLE 2-1. **Summary of hardware improvement trends.** The 1995 Baseline reflects the performance a high-end, desktop workstation in 1995, and the Yearly Improvement Rate's reflects trends discussed in greater detail in Section 2.1. Although network latency has historically improved slowly as indicated in the table, research in low latency network protocols may provide dramatic improvements in the future. Conversely, while memory cost has historically improved quickly, recent progress has been much slower. This chapter documents the rates of improvement.

general sets of observations. First, disks are slow, and their performance improves slowly relative to other technologies. Second, compared with disks, networks have somewhat better bandwidth and much better latency, and both latency and bandwidth are improving more quickly for networks than for disks. Third, processors and memories are much faster than either disks or networks, and their performance is improving quickly. Fourth, disk storage is two orders of magnitude less expensive than equivalent memory storage, and this gap is widening as disk costs improve more rapidly than memory costs. The rest of this section describes these trends in detail.

2.1.1. Disk Performance

Disk accesses rely on mechanical motions that are slow compared to electronic operations such as DRAM reads. Further, the rate of improvement in disk mechanical performance is lower than the rate of improvement of integrated circuit performance. Disk delays are dominated by three factors: seek time, rotational latency, and transfer time. During the seek time, an actuator moves the disk heads to the disk cylinder being accessed. Rotational latency allows a specific disk block to spin under the disk head, and transfer time allows data to be accessed as it passes under the disk head. Note that for physically sequential transfers, only the first block need pay seek and rotation times.

Table 2-2 summarizes disk performance trends for high-performance, commodity disks appropriate for workstation servers in 1987, 1990, and 1994. The 1987 and 1990 values reflect performance parameters reported by Gibson [Gibson, 1992] while the information for the more recent drive is from a product data sheet [Seagate, 1994].

During this period, seek and rotational latencies have improved slowly, at about 10% per year, while bandwidth has increased more quickly, at about 20% per year. These improvement rates are similar to those discussed by Hennessy and Patterson [Hennessy and Patterson, 1996]. Bandwidth increases more quickly than rotational speed because of increasing storage densities on the disk — more data spins under the disk head at a given rotational speed.

A consequence of disks’ high seek time and rotational latency is that large transfers use disks much more efficiently than small ones. As the table indicates, latency dominates the time to access 8 KB from disk while disk bandwidth dominates the time to transfer 1 MB. However, because disk bandwidths improve more rapidly than other aspects of disk performance, increasingly large transfers are needed over time to maintain a given level of efficiency. For instance, to reduce seek and rotational latency to less than 10% of the total access time required a 562 KB sequential transfer in 1987 and a 988 KB transfer in 1994.

2.1.2. Disk Capacity

Disk capacity per unit cost improves dramatically over time — at 100% per year since 1992. As the data in Figure 2-1 indicate, between 1983 and 1995 the cost of disk storage has fallen by more than a factor of 400, from about \$100/MB in January of 1983 to less than \$0.24/MB in July of 1995. For instance, advertisements in *Byte* magazine in January of 1983 offer a 44 MB hard drive for \$4,395 while advertisements in the July 1995 edition price a 9,000 MB drive at \$2,159. This improvement reflects an increase of 62% per year over the twelve and one-half year period with a faster rate since 1992 as PC manufacturing volumes and competition have ramped up.

2.1.3. Network Latency

Between 1989 and 1994 network RPC latency improved by less than a factor of three, from 2.66 ms [Schroeder and Burrows, 1990] to about 1.00 ms, a speedup of about 20% per year. However, a number of research efforts have demonstrated extremely low network latency using proto-

				Improvement Rates		
	1987	**1990**	**1994**	**1987-94**	**1987-90**	**1990-94**
Model	Fujitsu M2361A	Seagate ST-41600n	Seagate ST-15150n			
Average Seek Time	16.7 ms	11.5 ms	8.0/9.0 ms (rd/wr)	11%/yr	13%/yr	9%/yr
Rotational Speed/Avg. Latency	3600 rpm/8.3 ms	5400 rpm/5.5 ms	7200 rpm/4.2 ms	10%/yr	15%/yr	7%/yr
Bandwidth	2.5 MB/s	3-4.4 MB/s	6-9 MB/s	20%/yr	21%/yr	20%/yr
8 KB Transfer	28.3 ms	18.9 ms	13.1 ms	12%/yr	14%/yr	10%/yr
1 MB Transfer	425 ms	244 ms	123 ms	19%/yr	20%/yr	19%/yr

TABLE 2-2. **Disk performance parameters.** This table compares the parameters of a 1987, 1990, and 1994 commodity disk drive. Although these disks represent only a single data point for each year, I believe the trends accurately describe high-end commodity hardware available over the specified periods. Average seek time is the time it takes to seek one-third of the way across the disk surface, so it represents a pessimistic estimate of the seek time for a typical request because real requests will benefit from locality [Hennessy and Patterson, 1996]. For the 1994 disk, the seek time is higher for writes than for reads because of more stringent head settling requirements for writes. The rotational latency represents the time for the disk to complete half of a complete revolution at the given rotation speed. The bandwidth is the uncached transfer rate for a large data transfer, ignoring seek and rotational latencies. For the 1990 and 1994 drives, transfer bandwidth is higher near the edges of the disk than near the center because of the higher linear velocity of the media near the edges. To provide an upper bound on disk improvement rates, I use the more aggressive read seek time and outer-edge bandwidths when computing improvements over time. The 8 KB and 1 MB transfer times indicate the time to transfer an 8 KB or 1 MB contiguous block of data including the time for one "average" seek and rotational latency, plus the transfer size divided by the disk's maximum bandwidth.

cols somewhat more restrictive than those of RPC. Thekkath, for example, demonstrated a restricted RPC protocol with 72 μs round-trip latency on 155 Mbit/s ATM [Thekkath and Levy, 1993], and Martin and von Eicken have measured 29 μs and 52 μs round trip times for active message implementations using FDDI and ATM, respectively [Martin, 1994, von Eicken et al., 1995]. If any of these simplified protocols could be used as a fast path for file system communication, order of magnitude improvements in network latencies may be possible.

Whether or not research efforts provide new protocols, network latencies are significantly smaller than disk latencies, and they seem likely to improve more quickly in the future. As a result, networks will provide faster access to small data items than will disk.

2.1.4. Network Bandwidth

Over the past 15 years, the aggregate network bandwidth available to a 32-node cluster of high-end, desktop workstations has increased by a factor of 256, from a 10 Mbit/s shared Ethernet in the early 1980's to a 2,560 Mbit/s switched, Myrinet [Boden et al., 1995] network in which half

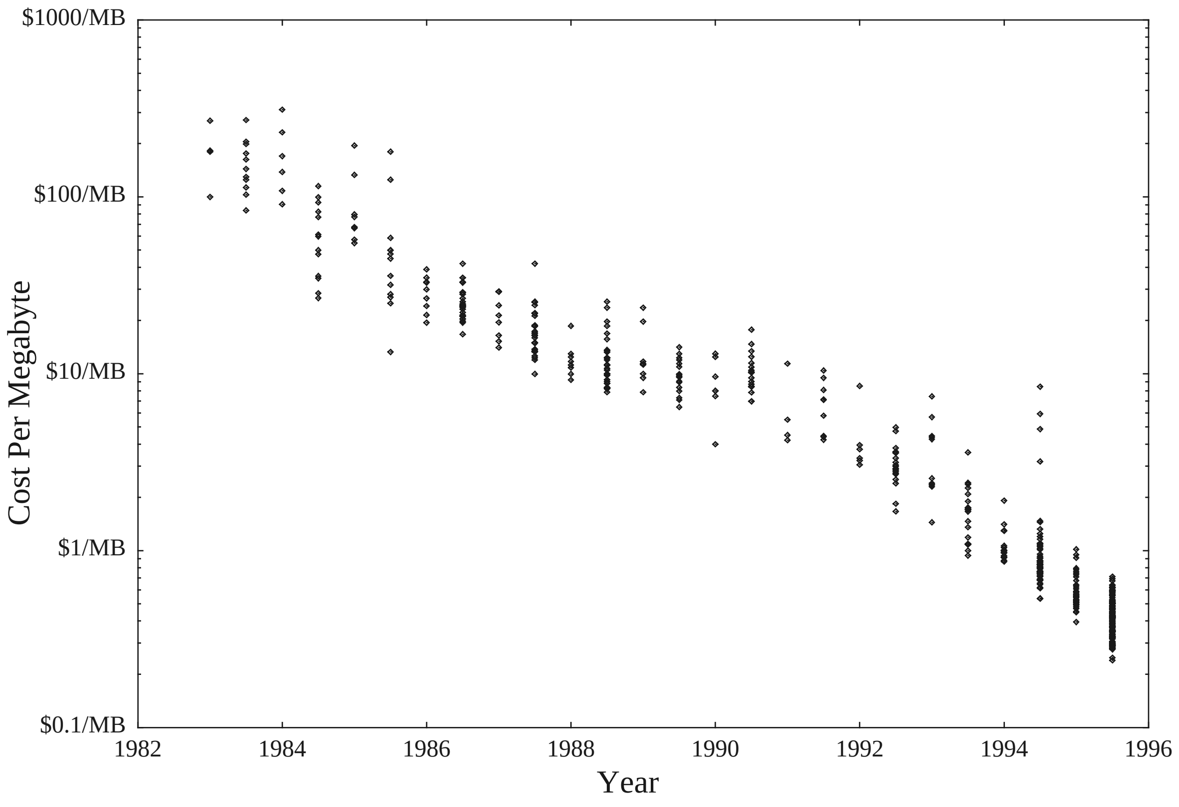


FIGURE 2-1. **Disk cost expressed in dollars per megabyte of capacity.** Each point indicates the capacity in megabytes divided by price in dollars for a hard disk drive advertised in the January or July edition of *Byte* magazine for the indicated year. Note the log scale for the y-axis.

of the workstations can each send at 160 Mbit/s while half receive at the same rate. This improvement represents an increase of about 45% per year over this period. This improvement in network speed has come from two sources, faster network links to each machine, and better topologies to increase the aggregate bandwidth above the per-link bandwidth. The per-link bandwidth has increased by a factor of 16 over this period, at about 20% per year. The other factor of 16 comes from the move from the bus-based Ethernet to switch-based topologies [Schroeder et al., 1991] such as ATM [ATM Forum, 1993], Myrinet, and switched Ethernet.

Although the one-time move from bus-based topologies to switch-based topologies contributed significantly to the improvement in network bandwidths over the last decade, large improvements in network speed are likely to continue because it is easier to increase the speed of switched network links than bus-based network links. First, many switched networks allow individual links to be upgraded independently; in the past, a network's speed could only be increased if all of the machines in a system were simultaneously upgraded. Second, electrical signalling is easier on these point-to-point networks; the bandwidth of the Myrinet network, for instance, is limited by the workstation's I/O bus bandwidth rather than the 640 Mbit/s physical link bandwidth [Martin, 1995].

This evaluation suggests that high-performance networks will be faster than high-performance disks, even for large data transfers.

2.1.5. Processor Performance

Processor performance benefits from improving integrated circuit technologies that allow designers to increase the number of transistors per chip and to increase the speed of these transistors. As a result, processor speed increases at 50% to 55% per year [Gwennap, 1995].

2.1.6. Memory Bandwidth

Improvements to DRAM architectures — including wider memories, memory interleaving, and efficient access modes [Bursky, 1992, Jones, 1992, Prince et al., 1992, Hart, 1992, Bondurant, 1992] — allow memory bandwidths to improve quickly. Table 2-3 summarizes the memory-to-memory copy bandwidths of machines available in 1989 compared to machines measured in late 1994 and suggests that memory bandwidths are increasing at 40% per year or better for each product line. In absolute terms, memory speeds are much faster than disk speeds, and the relatively high rate of memory speed improvements suggest this gap will remain large.

2.1.7. Memory Capacity

The capacity of memory chips increases at about 60% per year due to improvements in technology that quadruple the amount of memory that can fit on a chip every three years [Hennessy and Patterson, 1996]. As Figure 2-2 indicates, before 1992, memory prices reflected this trend, falling at over 100% per year during that time. During the last three years, however, the cost per megabyte of memory has been flat. It is not yet clear whether this drastically reduced rate of improvement in cost is a long-term effect or a temporary blip. Over the full seven-year period indicated in the figure, prices fell by an average of about 45% per year, but the most recent trends suggest that this may be an optimistic estimate for memory-size improvement rates. In any event, memory capacity seems likely to grow more slowly than disk capacity in the future.

2.2. Increasing Demands on File Systems

A key trend that impacts file systems is the dramatically increasing computing power available to users. These increases come from several sources. New, more powerful CPUs increase the demands of sequential applications, and fast networks give users the ability to exploit multiple processors to run distributed or parallel applications, further increasing both average and peak demands on file systems. Furthermore, expanding computing power enables new classes of demanding applications such as databases and multimedia programs that add additional pressure to I/O systems. As a result, I/O systems must advance quickly to avoid limiting system performance. This trend motivates the radically more scalable serverless design.

	1989		**1994**		**Five-year Average Rate of Increase**
Manufacturer	**Machine**	**Bcopy Bandwidth**	**Machine**	**Bcopy Bandwidth**	
SUN	SPARCStation 1	5.6 MB/s	SPARCServer 20 Model 6	27.9 MB/s	38%/year
Hewlett-Packard	9000-835CHX	6.2 MB/s	715/80	32.1 MB/s	39%/year
IBM	RT-APC	5.9 MB/s	6000/370	67.2 MB/s	63%/year
Digital Equipment	DS3100	5.1 MB/s	3000/400	43.8 MB/s	54%/year

TABLE 2-3. **Memory bandwidth for 1989 and 1994 machines.** The 1989 values were measured by Ousterhout [Ousterhout, 1990]. The 1994 values are for large, 10 MB transfers to reduce cache effects. The rate of increase is the change for each manufacturer for the products listed computed over five years. Note that these rates would differ somewhat if different products in each vendor's line were chosen.

2.2.1. Demands From Sequential Applications

The faster improvement rate of processor performance compared to that of disk performance suggests that future I/O systems be designed to take advantage of multiple-disk transfers; improving computers allow users to process more data more quickly, increasing the speed required from the file system. For example, the Case/Amdahl rule of thumb suggests that a balanced machine should have 1 Mbit of I/O bandwidth for each MIPS of CPU performance [Hennessy and Patterson, 1996]. Figure 2-3 plots the actual local single-disk bandwidth against processor performance for ten machines manufactured during the last decade. This graph suggests that high-performance machines with single disks no longer conform to the Case/Amdahl rule of thumb and that the gap is growing larger as processors get faster. The trend discussed above, whereby processors get faster more quickly than disk transfers, suggests that this imbalance will continue to grow.

If anything, Figure 2-3 understates the imbalance. First, it assumes that disk performance improves with disk bandwidth. In fact, because disk latency improves more slowly than disk bandwidth, overall disk performance is likely to improve more slowly as well. Second, as relative disk latencies increase, systems use more aggressive prefetching [Patterson et al., 1995], which

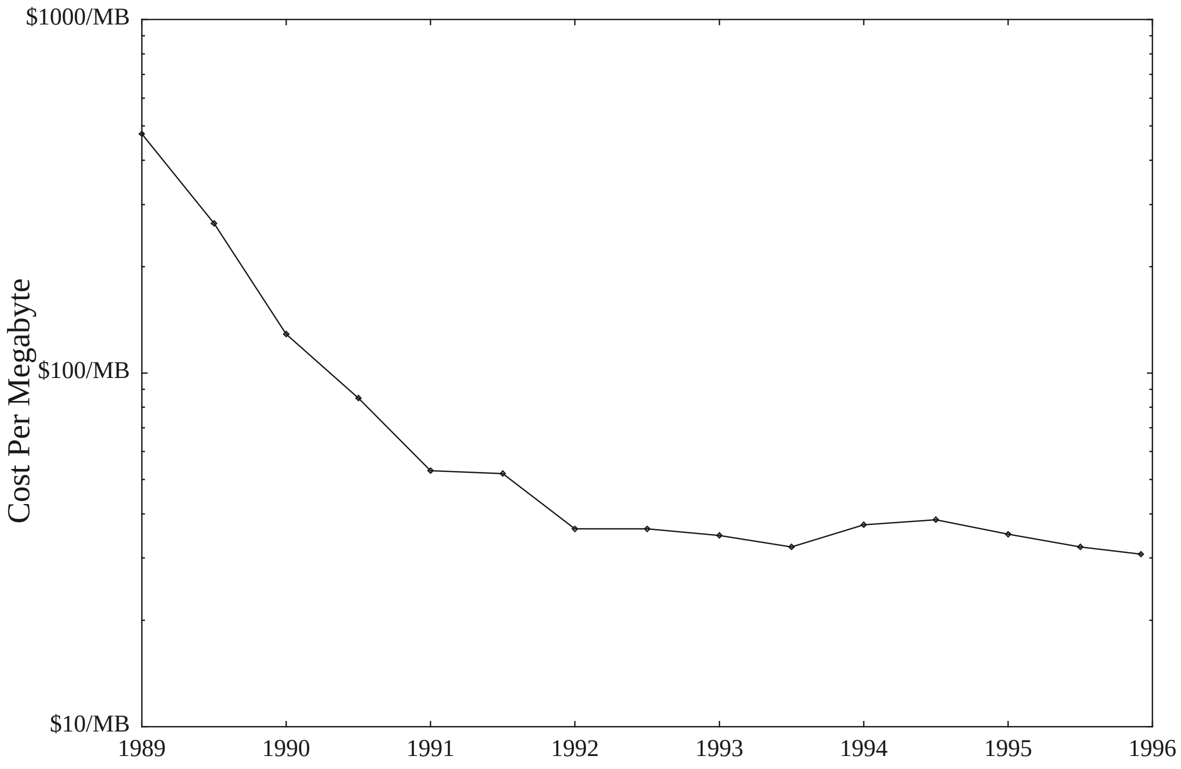


FIGURE 2-2. **DRAM price per megabyte.** Each point indicates the cost per megabyte of DRAM memory based on advertisements from *Byte* magazine. All prices reflect memory purchased in 9-bit-wide SIMM form with access times of 80 ns or better. To provide the best price for each data point, points before July of 1991 are the prices for 1MB SIMMs, and points from July of 1991 and later are one quarter of the prices of 4 MB SIMMs. Note the log scale for the y axis.

increases demands for disk throughput. In effect, prefetching reduces latencies for disk requests by increasing demands for bandwidth.

2.2.2. Demands From Parallel and Distributed Applications

Although sequential processors stretch the limits of conventional file systems, the increasing need for high-performance, parallel file access by mainstream users seems likely to swamp them. Efforts to allow users to harness the aggregate power of many workstations in Networks of Workstations (NOWs) [Anderson et al., 1995] will increase both the average and peak demands on file systems from parallel programs or multiple, simultaneous, sequential jobs.

The ability to harness multiple machines will allow users to tackle larger problems, increasing the total amount of data passed through file systems. Cypher [Cypher et al., 1993] measured the I/O demands of parallel supercomputing applications and found that average I/O requirements increased linearly as the size of a problem increased. This conclusion suggests that the perfor-

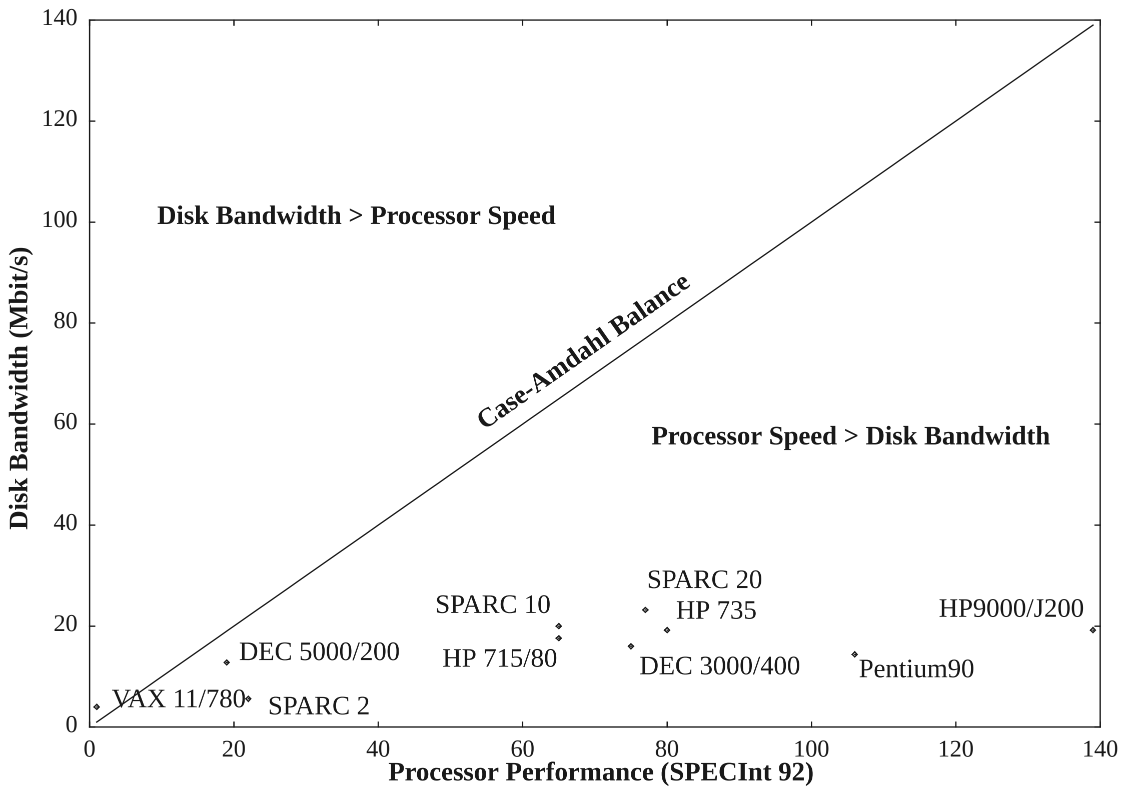


FIGURE 2-3. **Bandwidth and processor performance.** This comparison of local disk bandwidth against processor performance suggests that single-disk sequential machines have too little I/O capacity for their processing power and that this imbalance is increasing as processors get faster. Each point shows the measured bandwidth when reading a file that is too large to fit in the in-memory cache from a single local disk plotted against the published SPECInt performance for the machine. Note that SPECInt92 is used to rate processor power rather than MIPS. If the higher MIPS figures had been used, the machines would appear to be even more out of balance.

mance I/O systems must scale as quickly as that of aggregate — not just sequential — processing for systems that support parallel jobs.

Further, not only will NOWs increase total demands on file systems, these demands are likely to come in bursts as the actions of many machines become correlated. Where traditional file systems could depend on the “law of large numbers” to smooth file system demands over time, parallel and distributed jobs running on NOWs synchronize the actions of many machines. For instance, NOW users may compile programs using parallel `make` rather than sequentially. Even if a parallel `make` demands no more total work from the file system that a sequential one does, requests are sent to the file system much more quickly in the parallel case. To keep from reducing perceived end-user performance, the file system must provide higher peak performance to keep up with increased peak demands from users.

2.2.3. Demands From New Applications

Increasing workstation capabilities also enable new classes of demanding applications. For instance, I/O intensive database applications would challenge conventional file systems because they require large numbers of relatively small, concurrent accesses. Further, database vendors want access to a large number of disks and they want to control how data are distributed over those disks to assist in load balancing.

Continuous media applications such as video and audio will also pressure file systems by requiring them to supply large amounts of aggregate bandwidth. Video applications demand significant bandwidths for periods ranging from seconds to hours, requiring that storage systems accommodate large numbers of simultaneous, demanding users. Furthermore, the sequential nature of video streams may prevent file system caching from being useful, although by the same token prefetching may be effective. Finally, continuous media applications may require real-time guarantees for smooth playback. Although this thesis does not address continuous media applications or real-time guarantees explicitly, it does suggest methods by which storage systems may be built to provide large amounts of aggregate bandwidth. These techniques could then be used as a basis for a continuous media file system.

2.3. Implications For File System Design

Not only do changing technologies increase the demands on file systems, they also change the design trade-offs that should be made to satisfy those demands. Fast, switched, local area networks such as ATM and Myrinet enable serverless file systems by allowing LANs to be used as I/O backplanes, harnessing physically distributed processors, memory, and disks into single file systems. The switched networks provide aggregate bandwidths that scale with the number of machines on the network, and new, low latency network interfaces enable closer cooperation among machines than has been possible in the past. As the rest of this section describes, the trend towards faster, scalable networks motivates each of the pieces of the serverless design, although particular design decision were also shaped by other technology trends.

2.3.1. Cooperative Caching

The serverless design replaces central server caching with cooperative caching that coordinates the clients' caches to allow reads not satisfied by one client's cache to be supplied from another's. Trends in network performance and in memory capacity motivate cooperative caching, which Chapter 3 discusses in detail.

High-speed, low-latency networks provide the primary motivation for cooperative caching by allowing clients to access remote memory much more quickly than they can access remote or even local disk, as Table 2-4 indicates. Where fetching data from remote memory might be only three times faster than getting the data from remote disk on an Ethernet, remote memory may now be accessed ten to twenty times more quickly than disk, increasing the payoff for cooperative caching. At the same time, fast networks reduce the effectiveness of architectures like AFS [Howard et al., 1988] that use local disk as a cache rather than using remote memory; with fast networks, such an approach will be much slower than cooperative caching.

The recent lag in improvements to memory capacity provides a second motivation for cooperative caching. As memory capacity grows more slowly, designers can no longer rely on growing memories to maintain or improve hit rates or to cope with the larger working set sizes enabled by larger disks. Instead, designers must use available memory more efficiently. Cooperative caching does so by providing improved global hit rates with a given amount of memory and by reducing duplicate caching between clients and the server and among clients.

2.3.2. Distributed Storage

A serverless system replaces centralized disk storage with storage servers that distribute storage to different machines' disks. This approach works well because of high-bandwidth networks. Furthermore, to exploit trends in disk technologies, the specific storage server design discussed in Chapter 4 uses redundant, log-structured storage to provide availability and improve write performance.

Fast, scalable networks motivate distributed disk storage. When networks are faster than disks, a single client can read or write at its full network bandwidth by accessing multiple disks that are distributed on the network. Additionally, because a switched network's aggregate bandwidth can be orders of magnitude larger than its per-link bandwidth, the peak aggregate bandwidth of a distributed disk system is the sum of all of the disk bandwidths or network link bandwidths in the system. In contrast, a centralized system's bandwidth is limited to the network link bandwidth of a single machine — the central server — even if the server uses a local RAID to increase its disk bandwidth.

The storage server design described in Chapter 4 stores redundant data on disks to provide high availability. The low cost of disk storage makes this approach to availability more cost-effective than alternatives designs such as replicated servers [Walker et al., 1983, Kazar, 1989, Popek et al., 1990, Liskov et al., 1991, Kistler and Satyanarayanan, 1992, Birrell et al., 1993].

			Ethernet		**77 Mbit/s ATM (Half of 155 Mbit/s)**	
	Local Memory	**Local Disk**	**Remote Memory**	**Remote Disk**	**Remote Memory**	**Remote Disk**
Mem. Copy	250 us	250 us	250 µs	250 µs	250 µs	250 µs
Net Overhead	--	--	400 µs	400 µs	400 µs	400 µs
Data	--	--	6250 µs	6250 µs	800 µs	800 µs
Disk	--	14,800 us	--	14,800 µs	--	14,800 µs
Total	**250 us**	**14,800 us**	**6,900** µs	**21,700** µs	**1450** µs	**16,250** µs

TABLE 2-4. **Cache miss time.** Time to service a file system's local cache miss from remote memory or disk for a slow network, Ethernet, and a faster network, a 155 MBit/s ATM network that achieves half of its maximum link throughput. Copy time for local memory is the measured time it takes to read 8 KB from the file cache on a DEC AXP 3000/400. Network overhead times indicate round-trip, small-packet latencies based on TCP times reported in [Martin, 1994] for a Hewlett-Packard 9000/735 workstation. Transfer figures for Ethernet make the unrealistically optimistic assumption that data is transferred at the full 10 Mbit/s link speed (in reality, because Ethernet utilizes a shared-bus architecture, transfer times could often be several times those listed above). The ATM transfer time assumes that half of the full 155 Mbit/s bandwidth is attained to account for protocol overheads [Keeton et al., 1995]. The disk transfer time is based on measured, physical-disk time (excluding queueing) for the fastest of three systems measured under real workloads by Ruemmler and Wilkes [Ruemmler and Wilkes, 1993].

Trends in disk technology also favor the log-structured storage design used in the storage servers. Log structured file systems [Rosenblum and Ousterhout, 1992, Seltzer et al., 1993] commit modifications to disk using large, contiguous writes to the log; this approach uses disks efficiently by amortizing seek and rotational delays over large writes. Furthermore, log structured file systems are more efficient when free disk space is plentiful [Rosenblum and Ousterhout, 1992]; cheap disks make it feasible to provide extra disk space to get improved performance.

2.3.3. Distributed Management

A serverless file system implements data management separately from data storage. Although this separation can add an additional network hop to access data compared to a central server, low-latency, scalable networks mitigate this cost. Low-latency networks make the additional network hops needed to locate data inexpensive compared to the network or disk transfer of the data itself, and scalable networks prevent congestion that could interfere with low-latency communication.

2.4. Other Technology Trends

A number of other technologies will also influence file systems during the next decade. The increasing distribution of data over wide area networks (WANs), the growing use of portable computers, and the emergence of robotic, tertiary storage libraries will all demand new file system capabilities. This thesis focuses on file service within the context of a LAN with on-disk storage and does not address the opportunities and requirements of these other technologies in detail. To put this thesis in context, this section summarizes the relationship of the ideas it examines to these wider technology trends.

The World Wide Web (WWW) enables data to be distributed widely and shared, and I believe that there are many interesting research questions related to using file system techniques to improve the efficiency and convenience of the WWW. However, because WAN and LAN performance, availability, security, and cost characteristics are so different, the appropriate trade-offs for WAN and LAN file system protocols will be different. As a result, I believe that future WAN file systems will implement two sets of protocols, one for communication within a cluster of machines on a fast LAN network and the second for communication among these clusters over a slower WAN [Sandhu and Zhou, 1992]. Although the research for this thesis is devoted to file systems within a LAN, the techniques described here would also be appropriate for the LAN-specific protocol in a LAN/WAN file system. I have examined protocols appropriate for WAN file systems elsewhere [Dahlin et al., 1994].

This thesis also focuses on the needs of desktop, as opposed to portable, computers. Portable computers pose a number of challenges to file systems stemming from their frequent disconnection from the network [Kistler and Satyanarayanan, 1992] or their use of relatively low-bandwidth, wireless network connections [Le et al., 1995, Mummert et al., 1995]. Because the techniques described in this thesis assume an environment where machines are tightly coupled using a fast LAN, they would not work well as the primary file system for portable computers. In a mixed environment of portable and sedentary computers, however, I would envision these techniques being used on stationary workstations to provide a scalable file system "backbone." In such an environment, the portable machines would run a different protocol such as Coda [Kistler and Satyanarayanan, 1992] or Ficus [Popek et al., 1990], using this scalable backbone as their "server."

Finally, tertiary-storage, robotic libraries [Drapeau, 1993] provide an opportunity for large amounts of data to be stored more cheaply than they can be stored on disk. This advantage exists because these libraries allow massive archives to be built and some system services, such as file backup and restore, to be automated. Because the latency that results when these devices are accessed can be tens or hundreds of seconds, file systems must be carefully structured to hide this latency. Many of the techniques in this thesis address issues related to moving data among different storage servers and might be extended as a mechanism to provide migration to tertiary storage libraries. This thesis does not, however, examine the critical issue of migration *policy* — when to move data between storage levels to mask latency.

3 Cooperative Caching

Cooperative caching seeks to improve file system performance and scalability by coordinating the contents of client caches and allowing requests not satisfied by a client's local in-memory cache to be satisfied by the cache of another client.

Three technological trends discussed in Chapter 2 make cooperative caching particularly attractive. First, processor performance is increasing much more rapidly than disk performance (see Table 2-1 on page 8), making disk accesses a significant impediment to application performance [Rosenblum et al., 1995]. This divergence makes it increasingly important to improve the effectiveness of caching to reduce the number of disk accesses made by the file system. Second, emerging high-speed, low-latency switched networks can supply file system blocks across a network much faster than standard Ethernet, as indicated in Table 2-4 on page 18. Whereas fetching data from remote memory over an older network might be only three times faster than getting the data from a remote disk, remote memory may now be accessed ten to twenty times as quickly as a disk, increasing the payoff for cooperative caching. Third, the rising cost of memory capacity relative to the cost of disk capacity makes it important to use memory efficiently to maintain good cache hit rates. Traditional, central server caching uses the same technology — DRAM memory — for both client caches and server caches. As a result, a significant fraction of the server cache's contents may be duplicated in client DRAM caches, reducing the effectiveness of server caching and wasting precious DRAM capacity [Franklin et al., 1992, Muntz and Honeyman, 1992]. To combat this problem, cooperative caching unifies global and local DRAM caching into a single abstraction and explicitly manages data within the cooperative cache to replicate blocks only when doing so improves performance.

Existing file systems use a three-level memory hierarchy, which implements a limited form of "cooperative caching" by locating a shared cache in server memory in between the other two levels of storage: client memory and server disk. Although systems can often reduce the number of disk accesses by increasing the fraction of the system's RAM that resides in its server, three factors make true, distributed cooperative caching more attractive than physically moving memory from clients to the server. First, central server caching manages DRAM less efficiently than cooperative caching: the memory at the server does not improve clients' local hit rates, the data at one client cannot be accessed by other clients, and the DRAM at clients and servers often inefficiently replicate the same data blocks. Second, a server in a cooperative caching system will be less loaded than a server with a large cache because it can satisfy many requests by forwarding them to the clients' caches rather than having to transfer large volumes of data. Finally, cooperative cache systems are more cost effective than systems with extremely large server caches. For example, in 1995 it would be significantly cheaper to add 32 MB of industry-standard SIMM memory to each of one hundred clients than it would be to buy a specialized server machine capable of holding the additional 3.2 GB of memory. Section 3.3.5 quantifies the trade-offs between centralized and distributed caching in more detail.

Note that the analysis in this chapter assumes that clients cache file system data in their local memories but not on their local disks. Because of the technology trends driving fast local area networks, it will be much quicker for a client to fetch an 8 KB block from another client's memory than to fetch that data from a local disk.

The key challenge to exploiting distributed client memory as a cooperative cache is providing a distributed, global memory management algorithm that provides high global hit rates without interfering with clients' local cache hit rates. In this chapter I examine six cooperative caching algorithms and find that three factors determine their effectiveness. First, to benefit from data sharing, algorithms should allow each client to access the recently accessed data of other clients. Second, algorithms should reduce replication of cache contents by globally coordinating the client caches rather than allowing clients to greedily fill them with locally referenced data. Finally, to permit clients to maintain high hit rates to their local memories while also providing high hit rates to the global cooperative cache, algorithms should dynamically adjust the portions of client memories dedicated to local caching and global caching so that active clients can use most of their caches for local data, while idle clients allow most of their memories to be used for global data. Based on these factors, I have developed a simple algorithm, called N-Chance Forwarding, that provides nearly ideal performance. This algorithm is attractive because it provides good global coordination of cache contents without relying on global communication. Instead, it exploits randomized load balancing to get good performance [Eager et al., 1986, Adler et al., 1995].

To evaluate a range of algorithms, I use simulation studies driven by both traces of file system usage and by synthetic workloads. To provide a simple comparison with current systems, this study assumes that cooperative caching coordinates the contents of client memories in a traditional, central server system that retains a central server cache. Under these assumptions, the trace-based studies indicate that cooperative caching can often reduce the number of disk accesses by 50% or more, improving the read response time of a file system by 30% to 150%[1] for most of the workloads studied. A fully serverless system would get even better performance from cooperative caching by eliminating the separate, central cache memory and, instead, coordinating all of the system's memory as a single, cooperative cache. The synthetic-workload studies verify that cooperative caching will provide significant benefits over a wide range of workloads and that it will almost never hurt performance.

Note that the algorithms examined in this work do not affect the reliability of data storage because cooperative caching only deals with "clean" file system data. If a client modifies a data block and then another client requests that block via cooperative caching, the first client commits the block to disk before allowing the second client to see it.

Section 3.1 describes the six cooperative caching algorithms I examine. Section 3.2 describes the simulation methodology, and Section 3.3 examines key simulation results. Section 3.4 discusses related studies in cooperative caching and global memory management. Finally, Section 3.5 summarizes my conclusions.

3.1. Cooperative Caching Algorithms

This section examines six variations of cooperative caching in detail, covering a range of algorithm designs. Different cooperative caching algorithms could manage remote client memory in many different ways. Figure 3-1 illustrates four fundamental design options and the relationship of the six algorithms to these options. Although these algorithms are by no means an exhaustive set of cooperative caching algorithms, the subset contains representative examples from a large portion of the design space and includes a practical algorithm whose performance is close to optimal.

1. All improvement values for speedup and performance use the terminology in [Hennessy and Patterson, 1996]. Speedup is defined as the execution time of the slower algorithm divided by the execution time for the faster one. The improvement percentages for performance are calculated by subtracting 1.00 from the speedup and then multiplying by 100 to get a percentage.

The rest of this section examines these six algorithms; it then discusses several other possible approaches.

3.1.1. Direct Client Cooperation

A very simple approach to cooperative caching, *Direct Client Cooperation*, allows an active client to use an idle client's memory as backing store. This process works when the active client forwards cache entries that overflow its local cache directly to an idle machine. The active client can then access this private remote cache to satisfy its read requests until the remote machine becomes active and evicts the cooperative cache. The system must provide a mechanism and criteria for active clients to locate idle ones.

Direct Client Cooperation is appealing because of its simplicity — it can be implemented without modification to the server. From the server's point of view, when a client uses remote memory it appears to have a temporarily enlarged local cache. One drawback to this lack of global cooperation is that active clients do not benefit from the contents of other active clients' memories. A client's data request must, for example, go to disk if the desired block no longer happens to be in the limited server memory even if another client is caching that block. As a result, the performance benefits of Direct Client Cooperation are limited, motivating the next algorithm.

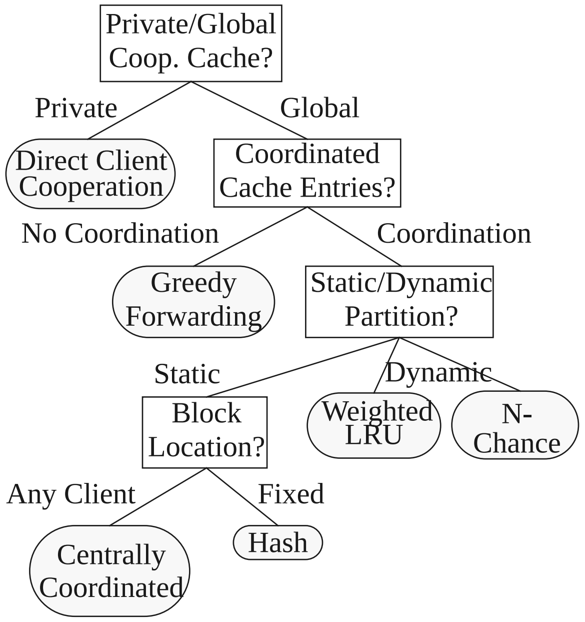


FIGURE 3-1. **Cooperative caching algorithm design space.** Each box represents a design decision while each oval represents an algorithm examined in this study.

3.1.2. Greedy Forwarding

Another simple approach to cooperative caching, *Greedy Forwarding*, treats the cache memories of all clients in a system as a global resource that may be accessed to satisfy any client's request, although the algorithm does not attempt to coordinate the contents of these caches. For Greedy Forwarding, as for traditional file systems, each client manages its local cache "greedily," without regard to the contents of the other caches in a system or the potential needs of other clients.

Figure 3-2 illustrates how Greedy Forwarding allows clients to supply data from their caches to one another. If a client does not find a block in its local cache, it asks the server for the data. If the server has the required data in its memory cache, it supplies the data; otherwise, the server consults a data structure that lists the contents of the client caches. If any client is caching the required data, the server forwards the request to that client. The client receiving the forwarded request then sends the data directly to the client that made the original request. Note that the system does not send the block through the server, because doing so would unnecessarily increase latency and add to the server's workload. If no client is caching the data, the request is satisfied by the server disk as it would have been without cooperative caching.

With Greedy Forwarding the only change to a file system is that the server needs to be able to forward requests, and the clients need to be able to handle forwarded requests; this support is also needed by the remaining algorithms discussed here. Server forwarding can be implemented with the data structures already present in systems implementing write-consistency with callbacks

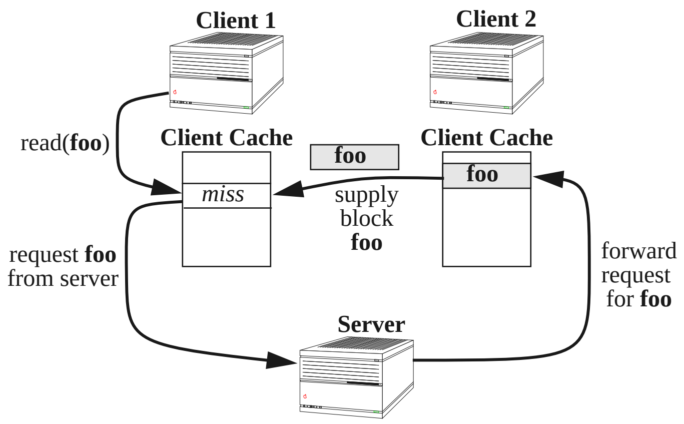


FIGURE 3-2. **Greedy Forwarding.** Using Greedy Forwarding, clients supply data already present in their caches to one another. Client 1 tries to read block foo, not found in its local cache. It requests the data from the Server that then forwards the request to Client 2, which is caching the data. Client 2 forwards block foo to Client 1, satisfying the request.

[Howard et al., 1988] or cache disabling [Nelson et al., 1988]. In such systems, the server tracks the files being cached by each client so that it can take appropriate action to guarantee consistency when a client modifies a file. Cooperative caching extends this callback data structure, sometimes called a *directory* [Lenoski et al., 1990],[1] to allow request forwarding by tracking the individual file blocks cached by each client. For systems such as NFS whose servers do not maintain precise information about what clients are caching [Sandberg et al., 1985], implementation of this directory may be simplified if its contents are taken as hints; some forwarded requests may be sent to clients no longer caching a certain block. In that case the client informs the server of the mistake, and the server either forwards the request to another client or gets the data from disk.

Although cooperative caching's per-block forwarding table is larger than traditional, per-file consistency callback lists, the additional overhead of server memory is reasonable since each entry allows the server to leverage a block of client cache. For instance, a system could implement the forwarding table as a hash table with each hash entry containing a four byte file identifier, a four byte block offset, a four byte client identifier, a four byte pointer for resolution of linked-list collisions, and two pointers of four bytes each for a doubly linked LRU list. In this configuration, the server would require 24 bytes for every block of client cache. For a system caching 8 KB file blocks, such a data structure would consume 0.3% as much memory as it indexes. For a system with 64 clients, each with 32 MB of cache, the server could track the contents of the 2 GB distributed cache with a 6 MB index.

Greedy Forwarding is also appealing because it allows clients to benefit from other clients' caches while preserving fairness — clients manage their local resources for their own good. On the other hand, this lack of coordination among the contents of caches may cause unnecessary duplication of data, which fails to take full advantage of the system's memory to avoid disk accesses [Leff et al., 1991, Franklin et al., 1992]. The remaining four algorithms attempt to address this lack of coordination.

3.1.3. Centrally Coordinated Caching

Centrally Coordinated Caching adds coordination to the Greedy Forwarding algorithm by statically partitioning each client's cache into two sections: one managed locally (greedily by that

1. In this dissertation, I avoid using the term "directory" to refer to cache consistency state to prevent confusion with file directories that provide a hierarchical file name space.

client) and one managed globally (coordinated by the server as an extension of its central cache.) If a client does not find a block in its locally managed cache, it sends the request to the server. If the server has the requested data in memory, it supplies the data. Otherwise the server checks to see if it has stored the block in centrally coordinated client memory. If it locates the data in client memory, it forwards the request to the client storing the data. If all else fails, the server supplies the data from disk.

Centrally Coordinated Caching behaves very much like physically moving memory from the clients to the server for central server caching. The server governs the globally managed fraction of each client's cache using a global replacement algorithm. In this way, when the server evicts a block from its local cache to make room for data fetched from disk, it sends the victim block to replace the least recently used block among all of those in the centrally coordinated distributed cache. When the server forwards a client request to a distributed cache entry, it renews the entry on its LRU list for the global distributed cache. Unless otherwise noted, I simulate a policy where the server manages 80% of each client's cache.

The primary advantage of Centrally Coordinated Caching is the high global hit rate that it can achieve by managing the bulk of its memory resources globally. The main drawbacks to this approach are that the clients' local hit rates may be reduced because their local caches are effectively made smaller and also that the central coordination may impose a significant load on the server.

3.1.4. Hash Coordinated Caching

Hash Coordinated Caching resembles Centrally Coordinated Caching in that it statically divides each client's cache into two portions — local and cooperative cache — but it avoids accessing the server on a hit to the cooperative cache by spreading the contents of the cache across clients based on block identifiers. Each client manages one cache partition that contains blocks selected by hashing on the blocks' identifiers. On a local miss, a client uses the hash function to send its request directly to the appropriate client without first going through the server. That client then supplies the data if it is currently caching that block, or it forwards the request to the server if it does not have the block. To store data in the cooperative cache, the central server sends blocks displaced from its local cache to the appropriate partition of the cooperative cache.

Hash-Coordinated caching performs similarly to Centrally Coordinated Caching. Hash function partitioning of the centrally managed cache has only a small impact on hit rates, and going to the cooperative cache before going to the central server reduces server load because many requests satisfied by the cooperative cache don't go through the server. Direct access to the cooperative cache also improves the latency of its hits, but direct access hurts latency for central server cache hits or disk accesses.

3.1.5. Weighted LRU Caching

To disadvantage of Centrally Coordinated and Hash Coordinated caching is that those algorithms statically partition clients' memories into global and local portions. This approach hurts active clients because it reduces their local hit rates. Furthermore, it fails to take full advantage of idle clients' memories. The next two algorithms address this problem by dynamically balancing the fraction of each client's cache used for local caching and the fraction used for global caching.

The *Weighted LRU* policy uses global knowledge to attempt to make a globally optimal replacement decision whenever it makes room in a cache to add a new block. When a client adds a new block to an already full cache, it ejects either the least-recently-used *singlet* (a block stored in only one client cache) or the least recently used *duplicate* (a block stored in more than one client cache). When deciding between the LRU singlet and LRU duplicate, the algorithm weighs the blocks by the expected global cost of discarding the duplicate, discarding the singlet, or forwarding the singlet to another cache. Hence, the name Weighted LRU. Although the basic algorithm requires potentially unreasonable amounts of communication because it uses global knowledge to make all replacement decisions, a more practical implementation that relies on hints rather than on constant global communication performs nearly as well.

On each cache miss, Weighted LRU performs a global benefit/cost calculation to decide which block to replace. The *benefit* of caching a block is the reduction in latency that will be realized the next time the block is referenced. For instance, if a client keeps a singlet in its local cache and then later references it, it saves ($Latency_{diskAccess}$ - $Latency_{localAccess}$) compared to the time for that reference if it discards the singlet and has to read it from disk. The opportunity *cost* of caching a block until it is referenced is the cache space that it consumes until the reference — the space-time product of its size and the time until the next reference [Smith, 1981]. Because the system cannot know future reference times and because each block in the system is the same size, the algorithm approximates the cost of caching a block until its next reference as the time since the block’s last reference.

When a client makes space in its local cache, it has five options:

1. Discard a duplicate from its local cache.
2. Discard a singlet from its local cache.
3. Forward a singlet from its local cache to a remote client’s cache that has free space.
4. Forward a singlet from its local cache to a remote client’s cache, forcing the remote cache to discard a duplicate.
5. Forward a singlet from its local cache to a remote client’s cache, forcing the remote cache to discard a singlet.

Notice that it never makes sense to forward a duplicate, because having two remote copies saves no time (e.g., adds no benefit) compared to having one remote copy; in either case, if the client later references the block, the latency will be the remote network access time.

Table 3-1 summarizes the benefit/cost of each option. For each replacement, the system chooses to discard or forward the block that results in the smallest reduction in the global cache system’s total benefit/ cost.

If the client discards a local duplicate, only the local machine is hurt because only that machine benefits from the local copy. If any other machine accesses that block, it can get one of the other duplicates as easily as the copy being considered. The local machine, however, benefits from the local copy since it will save a network latency when it next references the block. Thus, the system computes the global reduction in benefit/cost caused by discarding a local duplicate as the difference between the latency of a remote memory hit and a local hit divided by the time since the last local reference to that duplicate, as the first formula in Table 3-1 indicates.

If the client discards a local singlet, it loses the local benefit of that singlet as it does for a local duplicate, but other clients may be hurt as well. In addition to the cost resulting from the fact that the client no longer has a local copy of the data, the next access to that data by any machine in the system will have to access disk rather than reading the block from the client’s memory. Therefore, the system calculates the benefit/cost loss caused by discarding a singlet as the local benefit/cost (calculated as for a duplicate) plus the global benefit/cost: the difference between the latency of remote disk fetch and a remote memory fetch, divided by the time since the last global reference.

The second formula in the table indicates this case. If the last global reference was by the local client, the benefit/cost formula simplifies to $\left(\frac{(Latency_{disk} - Latency_{local})}{timeSinceLastLocalReference_{Singlet}}\right)$ as it should.

Note that the "global" term representing the cost of accessing disk will dominate this equation when disk latency is larger than remote memory fetch latency. This term makes it harder for the system to discard a singlet than to discard a duplicate, because discarding a singlet may result in a future disk access, while discarding a duplicate can only entail a less expensive future network access.

Instead of discarding a singlet, a client can forward one to another client's cache. If the remote cache has extra space available, then — as in the case where a client discards a local duplicate — the client loses local access to the data, but no other clients are hurt. The third formula shows the impact to the global benefit/cost of the cache in this case.

Finally, as the last two lines of the table indicate, a client can forward a singlet to a remote cache and displace either a singlet or duplicate from that cache. In these cases, the client loses the local value of the forwarded singlet, and the remote cache loses the local value of the block it dis-

Option	Expected Lost Benefit/Cost to Global Cache System
Discard Duplicate	$\left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{Duplicate}}\right)$
Discard Singlet	$\left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{Singlet}}\right) + \left(\frac{(Latency_{disk} - Latency_{remoteMemory})}{timeSinceLastGlobalReference_{Singlet}}\right)$
Forward Singlet to Empty Slot	$\left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{Singlet}}\right)$
Forward Singlet to Replace Duplicate	$\left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{Singlet}}\right) + \left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{RemoteDuplicate}}\right)$
Forward Singlet to Replace Singlet	$\left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{Singlet}}\right) + \left(\frac{(Latency_{remoteMemory} - Latency_{local})}{timeSinceLastLocalReference_{RemoteSinglet}}\right) + \left(\frac{(Latency_{disk} - Latency_{remoteMemory})}{timeSinceLastGlobalReference_{RemoteSinglet}}\right)$

TABLE 3-1. **Weighted LRU benefit/cost values.** Weighted LRU clients use these values when deciding whether to discard a duplicate, discard a singlet, or forward a singlet. The system takes the action with the lowest benefit/cost. Latency and time values are in the same time units (e.g., microseconds.) The time since the last local reference to a block is the predicted time until the next reference to the block by the client caching it. The time since the last global reference to a block is the predicted time until any client in the system next references that block.

cards. Additionally, if the discarded remote object is a singlet, the next client in the system to reference that block will have to go to disk rather than to that remote client's cache.

Conceptually, for each replacement, a client evaluates the cost of discarding or forwarding each of its local singlets and duplicates and the cost of discarding each of the remote singlets and duplicates, and it chooses the combination with the least negative impact on the system's global benefit/cost state. In fact, the system will always discard either a locally-LRU duplicate at some client or discard the globally-LRU singlet in the system. Therefore, the system needs to consider only a small subset of the blocks. Locally, it only considers discarding the duplicate that it referenced least recently or discarding or forwarding the singlet that it referenced least recently; evicting any other blocks from the local cache would incur an unnecessarily large "local" term in the benefit/cost calculations. Similarly, remote clients only consider discarding the duplicate they referenced least recently or the singlet referenced least recently by any machine in the system.

3.1.5.1. Limiting Weighted-LRU Communication Requirements

As described so far, Weighted LRU requires considerable communication and computation to make its replacement decisions. Each time a client replaces a block, for instance, it must contact the server to determine if any of the blocks it considers for replacement are singlets, and to evaluate the cost of fowarding a block it must contact all other clients to determine the cheapest block to replace. In addition, whenever a client discards or forwards a block, it must notify the server so that the server can maintain cache consistency and cooperative cache forwarding information. In practice, systems using Weighted LRU replacement would reduce communication by maintaining information to track which local blocks are singlets, by using hints to guess the cost and location of the best remote block to replace, and by combining messages to the server. In the simulations described in this chapter, unless otherwise noted, I use three sets of optimizations described below. Section 3.3.2 examines the performance impact of the hints.

The first set of optimizations reduces the number of messages asking the server if blocks are singlets when clients evaluate the blocks' benefit/cost values. Each client maintains three LRU lists — one for blocks thought to be duplicates, one for blocks thought to be singlets, and one for blocks of unknown state. Clients add blocks to their singlet LRU lists when singlets are forwarded to them by other clients. Clients add blocks to their unknown lists when they read blocks during normal operation. They move blocks from the tail of the unknown list to the singlet and duplicate lists when they search for the LRU singlet and LRU duplicate in their local caches. Each time a client needs to make space in its local cache, it starts by finding its LRU singlet and LRU duplicate. To do this, it first ensures that the oldest singlet is at the tail of the singlet LRU list and that the oldest duplicate is at the tail of the duplicate LRU list. If the tail of the unknown list was referenced less recently than the tail of either the singlet list or duplicate list, that item might be the LRU singlet or LRU duplicate, so the client asks the server which it is and moves the item to the correct list. It continues to do this until the tails of both the singlet and duplicate LRU lists are older than the tail of the unknown list. Once that is true, it can proceed with the benefit/cost analysis.

Note that the singlet/duplicate classification is only a hint — for instance, a duplicate could become a singlet if the other copy is discarded — but reducing communication justifies occasional mistaken benefit/cost calculations. This approach reduces communication because clients never ask the server about blocks forwarded to them by other clients, and they only ask about blocks that they read themselves one time — when the block is about to be replaced because it has not been referenced for a long time and has therefore reached the end of the unknown-status LRU list.

The second set of optimizations uses hints stored at the server and clients to estimate the benefit/cost impact of forwarding a singlet to a remote machine. By using these hints, clients avoid polling all of the other clients each time they consider forwarding a singlet. Instead, the server keeps hints of the benefit/cost impact of forwarding a singlet to each client, and clients keep hints of the current best client to which to forward data and the expected cost of forwarding to that client. Each message from a client to the server includes the current benefit/cost of forwarding a singlet to that client. Each message from the server to a client includes the name of the client that will be least impacted by accepting forwarded singlets and the last known benefit/cost of forwarding a singlet to that client. These values are hints because they can become incorrect if clients access the data that they had planned to sacrifice for the next singlet forwarded to them.

Finally, the system reduces communication costs by combining messages. For instance, the system can piggy-back hints about client benefit/cost values on regular server requests as described in the previous paragraph. Second, clients tell the server about changes to their cache contents in the same message in which they request data. This update indicates what block a client discarded from its cache to make room for the new data, and it indicates where, if anywhere, it forwarded that block.

3.1.6. N-Chance Forwarding

The final algorithm that I quantitatively evaluate, *N-Chance Forwarding*, also dynamically adjusts the fraction of each client's cache that is managed cooperatively, depending on client activity. Like Weighted-LRU, the N-Chance algorithm recognizes that singlets are more globally valuable to the system than duplicates, so it preferentially caches singlets. N-Chance, however, is much simpler than Weighted-LRU because it uses randomized load balancing rather than global knowledge to distribute singlets across the global cache. In fact, the N-Chance algorithm is nearly as simple as the Greedy algorithm: except for singlets, N-Chance works like Greedy Forwarding. The simple bias towards singlets in the algorithm, however, is enough to give it performance comparable to that of the more complex Weighted-LRU approach.

Like Greedy Forwarding, a client in the N-Chance algorithm always discards the locally least recently used object from its cache when it makes space for new blocks. However, if the discarded item is a singlet, the client forwards the singlet to another client's cache rather than allow the last copy of the block to drop out of the cooperative cache. The client that receives the data adds the block to its LRU list as if it had recently referenced the block.

To limit the amount of memory consumed by old singlets, each block has a *recirculation count* that clients increment when forwarding singlets. Clients discard rather than forward singlets whose recirculation count reaches *n*. If a client references a local singlet, it resets the recirculation count to zero, and if a client references a remote recirculating singlet, the remote client discards the singlet after resetting the recirculation count to zero and forwarding it to the client that referenced it. Thus, an unreferenced singlet survives *n* cache lifetimes in the global cooperative cache, giving the algorithm its name; Greedy Forwarding is simply the degenerate case of this algorithm with $n = 0$. Unless otherwise noted, the simulations discussed in this thesis use $n = 2$.

Using the recirculation count, this algorithm provides a dynamic trade-off for each client cache's allocation between local data (data being cached because the client referenced it,) and global data (singlets being cached for the good of aggregate system performance.) Active clients will tend to force any global data sent to them out of their caches quickly as local references displace global data. Idle clients, in contrast, will tend to accumulate global blocks and hold them in memory for long periods of time. One enhancement that I leave as future work is to forward singlets preferentially to idle clients, thus avoiding disturbing active clients. For the current study, clients forward singlets uniformly randomly to the other clients in the system. This simple approach is sufficient to provide good performance for all trace workloads examined.

An implementation of this algorithm must prevent a ripple effect where a block forwarded from one client displaces a block to another client and so on. Note that in the most common case, the displaced block is not a singlet, so no ripple occurs. However, to guard against the uncommon case, the simulator in this thesis imposes a policy that prevents deep recursion from ever occurring: a client receiving a recirculating block is not allowed to forward a block to make space. When a client receives such a block, it uses a modified replacement algorithm, discarding its oldest duplicate. If the cache contains no duplicates, the client discards the oldest recirculating singlet with the fewest recirculations remaining.

Like Weighted LRU, N-Chance Forwarding optimizes communication with the server by using implicit knowledge, hints, and combined messages. In particular, it applies the same knowledge used by Weighted LRU to hint at which blocks are singlets without asking the server in the common case, and it combines updates to cache consistency state with other server messages just like Weighted LRU does.

3.1.7. Other Algorithms

Although the algorithms discussed above cover a broad range of alternatives, some improvements and variations on those algorithm remain to be studied in detail.

Feeley et al. [Feeley et al., 1995] examines a prototype network virtual memory system that uses a global replacement algorithm similar to Weighted LRU. It improves upon Weighted LRU by providing a more practical way to track global age information. To reduce the amount of global knowledge needed, the system divides time into epochs and also estimates the fraction of low-value pages stored at each node. During the epoch, the system forwards global pages to different nodes with probabilities determined by the fraction of low-value pages at each node. This algorithm also includes a cut-off that turns forwarding off when all machines are active. Feeley found that this approach out-performed the N-Chance algorithm when the amount of idle memory in the system was limited and the distribution of idle memory across machines was skewed.

I plan to evaluate this algorithm in detail in the future using the same methodology that I use for the other algorithms in this chapter. I hope to understand the importance of three design decisions. First, Feeley's algorithm uses global knowledge to skew replacement to idle machines; a key issue is quantifying the trade-off between different degrees of global knowledge and performance. Second, the algorithm includes a cut-off to avoid pathological behavior; I will quantify the importance of this feature. Third, the algorithm presented by Feeley favors replacing global-cache singlets rather than duplicates; my hypothesis is that this bias is a bug — systems should value singlets more highly than duplicates because discarding a singlet can cause future disk accesses while discarding a duplicate can only result in less expensive future network accesses [Leff et al., 1991].

Once this evaluation is complete, I plan to modify the N-Chance algorithm to take advantage of any important factors discovered and then evaluate the resulting algorithm. It should be relatively easy to modify the N-Chance algorithm to skew replacement decisions towards idle nodes or to add code to cut-off forwarding when all machines are active. For instance, Eager et al. [Eager et al., 1986] and Adler et al. [Adler et al., 1995] demonstrate a randomized load balancing algorithm for allocating jobs to processes that is similar to the N-Chance approach to cache management. However, where the N-Chance algorithm forwards a block *n* times, this algorithm examines *n* machines, and forwards each job only once — to the least loaded of the *n* machines examined. Just as I find that $n = 2$ works well for N-Chance Forwarding, Adler et al. find that $n = 2$ works well for their algorithm. I hypothesize that Adler's algorithm could be adapted for forwarding singlets among caches and that it would provide most of the performance benefits of Feeley's algorithms for skewed workloads when memory is scarce while retaining much of the simplicity of the N-Chance algorithm.

Finally, work is needed to enforce global resource allocation so that a single user or process cannot consume excessive amounts of the system's resources. Although trace-based simulations studied in this chapter do not encounter this problem, systems should bound the worst-case damage that a resource hog can inflict on other users. The challenge is to balance worst-case fairness and best-case performance.

3.2. Simulation Methodology

I use trace-driven simulation to evaluate the cooperative caching algorithms. The simulator tracks the state of all caches in the system and monitors the requests and hit rates seen by each client. It assumes a cache block size of 8 KB, and it does not allow partial blocks to be allocated even for files smaller than 8 KB. I verified the simulator by using the synthetic workload described in [Leff et al., 1993a] as input, and reproduced Leff's results.

I calculate response times as the weighted sum of the latencies to local memory, remote client memory, server memory, and server disk times the fraction of hits to those levels of the cache hierarchies. My baseline technology assumptions are similar to those of the ATM column of Table 3-2 on page 30, but they assume that the full 155 Mbit/s network bandwidth is achieved. Under these assumptions, an 8 KB block can be fetched from local memory in 250 µs, a fetch from remote memory takes an additional 400 µs plus 200 µs per network hop, and an average disk access takes a further 14,800 µs. Table 3-2 summarizes access times to different resources for the algorithms. In Section 3.3.3 I examine the sensitivity of the results to changes in technology, including different network speeds.

Note that these simulations do not include any queueing delays in response time results. Since the most attractive algorithms studied do not increase server load and since emerging, high-performance networks use a switched topology, queueing would not significantly alter the results.

To maintain data consistency on writes, I assume that modifications to data are written through to the central server and that the server keeps client caches consistent using a write-invalidate pro-

	Local Mem.	Remote Client Mem.	Server Mem.	Server Disk
Direct	250 µs	1050 µs	1050 µs	15,850 µs
Greedy	250 µs	1250 µs	1050 µs	15,850 µs
Central	250 µs	1250 µs	1050 µs	15,850 µs
Hash	250 µs	1050 µs	1250 µs	16,050 µs
Weighted LRU	250 µs	1250 µs	1050 µs	15,850 µs
N-Chance	250 µs	1250 µs	1050 µs	15,850 µs

TABLE 3-2. **Memory hierarchy access times.** Access times for the different levels in the memory hierarchy for different cooperative caching algorithms, assuming that transferring an 8 KB file block takes 250 µs for local memory, 400 µs plus 200 µs per hop for the network, and 14,800 µs for disk as described in Table 3-2 on page 30. A remote client memory hit takes 1250 µs for most algorithms because it includes a local transfer that costs 250 µs, a network transfer that takes 400 µs, and three network hops (client request, server forward request, and client supplies data) requiring 200 µs each. The Direct and Hash algorithms access the cooperative cache in 1050 µs because they save a network hop by accessing it directly. Similarly, a server memory hit takes 1050 µs for most algorithms because clients usually access the server directly. The Hash algorithm, however, requires an extra network hop to get to the server because it first accesses the client indicated by the hash function.

tocol [Archibald and Baer, 1986]. Since this chapter focuses on read performance, a delayed-write or write-back policy would not affect these results.

For most of the results in this chapter, I use traces five and six from the Sprite workload, described in detail by Baker et al. [Baker et al., 1991]. The Sprite user community included about 30 full-time and 40 part-time users of the system, among whom were operating systems researchers, computer architecture researchers, VLSI designers, and “others,” including administrative staff and graphics researchers. Baker gathered four two-day traces of about forty client machines and six servers. For simplicity, my initial evaluation shows results only for one two-day trace that follows the activity of 42 client machines and one server. This part of the trace contains over 700,000 read and write block accesses, and each simulation run uses the first 400,000 accesses (a little over a day) to warm the caches. Section 3.3.4 describes the simulation results for several other workloads including the rest of the Sprite traces.

When reporting results, I compare them against a set of baseline cache-management assumptions and also against an unrealistic best case model. The base case assumes that each client has a cache and that the central server also has a cache, but that the system does not use cooperative caching. The unrealizable best case assumes a cooperative caching algorithm that achieves a global hit rate as high as if all client memory were managed as a single global cache, but one that simultaneously achieves local hit rates as if each client’s memory were managed as a private, local cache. This best case provides a lower bound for the response time for cooperative caching algorithms that physically distribute client memory to each client equally and that use LRU replacement. I simulate this algorithm by doubling each client’s local cache and allowing the clients to manage half of it locally and allowing the server to manage half of it globally, as it does for the centrally coordinated case. For the best case, I assume that clients access data found in remote client memory with three network hops (request, forward, and reply) for a total of 1250 µs per remote memory hit.

3.3. Simulation Results

This section presents the principal results from the simulation studies of cooperative caching. Section 3.3.1 compares the different cooperative caching algorithms to the base case, to each other, and to the unrealizable best case. For clarity, this subsection makes this comparison by assuming a particular set of parameters for each algorithm, a fixed set of technology and memory assumptions, and the use of a single workload. Section 3.3.2 examines the individual algorithms more closely, studying different values for the algorithms' parameters. Section 3.3.3 examines the sensitivity of these results to technology assumptions such as cache size and hardware performance. Section 3.3.4 examines the algorithms under several additional workloads. Finally, Section 3.3.5 summarizes the results, highlights key conclusions, and compares cooperative caching to an alternative strategy — moving more of the system's memory to the server.

3.3.1. Comparison of Algorithms

This section compares the algorithms' response times, hit rates, server loads, and their impact on individual clients. Initial comparisons of the algorithms fix the client caches at 16 MB per client and the server cache at 128 MB for the Sprite workload. For the Direct Cooperation algorithm, I make the optimistic assumption that clients do not interfere with one another when they use remote caches; I simulate this assumption by allowing each client to maintain a permanent, remote cache of a size equal to its local cache, which effectively doubles the amount of memory dedicated to each client. For the Central Coordination algorithm, I assume that each client dedicates 80% of its local cache memory to the cooperative cache and that each manages 20% locally. For the N-Chance algorithm, I choose a recirculation count of two; unreferenced data will be passed to two random caches before being purged from memory. Section 3.3.2 examines why these are appropriate parameters.

Figure 3-3 illustrates the response times for each of the algorithms being examined and compares these times to the base case on the left and the best case on the right. It can be seen that Direct Cooperation provides only a small speedup of 1.05 compared to the base case despite optimistic assumptions for this algorithm. Greedy Forwarding shows a modest but significant performance gain, with a speedup of 1.22; the remaining algorithms that coordinate cache contents to reduce redundant cache entries show more impressive gains. Central Coordination provides a speedup of 1.64, nearly matched by the Hash version of the algorithm with a speedup of 1.63; Weighted LRU improves upon this result with a speedup of 1.74. The simpler, N-Chance Forwarding algorithm nearly equals Weighted LRU for this workload with a performance improvement of 1.73. All four coordinated algorithms are within 10% of the unrealistic best case response time.

Two conclusions seem apparent based on the results illustrated in Figure 3-3. First, disk accesses dominate latency for the base case, so efforts like cooperative caching that improve the overall hit rate and reduce disk accesses will be beneficial. Second, the most dramatic improvements in performance come from the coordinated algorithms, where the system makes an effort to reduce the duplication among cache entries to improve the overall hit rate. The performance of all coordinated algorithms is close enough that other factors such as implementation simplicity and fairness should be considered when selecting among them.

Figure 3-4 provides additional insight into the performance of the algorithms by illustrating the access rates at different levels of the memory hierarchy. The total height of each bar represents the miss rate for each algorithm's local cache. The base, Direct Cooperation, Greedy, and best case algorithms all manage their local caches greedily and so have identical local miss rates of 22%.[1] Central Coordination has a local miss rate of 36%, over 60% higher than the baseline local miss rate. This algorithm makes up for this deficiency with aggressive coordination of most of the

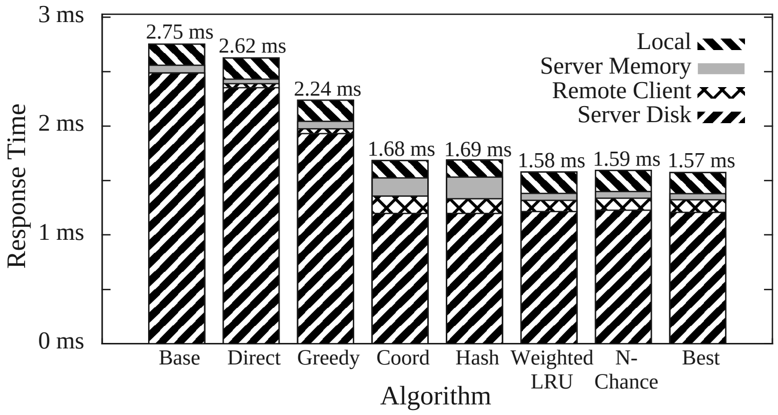


FIGURE 3-3. **Average block read time.** Each bar represents the time to complete an average read for one of the algorithms. The segments of the bars show the fraction of the total read time for data accesses satisfied by *Local* memory, *Server Memory*, *Remote Client* memory, or *Server Disk*.

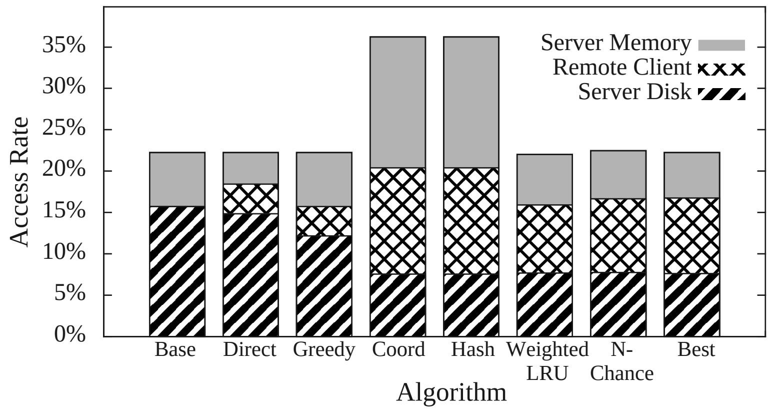


FIGURE 3-4. **Cache and disk access rates.** The bars represent the fraction of requests satisfied at each level of the memory hierarchy for different algorithms. The total height of the bar is the local miss rate for each algorithm. The sum of the *Server Disk* and *Remote Client* segments shows the miss rate for the combined local and server memories. The bottom segment shows the miss rate once all memories are included, i.e. the disk access rate.

memory in the system, which provides global memory miss rates essentially identical to those achieved in the best case, with just 7.6% of all requests going to disk. In other words, Centrally Coordinated Caching's disk access rate is less than half of the 15.7% rate for the base caching scheme; Hash Coordinated Caching performs like Centrally Coordinated Caching.

The two algorithms that dynamically balance singlet and duplicate caching provide local miss rates only slightly worse than the greedy algorithms and disk access rates nearly as low as the algorithms that statically devoted 80% of their memory to global caching. The Weighted LRU algorithm achieves a local miss rate of 22% and a disk access rate of 7.6%. The recirculation of the N-Chance algorithm increases the local miss rate from the greedy 22% rate to 23%, but it reduces the disk access rate to 7.7%.

A comparison between the algorithms that statically partition memory, Central Coordination and Hash Coordination, and those that partition it dynamically, Weighted LRU and N-Chance, illustrates that both the local and global miss rates must be considered when evaluating these algorithms. Although the static algorithms reduce the disk access rate, this reduction comes at the cost of diminished local cache performance. In contrast, the Weighted LRU and N-Chance algorithms interfere less aggressively with local caching, protecting the local cache hit rate but sacrificing some global hits.

Another important metric of comparison is the load on the server imposed by the algorithms. If a cooperative caching algorithm significantly increases server load, increased queueing delays might reduce any gains in performance. Figure 3-5 illustrates the relative server loads for the algorithms compared to the base case.

Because I am primarily interested in verifying that the increased server load of cooperative caching does not also greatly increase server load, I make a number of simplifications when I calculate this load. First, I only include the load associated with servicing read requests; I do not include the load for write-backs, deletes or file attribute requests in the comparison. These other sources of server load are likely to be at least as large as the load from reads; for instance, in the SPEC-SFS NFS server benchmark [Wittle and Keith, 1993], reads account for only about one-third of the server load. Including the other sources of load would add equally to the load for each

1. The simulated local miss rate is lower than the 40% miss rate measured for the Sprite machines in [Baker et al., 1991] because I simulate larger caches than the average 7 MB ones observed in that study and because these larger caches service requests to only one server.

algorithm, reducing the relative differences among them. The results of this simulation can thus be used in support of the hypothesis that there is little difference between the algorithms in terms of server load, or, more specifically, that none of the algorithms greatly increases the server load. It should not be used to claim that one algorithm significantly reduces load compared to another because actual differences will be smaller than shown here.

As another simplification, I base the calculations of server load on the network messages and disk transfers made by the server for each algorithm. I assume that a network message overhead costs one load unit and that a network data block transfer costs two load units. A small network message, therefore, costs one unit; a network data transfer costs one for overhead plus two for data transfer, for a total of three units. I also charge the server two load units for transferring a block of data from disk.

The results in Figure 3-5 suggest that the cooperative caching algorithms do not significantly increase server load, justifying the approximation of ignoring queueing delay. The Centralized Coordinated algorithm does, however, appear to increase server load somewhat, at least under these simple assumptions; the centralized algorithm significantly increases the local miss rate, and clients send all local misses to the server. More detailed measurements would have to be made to determine if the centralized algorithm can be implemented without increasing server queueing delays. The Hash Coordinated version of the static-partition algorithm alleviates the centralized version's increase because requests that hit in the cooperative cache never go to the server.

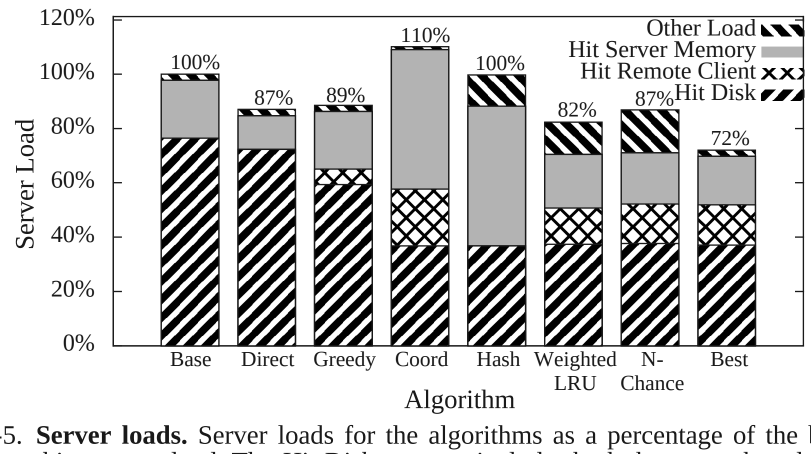


FIGURE 3-5. **Server loads.** Server loads for the algorithms as a percentage of the baseline, no-cooperative-caching server load. The *Hit Disk* segment includes both the network and disk load for all requests satisfied at the server disk. The *Hit Remote Client* segment shows the server load for receiving and forwarding requests to remote clients. The *Hit Server Memory* segment includes the cost of receiving requests and supplying data from the server's memory. Local hits generate no server load. The *Other Load* segment includes server overhead for invalidating client cache blocks and for answering client queries (e.g. Weighted LRU and N-Chance asks, "Is this block the last cached copy?").

A final comparison among the algorithms focuses on individual client performance rather than the aggregate average performance. Figure 3-6 illustrates the relative performance for individual clients under each cooperative caching algorithm compared to that client's performance in the base case. The graph positions data points for the clients so that inactive clients appear on the left of the graph and active ones on the right. Speedups or slowdowns for inactive clients may not be significant, both because they are spending relatively little time waiting for the file system in either case and because their response times can be significantly affected by adding just a few disk accesses.

One important aspect of individual performance is fairness: are any clients significantly worse off because they contribute resources to the community rather than managing their local caches greedily? Fairness is important because even if cooperative caching improves performance averaged across all clients, some clients may refuse to participate in cooperative caching if their individual performance worsens.

The data in Figure 3-6 suggest that fairness is not a widespread problem for this workload. Direct Client Cooperation, Centrally Coordinated Caching, and Hash Coordinated Caching slow a few clients by modest amounts; Greedy Forwarding, Weighted LRU, and N-Chance Forwarding do no harm at all.

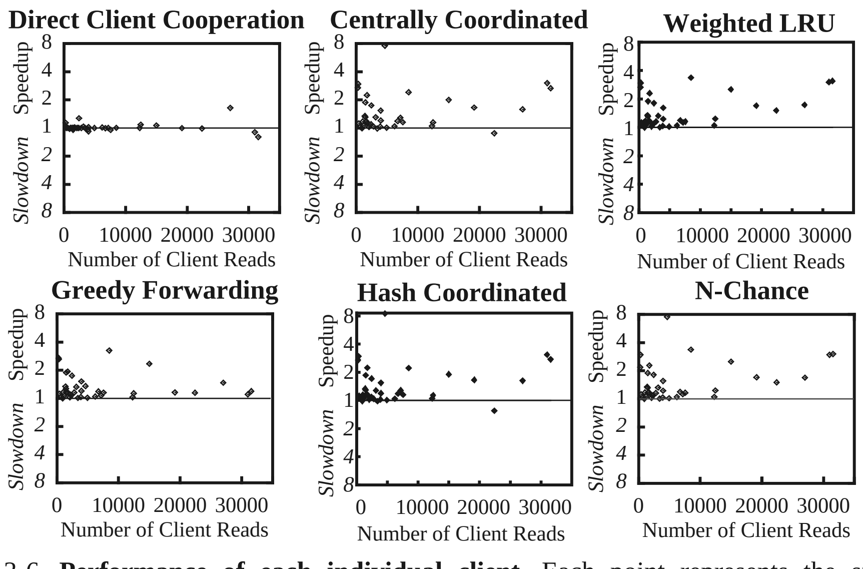


FIGURE 3-6. **Performance of each individual client.** Each point represents the speedup or slowdown seen by one client for a cooperative caching algorithm compared to that client's performance in the base case. Speedups are above the line and slowdowns are below it. A client's slowdown is defined as the inverse of its speedup, if its speedup is less than one. The x-axis indicates the number of read requests made by each client; relatively inactive clients appear near the left edge of the graph, and active ones appear on the right.

Although one would expect the two algorithms that manage client caches greedily to be consistently fair, Direct Client Cooperation causes a few clients' performance to decline up to 25% compared to their performance without the additional cooperative cache memory. This reduction occurs because Direct Client Cooperation does not effectively exploit sharing among clients. The clients' cooperative caches are private, limiting the benefits from cooperative caching, while the server cache hit rates decline compared to the base case because the correlation among client access streams to the server is reduced by clients' accesses to their remote, private caches. The Greedy algorithm, in contrast, is always fair because the cooperative cache exploits sharing among clients just as the central server cache does.

Although the Centrally Coordinated, Hash Coordinated, Weighted LRU, and N-Chance algorithms disturb local, greedy caching to some degree, the significant improvements they yield in global caching provide a net benefit to almost all clients. This trend of widespread improvement is dominant for Weighted LRU and N-Chance Forwarding, which hurt no clients for this workload. Centrally Coordinated Caching damages the response of one client by 14% and Hash Coordinated Caching hurts the same client by 29%. None of these algorithms helps a client whose working set fits completely into its local cache, but such a client can sometimes be hurt by interference with its local cache contents. Because the dynamic algorithms, Weighted LRU and N-Chance Forwarding, interfere with local caching less than the algorithms that partition client caches statically (as was indicated in Figure 3-4), they are less likely to be unfair to individual clients. Note that the current, central server caching approach, whereby cache memory is physically moved from the clients to the server, would suffer from the same vulnerability as the static division algorithms.

The measurements presented in this section suggest that any of the four coordinated algorithms can significantly improve response time but that the dynamic algorithms are superior to the static algorithms by other measures. In particular, the dynamic algorithms are more likely to be fair across all clients because they interfere with local caching less. Likewise, while the Weighted-LRU and N-Chance algorithms provide similar performance, the N-Chance algorithm is significantly simpler to implement because it relies less on global knowledge.

3.3.2. Detailed Analysis of the Algorithms

This subsection examines the cooperative caching algorithms in more detail and evaluates their sensitivity to algorithm-specific parameters.

3.3.2.1. Direct Client Cooperation

Although Direct Client Cooperation is appealingly simple, its performance gains are limited for two reasons. First, clients do not benefit from sharing — a client must access disk even if another client is caching the data it needs. Second, many clients need more than 16 MB of additional cache to get any advantage.

Furthermore, achieving even the modest 5% improvement in the response time seen above may be difficult. The above results were based on the optimistic assumption that clients could recruit sufficient remote cache memory to double their caches without interfering with one another. In reality, the algorithm must meet three challenges to provide even these modest gains.

First, clients may not be able to find enough remote memory to significantly affect performance. Figure 3-7 plots Direct Cooperation's improvement in response time as a function of the amount of remote memory recruited by each client. If, for instance, clients can only recruit enough memory to increase their cache size by 25% (4 MB), the improvement in response time drops to under 1%. Significant speedups of 40% are achieved only if each client is able to recruit about 64 MBs — four times the size of its local cache.

Interference from other clients may further limit the benefits of Direct Client Cooperation. For instance, when a client donating memory becomes active, it will flush any other client's data from its memory. Another client trying to take advantage of remote memory thus sees a series of temporary caches, which reduces its hit rate because a new cache will not be warmed with its data. Studies of workstation activity [Nichols, 1987, Theimer and Lantz, 1989, Douglis and

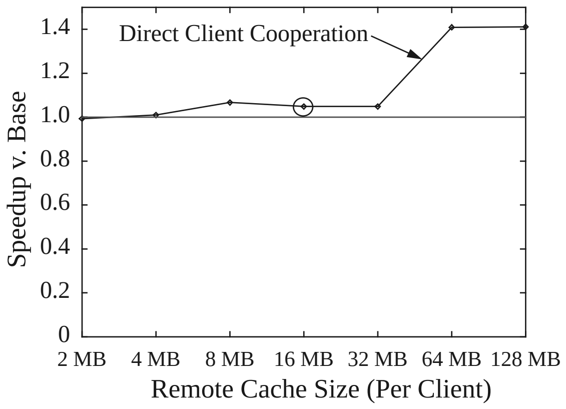


FIGURE 3-7. **Direct Client Cooperation speedup.** The top line indicates the speedup compared to the base case as a function of each client's remote cache size. The circle indicates the result for the 16 MB per client remote cache assumed for this algorithm in the previous section.

Ousterhout, 1991, Mutka and Livny, 1991, Arpaci et al., 1995] suggest that although many idle machines are usually available, the length of their idle periods can be relatively short. For instance, Arpaci et al. found that 70% of idle periods during the working day lasted ten minutes or less. To evaluate the potential impact of periodically vacating remote caches, I ran a simulation in which clients were forced to give up remote client memory after a random period of time that averaged ten minutes; when remote caches were 16 MB, performance declined by about 10% compared to the permanent remote caches illustrated in the figure. To achieve performance equivalent to the permanent remote caches simulated, clients could send evicted data to a new, idle client rather than discarding it, although doing so would increase the system's complexity.

Finally, Direct Client Cooperation must dynamically select which clients should donate memory and which should use remote memory. This problem appears solvable; if only the most active 10% of clients are able to recruit a cooperative cache, they would achieve 85% of the maximum benefits available to Direct Client Cooperation for this trace. On the other hand, the implementation of a recruiting mechanism detracts from the algorithm's simplicity and may require server involvement.

3.3.2.2. Greedy Forwarding

The greedy algorithm provides modest gains in performance, and it is particularly attractive because of its simplicity, because it does not increase server load, and because it is fair. In other words, this 22% improvement in performance comes essentially for free once the clients and server have been modified to forward requests and the server's callback state is expanded to track individual blocks.

3.3.2.3. Centrally Coordinated and Hash Coordinated Caching

Centrally Coordinated Caching can provide significant speedups and very high global hit rates. Unfortunately, devoting a large fraction of each client's cache to Centrally Coordinated Caching reduces the local hit rate, potentially increasing the load on the server and reducing overall performance for some clients. This section provides detailed measurements of Centrally Coordinated Caching to examine what fraction of each client's cache to dedicate to the global cache. These measurements also apply to the Hash Coordinated algorithm.

The fraction of cache that each client dedicates to central coordination determines the effectiveness of the algorithm. Figure 3-8 plots the overall response time against the fraction of cache devoted to global caching. Increasing the fraction improves the global hit rate and reduces the time spent fetching data from disk. At the same time, the local hit rate decreases, driving up the time spent fetching from remote caches. These two trends create a response time plateau when 40% to 90% of each client's local cache is managed as a global resource. Note that these measurements do not take increased server load into account; devoting more space to the global cache also increases the load on the central server because local caches satisfy fewer requests. This effect may increase queueing delays at the server, reducing overall speedups, and pushing the "break-even" point towards smaller centrally-managed fractions.

I chose to use 80% as the default centrally managed fraction because, as Figure 3-9 suggests, that appears to be the more "stable" part of the plateau under different workloads and cache sizes. For instance, the plateau runs from 60% to 90% with 8 MB client caches for the same workload. For large caches, low centrally managed fractions work well because even a small percentage of a large cache can provide a large cooperative cache. A high centrally managed fraction tends to achieve good performance regardless of client cache size because of the large disparity between disk and network memory access times compared to the gap between network and local memory. If the network were slower, a smaller percentage would be appropriate.

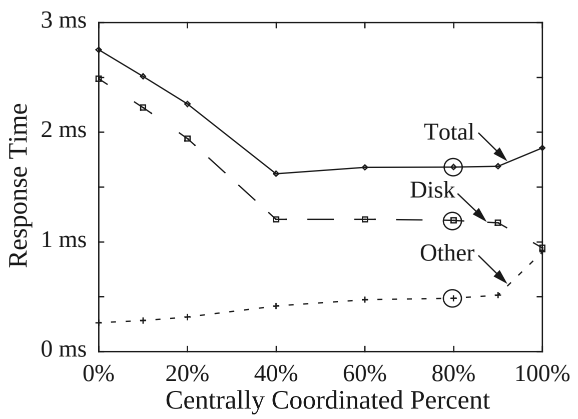


FIGURE 3-8. **Response time for Centrally Coordinated Caching.** Response time varies with the percent of the cache that is centrally coordinated. Zero percent corresponds to the baseline, no-cooperative-caching case. The Total time is the sum of the time for requests that are satisfied by the Disk and the time for Other requests that are satisfied by a local or remote memory. The rest of this study uses a centrally coordinated fraction of 80% for this algorithm, indicated by the circled points.

3.3.2.4. Weighted LRU

The Weighted LRU algorithm I simulate elsewhere in this chapter approximates the ideal Weighted LRU algorithm, but it uses two sets of hints to reduce communication. First, once a client finds out whether a block is a singlet or a duplicate, it assumes that the information remains valid until it evicts the block from the cache or until it forwards a singlet to another client. If another client reads a “singlet” from the server cache or if another client discards its copy of a “duplicate”, the local hint can be wrong. However, because clients ask about the singlet/duplicate status only for blocks near the end of their LRU lists, they tend to discard blocks soon after discovering their status.

The second set of hints avoids computing a global minimum benefit/cost value over all clients’ cache entries. Each time a client asks the server for a block, it also tells the server the current value of its minimum benefit/cost item; the server maintains a table containing hints about all clients’ minimums. The server then informs clients of the current global-minimum hint, including both the lowest benefit/cost value and the client currently caching that item, in its reply to each client’s read request. Clients use this global minimum value hint when evaluating the cost of forwarding a singlet, and, if they do forward a block, they use the hint to decide which client to victimize.

Table 3-3 summarizes the impact of those hints on cache performance. Even using the simple simulation model that ignores queuing delays, the option that reduce network communication by utilizing hints provides nearly the same performance as those systems that use more precise, global knowledge.

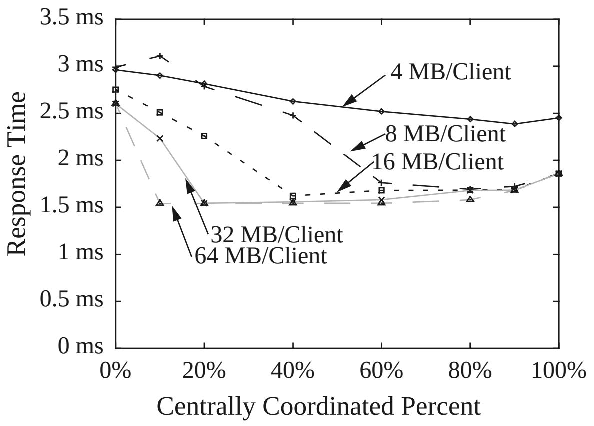


FIGURE 3-9. **Response time plateau.** For Centrally Coordinated Caching the plateau varies as the size of a client’s cache changes.

3.3.2.5. N-Chance Forwarding

N-Chance Forwarding also provides very good overall performance by improving overall hit rates without significantly reducing local hit rates. This algorithm also has good server load and fairness characteristics.

Figure 3-10 plots response time against the recirculation count parameter, *n*, for this algorithm. The largest improvement comes when *n* is increased from zero (the Greedy algorithm) to one. Increasing the count from one to two also provides a modest improvement of about 5%; larger values make little difference. Relatively low values for *n* are effective since data blocks that are recirculated through a random cache often land in a relatively idle cache and thus remain in memory for a significant period of time before being flushed. When the parameter is two, the random forwarding almost always gives a block at least one relatively long period of time in a mostly

Weighted LRU Version	**Response Time**
Global Knowledge	1.57 ms
Singlet Hints	1.57 ms
Singlet Hints + Benefit/Cost Hints	1.58 ms

TABLE 3-3. **Impact of hints on Weighted LRU performance and communication.** The first line shows the characteristics of the algorithm that uses global knowledge for all decisions, the second line shows the performance when clients keep track of which blocks they believe to be singlets, and the final line shows the impact of client and server hints that estimate the benefit/cost and location of the cheapest item to replace if the system forwards a singlet. All other simulations of Weighted LRU in this chapter use the algorithm described in the third line because it significantly reduces communication and without significantly reducing performance.

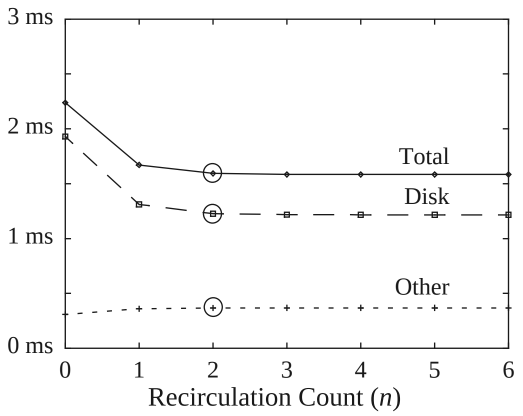


FIGURE 3-10. **Response time for N-Chance.** The performance of the algorithm depends on the number of times unreferenced blocks are recirculated through random caches. Zero corresponds to the Greedy algorithm (no recirculation). The Total time is the sum of the time for requests that are satisfied by going to Disk and Other requests that are satisfied by a local or remote memory. The rest of this study uses a recirculation count of two for this algorithm, indicated by the circled points.

idle cache. Higher values make little additional difference both because few blocks need a third try to find an idle cache and because the algorithm sometimes discards old cache items without recirculating them all *n* times to avoid a “ripple” effect among caches.

3.3.3. Sensitivity to Hardware Parameters

This subsection investigates how sensitive the above results are to assumptions about hardware technology. It first examines the performance of the algorithms for different cache sizes, and then examines the performance as hardware performance changes.

3.3.3.1. Client Cache Size

Figure 3-11 plots the performance of the algorithms as a function of the size of each client’s local cache. The graph shows that the four coordinated algorithms, Centralized Coordination, Hash Coordination, Weighted LRU, and N-Chance Forwarding, perform well as long as caches are reasonably large. If caches are too small, however, coordinating the contents of client caches provides little benefit, because borrowing any client memory causes a large increase in local misses with little aggregate reduction in disk accesses. The simple Greedy algorithm also performs relatively well over the range of cache sizes.

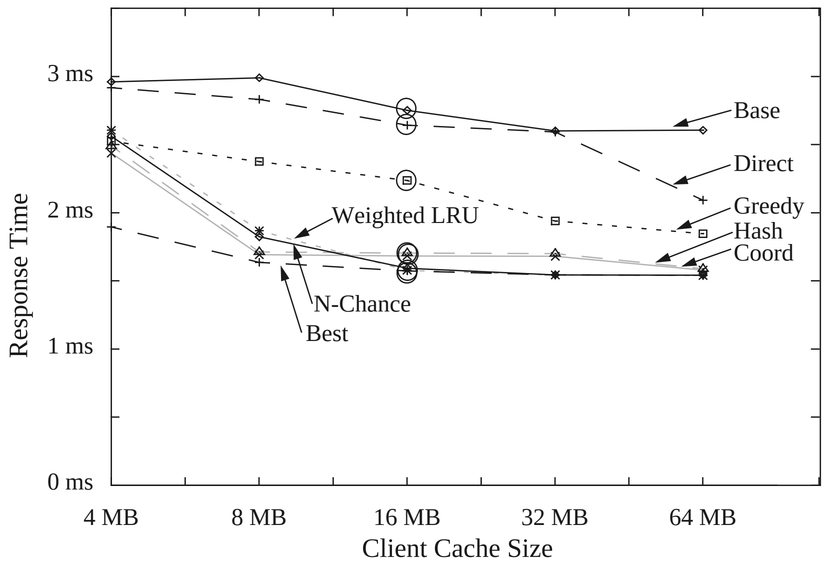


FIGURE 3-11. **Response time as a function of client cache memory for the algorithms.** Previous graphs in this chapter have assumed a client cache size of 16 MB (circled).

3.3.3.2. Dynamic Cache Sizing

Many modern file systems dynamically adjust the size of each client’s cache in response to the demands of virtual memory [Nelson et al., 1988]. Although the simulations examined earlier in this chapter did not dynamically change cache sizes, cooperative caching may be even more attractive for systems that do so.

Current systems’ dynamic cache sizing effectively reduces the size of the most active clients’ caches: clients will have their smallest caches exactly when they need their caches the most! This effect reduces clients’ local hit rates and makes them even more dependant on the improved global hit rates provided by cooperative caching. At the same time, dynamic cache sizing allows idle clients to supply very large amounts of cache memory to the global cooperative cache, improving the global cooperative cache hit rate.

To verify these effects experimentally, I simulated a system assuming that the cache sizes of the ten most active clients were halved, while those of the ten least active clients were doubled. This assumption may be conservative; in many systems, most clients are idle most of the time [Arpaci et al., 1995].

Table 3-4 summarizes response times, comparing N-Chance Forwarding to the base case. Although smaller local caches hurt the performance of the most active clients and thus hurt overall performance, cooperative caching reduces this effect. N-Chance Forwarding’s speedup compared to the base case was 83% under this simple simulation of dynamic cache sizing, compared to 73% for the system with client cache sizes that were static.

3.3.3.3. Server Cache Size

Because cooperative caching attacks the same problem that central server caches do, large central server caches reduce the benefits of cooperative caching. Figure 3-12 illustrates the effect of varying the size of the central server cache. Increasing its size significantly improves the base,

	Base	**N-Chance**	**Speedup**
Static Cache	2.75 ms	1.59 ms	1.73
Dynamic Cache	3.05 ms	1.66 ms	1.83

TABLE 3-4. **Read response time.** These results are for the Base and N-Chance algorithms under both the default Static Cache assumptions and simple assumptions meant to represent the effects of Dynamic Cache sizing. Dynamic cache sizing was simulated by halving the cache size of the ten most active clients and doubling the cache size of the ten least active clients.

no-cooperative-caching case, while only modestly improving the performance of the cooperative algorithms that already have good global hit rates. For sufficiently large server caches, cooperative caching provides no benefit once the server cache is about as large as the aggregate client caches. Such a large cache, however, would double the cost of the system's memory compared to using cooperative caching. Note that when the server cache is very large, Centrally Coordinated Caching and Hash Coordinated Caching perform poorly because their local hit rates are degraded.

Although it would appear that server memories would increase in size and make cooperative caching less attractive over time, Figure 3-12 showed performance for a workload that was generated in 1991. The technology trends discussed in Chapter 2 suggested that workloads tend to increase in size nearly as quickly as memories. Thus, I expect the performance advantage of cooperative caching to remain significant in the future.

3.3.3.4. Network Speed

One of the motivations for cooperative caching is the emergence of fast, switched networks. 10 Mbit/s Ethernet-speed networks, even if switched, are too slow to derive large benefits from cooperative caching because transferring a block from remote memory takes almost as long as a disk transfer. Fortunately, emerging high speed networks, such as ATM, Myrinet, and 100 Mbit/s Ethernet, promise to be fast enough to see significant improvements. Figure 3-13 plots response time as a function of the network time to fetch a remote block. For an Ethernet-speed network, where a remote data access can take nearly 10 ms, the maximum speedup seen for a cooperative

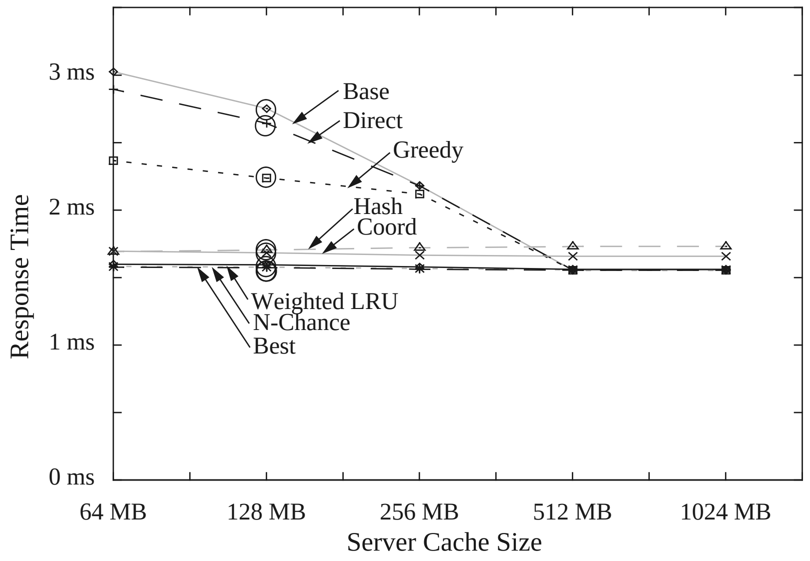


FIGURE 3-12. **Response time as a function of size of central server cache.** The circled points highlight the results for the default 128 MB server.

caching algorithm is 20%. If, however, network fetch times were reduced to 1 ms, for instance by using an ATM network, the peak speedup increases to over 70%. This graph shows little benefit from reducing a network's block fetch time below 100 μs because once the network is that fast, it is not a significant source of delay compared to the constant memory and disk times assumed in the graph.

Although any of the coordinated algorithms used in this study can provide nearly ideal performance when the network is fast, the dynamic Weighted LRU and N-Chance Forwarding algorithms appear to be much less sensitive to network speed than the static Centrally Coordinated and Hash Coordinated algorithms. Static partitioning of client memory only makes sense in environments where accessing remote data is much closer in speed to accessing local data than going to disk. Otherwise, the reduced local hit rate outweighs increased global hit rate.

3.3.3.5. Future Projections

Figure 3-14 projects the performance of cooperative caching in the future, assuming the trends in technology outlined in Chapter 2 and summarized in Table 3-5 continue. Since cooperative caching replaces disk transfers with network transfers, and since network speeds are improving more rapidly than disk speeds, cooperative caching performance improves relative to the base case over time. However, this relative improvement is a modest one because, as Figure 3-3 suggests, disk access time dominates response time even under cooperative caching; improvements to other

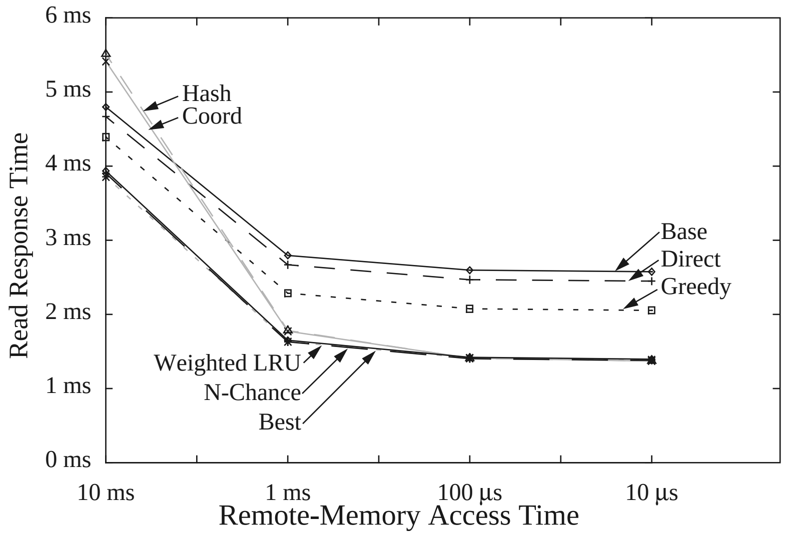


FIGURE 3-13. **Response time as function of network speed.** The x axis is the round trip time it takes to request and receive an 8 KB packet. Disk access time is held constant at 15 ms, and the memory access time is held constant at 250 μs. For the rest of this study I have assumed 200 μs per hop plus 400 μs per block transfer for a total remote fetch time of 800 μs (request-reply excluding memory copy time), indicated by the vertical bar.

technologies do not change total response time significantly once networks are “fast enough.” For instance, the speedup for N-Chance Forwarding increases from 73% in 1994 to 87% in 1999 under these assumptions.

Note that this projection accounts for neither the increase in memory sizes expected in the future nor the expected increase in disk and workload sizes. These trends largely offset each other [Baker et al., 1991], resulting in little increase in effective cache size. Figures 3-11 and 3-12 suggest that significant increases in effective cache sizes would reduce the need for cooperative caching slightly because they would reduce the need to manage memory carefully.

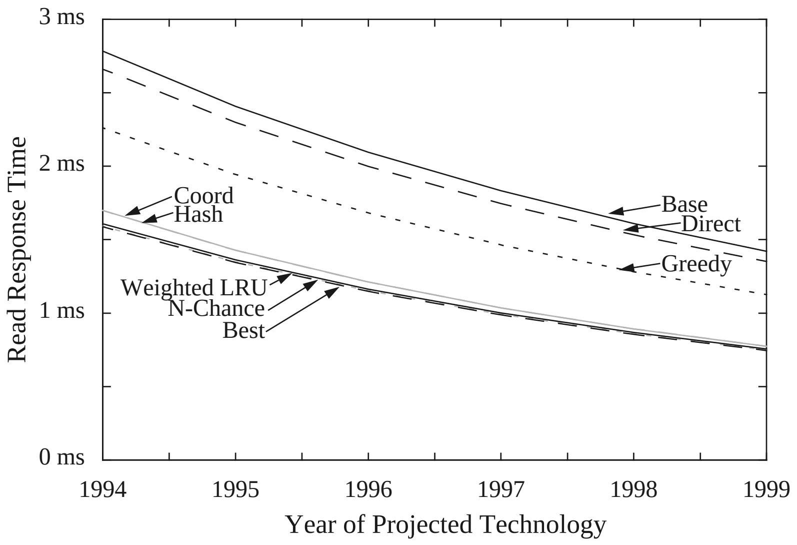


FIGURE 3-14. **Cooperative caching response time under assumed technology trends.** Note that while the absolute difference (measured in milliseconds) between cooperative caching and the base case falls over time, the relative difference (e.g. the speedup) increases, suggesting that cooperative caching will become more valuable in the future.

Coop. Cache Parameter	**Performance Trend**	**1994**	**1995**	**1996**	**1997**	**1998**	**1999**
Memory Copy	40%	250 us	179 us	128 us	91 us	65 us	46 us
Net Overhead	20%	200 us	167 us	139 us	116 us	96 us	80 us
Bandwidth	45%	400 us	276 us	190 us	131 us	90 us	62 us
Disk Overhead	10%	11,000 us	10,000 us	9,091 us	8,264 us	7,513 us	6,830 us
Bandwidth	20%	4,000 us	3,333 us	2,778 us	2,314 us	1,929 us	1,608 us

TABLE 3-5. **Technological trends.** Summary of technological trends relevant to the performance of cooperative caching. For more detail, see Chapter 2. Although this table assumes that performance of all aspects of the system continuously improve, in fact, different parts of systems improve discontinuously as users upgrade different pieces of their systems at different times. However, over the long term, performance upgrades should resemble the trends outlined here.

3.3.4. Other Workloads

To evaluate the performance of cooperative caching over a wider range of environments, the next three subsections simulate the algorithms using several additional workloads. The first subsection uses several other Sprite traces, and the next subsection looks at cooperative caching performance for a system with more clients than Sprite had. Finally, Section 3.3.4.3 uses synthetic workloads to bound the worst-case performance of the algorithms.

3.3.4.1. Other Sprite Traces

Figure 3-15 illustrates the response time for the algorithms under four additional Sprite traces gathered by Baker et al. [Baker et al., 1991]. Each trace covers a two-day period and includes the activity from all Sprite servers. The third graph, labeled Traces 5 and 6, covers the same time period examined in the previous sections, but it includes the activity of file systems exported by all servers as opposed to the single file server trace examined previously.

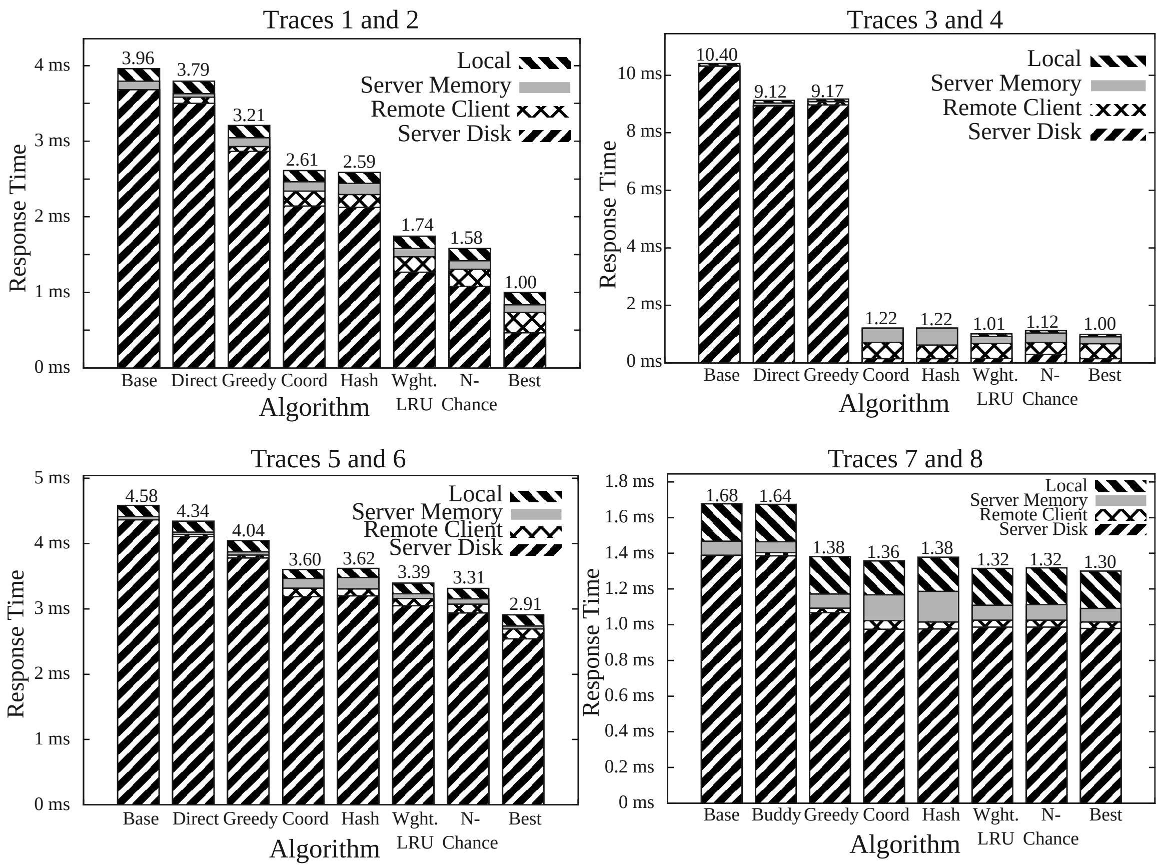


FIGURE 3-15. **Response time for four, two-day Sprite traces.** The traces are labeled by their names in [Baker et al., 1991], which analyzed each day separately. The graph labeled Traces 5 and 6, covers the same time period examined in the previous sections, but it includes activity to file systems exported by all servers as opposed to the single file server trace examined previously. Note that the graphs have different scales for their y-axis.

The results for these traces follow the previous results in most respects. Cooperative caching provides significant speedups for all of the traces, with the best performance coming from the coordinated algorithms in general and the dynamic, coordinated algorithms in particular. The numerical speedups observed, however, vary significantly from trace to trace. The first, third, and fourth pairs of traces (traces 1 and 2, 5 and 6, 7 and 8) show results qualitatively similar to the earlier ones. For instance, N-Chance Forwarding provides speedups of 2.51, 1.38, and 1.27 for these traces. The differences among these traces relates to the amount of data touched by the workloads. In traces 7 and 8, the workload touches relatively little data, so all of the algorithms that share client caches work well. Traces 5 and 6 access more data than will fit in the system's memory, so even the unrealistic best case algorithm has a relatively high response time of 2.91 ms; for this trace, the dynamic coordinated algorithms manage memory better than the less sophisticated algorithms. For traces 1 and 2, the working set just barely fits in the system's memory; the unrealistic best-case algorithm has no capacity misses, but all of the other algorithms do. Again, coordination of memory proves to be important for achieving good results.

For traces 3 and 4, N-Chance Forwarding achieves a remarkable speedup of 9.29, and the other coordinated algorithms perform similarly. The dominant activity during this trace is a single user processing large, trace files [Baker et al., 1991]. Because that user's working set does not fit in a machine's local memory, the base, Direct, and Greedy algorithms are ineffective. The remaining algorithms that coordinate client memories do quite well, and dynamic coordination is not vital because, although the user's working set is too large for a single machine, it is small compared to the system's total memory resources.

Figure 3-16 shows the server load for the four sets of Sprite traces; cooperative caching does not appear to significantly increase load compared to central server caching. In Traces 1 and 2 and in Traces 3 and 4, cooperative caching may reduce server load slightly, and in Traces 5 and 6 and Traces 7 and 8, it increases load by at most 20%. As noted earlier, because there are several other significant sources of server load, the impact of cooperative caching on total server load may be even less than indicated in the graphs. Also note that the Direct algorithm sometimes (in Traces 1 and 2) reduces load compared to the base case because clients can cooperate without accessing the server. Similarly, for all of the traces, the Hash Coordinated algorithm reduces load compared to the Centrally Coordinated algorithm because clients access the cooperative cache without accessing the server.

3.3.4.2. Berkeley Auspex Workload

The Berkeley Auspex workload provides another set of data points to evaluate cooperative caching. This workload traces the NFS file system network requests for 237 clients in the U.C. Berkeley Computer Science Division that were serviced by an Auspex file server during one week of October, 1993. As in the Sprite traces, the users consist primarily of computer science researchers and staff. At the time this trace was gathered, the department's Auspex file server had 64 MB of file cache and most clients had between 32 MB and 64 MB of DRAM. The simulations, however, use the same parameters as they do for the Sprite traces: 16 MB of file cache per client and 128 MB of server cache.

This trace differs from the Sprite traces in three ways. First, it is more recent; this traces was gathered in late 1993, while the Sprite traces were gathered in 1991. Second, it includes the activ-

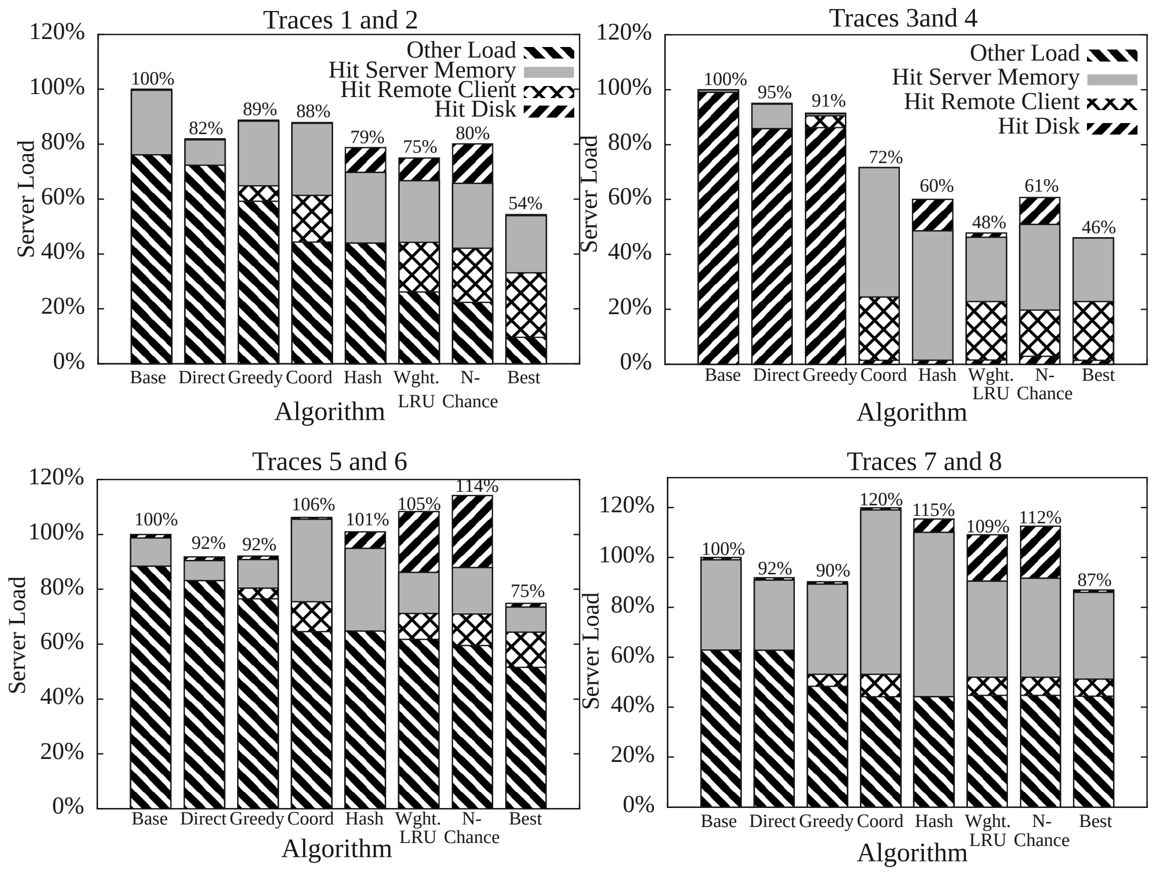


FIGURE 3-16. **Server loads for the algorithms.** Loads are expressed as a percentage of the baseline, no-cooperative-caching server load for the four, two-day Sprite traces. The traces are labeled by their names in [Baker et al., 1991], which analyzed each day separately. The graph labeled Traces 5 and 6 covers the same time period examined in the previous sections, but it includes activity to file systems exported by all servers, as opposed to the single file server trace examined previously.

ity of a much larger number of clients; the Berkeley Auspex workload tracks the activity of 237 active clients while Sprite had fewer than 50. The large number of clients provides an extremely large pool of memory for cooperative caching to exploit. Third, this seven-day workload covers a longer period of contiguous time than any of the two-day Sprite traces; this reduces the impact of start-up effects. The simulator uses the first one million read and write events to warm the caches and gathers statistics during the remaining four million events.

The trace has one major limitation, however. It was gathered by snooping on the network; because of this, it does not include clients' local hits. To approximate the response time results based on this incomplete trace, I use Smith's Stack Deletion method [Smith, 1977]. Smith found that omitting references that hit in a small cache made little difference in the number of faults seen when simulating a larger cache. The actual miss rate can be accurately approximated by dividing the number of faults seen when simulating the reduced trace by the actual number of references in the full trace.[1] As a further refinement, I used the *read attribut*e requests present in the trace to more accurately model the local client LRU lists. NFS uses read-attribute requests to validate cached blocks before referencing them. The simulator can, therefore, use read-attribute requests as a hint that a cached block is being referenced even though the block requests do not appear in the trace. The attribute requests still provide only an approximation — an attribute cache hides attribute requests validated in the previous three seconds, and not all read-attribute requests really signify that a file's cached blocks are about to be referenced — but they do allow the simulator to infer some of the "missing" block hits.

Although the results for the Auspex workload are only approximate, they support the results seen for the Sprite workloads as Figure 3-17 indicates. The relative ranking of the algorithms under the Auspex workload follows the results for the Sprite workload: Centrally Coordinated Caching, Hash Coordinated Caching, Weighted LRU, and N-Chance Forwarding work nearly as well as the best case, and the Greedy algorithm also provides significant speedups; Direct Cooperation provides more modest gains. This result is insensitive to the "inferred" hit rate; the exact

1. Unfortunately, the Auspex trace does not indicate the total number of references. For the results in Figure 3-17, I assume a "hidden" hit rate of 80% (to approximate the 78% rate simulated for the Sprite trace), giving a maximum speedup of 2.02 for cooperative caching. If the local hit rate were higher, all of the bars would have a slightly larger constant added and the differences among the algorithms would be smaller (e.g. a 90% local hit rate reduces the best case speedup to 1.68). If the local hit rate were lower, the differences would be magnified (e.g. a 70% local hit rate gives a best case speedup of 2.22).

speedup predicted for the Auspex workload depends on the inferred hit rate, but cooperative caching provides significant advantages over a wide range of assumed local hit rates.

Figure 3-18 shows the impact cooperative caching has on server load for the Auspex workload, and the results generally follow the same pattern as for the other workloads. However, the Hash Distributed Caching has a higher load than the Centrally Coordinated algorithm. Because of the large number of clients for this workload, the static hash function reduces the effectiveness of the global cooperative cache compared to the centrally coordinated version. As a result, more requests are satisfied by the server cache in the Hash algorithm than in the Centrally Coordinated one.

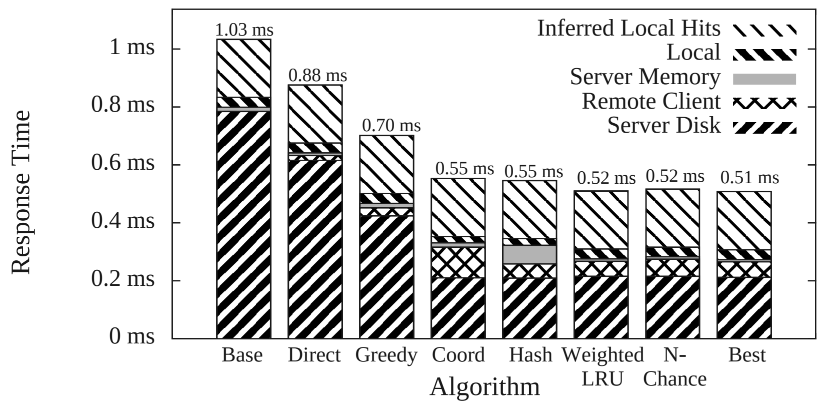


FIGURE 3-17. **Response time for algorithms under the Auspex workload.** The *Inferred Local Hits* segment indicates an estimate of the amount of time spent processing local hits that do not appear in the incomplete Auspex traces, assuming that the traced system had an 80% local hit rate.

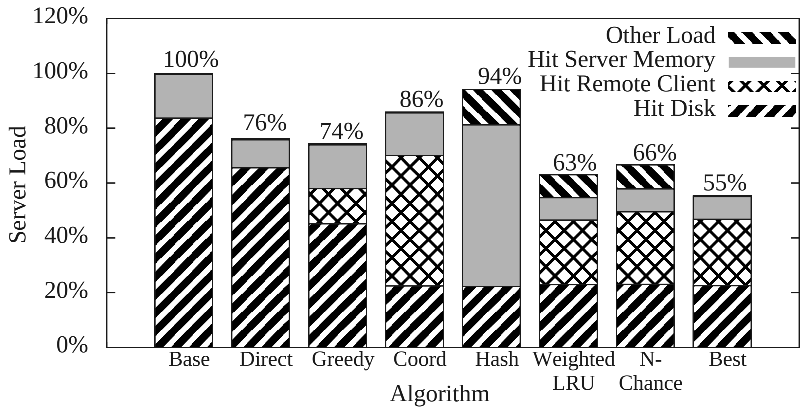


FIGURE 3-18. **Server load for the algorithms.** Load is expressed as a percentage of the baseline, no-cooperative-caching load for the Auspex trace.

3.3.4.3. Worst-case Workloads

Although the above results suggest that cooperative caching can provide significant speedups compared to central server caching, the algorithms can interfere with clients' local caching. Cooperative caching can therefore conceivably hurt performance for some workloads. This section bounds the worst case performance of cooperative caching and finds that (1) cooperative caching degrades the performance of very few workloads, and (2) even in the worst cases, cooperative caching hurts performance only slightly compared to the base case. Thus, it appears that cooperative caching is worth the risk for most types of workloads.

The coordinated algorithms — Centrally Coordinated Caching, Hash Coordinated Caching, Weighted LRU, and N-Chance Forwarding — can potentially hurt a system's performance compared to greedy algorithms. Three factors determine when these algorithms hurt performance: the working set size of the clients, the load imbalance among the clients, and the degree of data sharing among clients. This section uses a synthetic workload to control these key factors and examine their impact on the performance of the coordinated algorithms compared to the greedy algorithm as well as to the base case.

The synthetic workload controls the three criteria independently. Each client has a set of "private" data, the size of which determines the client's "working set size." Clients access data in bursts that are on average much longer than their local cache size; the simulator randomly determines the number of requests in each string of accesses by generating a uniform random number between 0.1 and ten times the cache size. The length of time between a client's bursts controls the system's load balance. If the interval between bursts is short (i.e. 0), then all clients are constantly active, and there is never even a temporary load imbalance to exploit. On the other hand, if the inter-burst time is long, then during a client's burst, it is likely that most other clients are inactive, which means that the active clients can exploit the resources of idle ones. Finally, the workload controls the degree of sharing by allowing a fraction of each client's requests to go randomly to other clients' private data. If 100% of requests go to a client's private data, there is no sharing, while if 0% of a client's requests go to private data then all data are shared equally.

The simulations in this section fix the number of clients at ten and the size of client caches at ten elements per client. Further, I turn off server caching and use the same base assumptions about technology as in the previous sections: a local hit takes 250 us, a server hit takes 1,050 us, a remote client hit takes 1,250 us, and a disk hit takes 15,850 us.

Based on these simulations, it appears that the dynamic, coordinated algorithms, Weighted LRU and N-Chance Forwarding, can only hurt performance in a restricted set of cases when *all* three parameters take “bad” values: when clients are accessing a working set slightly larger than their local caches, when load is almost completely balanced across machines, and when there is virtually no sharing of data among machines. Further, because the local miss penalty is relatively small when data are found in remote memory, the worst-case degradation is limited to about 20%. However, the static Centrally Coordinated Caching and Hash Coordinated Caching algorithms degrade performance compared to the base case under a wider set of conditions and can hurt performance more than the other algorithms because they can hurt both local and global hit rates in degenerate cases.

Working Set Size

The sizes of clients’ working sets determine the degree to which cooperative caching can hurt performance in the rare cases when it degrades performance compared to the base case. If working sets are small enough to fit into local client caches, then cooperative caching is not be invoked, and it cannot hurt performance. If, on the other hand, clients have extremely large working sets, then they spend most of their time going to disk with or without cooperative caching. In that case, the additional network latencies imposed by incorrect cooperative caching coordination decisions make little difference to average performance.

Figure 3-19 illustrates the performance of the cooperative caching algorithms and the base case as a function of working set size. Note that the other synthetic load parameters are set to the worst case for cooperative caching: 100% of each client’s accesses go to its local data set, so no data are shared, and all clients are always active, so there is no idle memory to exploit. As a result, as the working set size increases above 100% of the local cache size, an increasing number of requests have to go to disk, regardless of the algorithm used.

For this workload, the base and Greedy algorithms provide the best performance over the entire range of working set sizes; because there is no sharing and no idle memory, the system does best when clients do not interfere with one another’s caches. The Weighted LRU and N-Chance algorithms provide slightly inferior performance when the working set size is larger than the cache size because clients forward data to one another, reducing their local hit rates without improving the system’s global hit rate.

Despite the fact that cooperative caching cannot possibly help this workload, the worst-case degradation is small for N-Chance and Weighted LRU. At worst, their performance is 20% worse than the base case when the working set size is 10% larger than local caches. For small working set sizes, the system does not invoke cooperative caching and suffers no degradation. For large working set sizes, disk accesses dominate the response time, but the algorithms have no effect on the time spent accessing disk because they do not change the global hit rate.

Centrally Coordinated Caching suffers poor performance over a wider range of working set sizes. Because this static algorithm uses only 20% of each client's local cache for local data, client performance begins to degrade when the size of the working set grows larger than 20% of a client's local cache. Further, because this version of the algorithm allows duplication between the local and centrally coordinated portions of the caches, the effective size of the global cache is just 80% of the sum of the client caches. Therefore, disk access rates begin to rise sharply once the working set size is larger than 80% of a client's cache size. Although not shown in the graph, the performance of Hash Coordinated Caching or traditional, central server caching is comparable to that of Centrally Coordinated Caching.

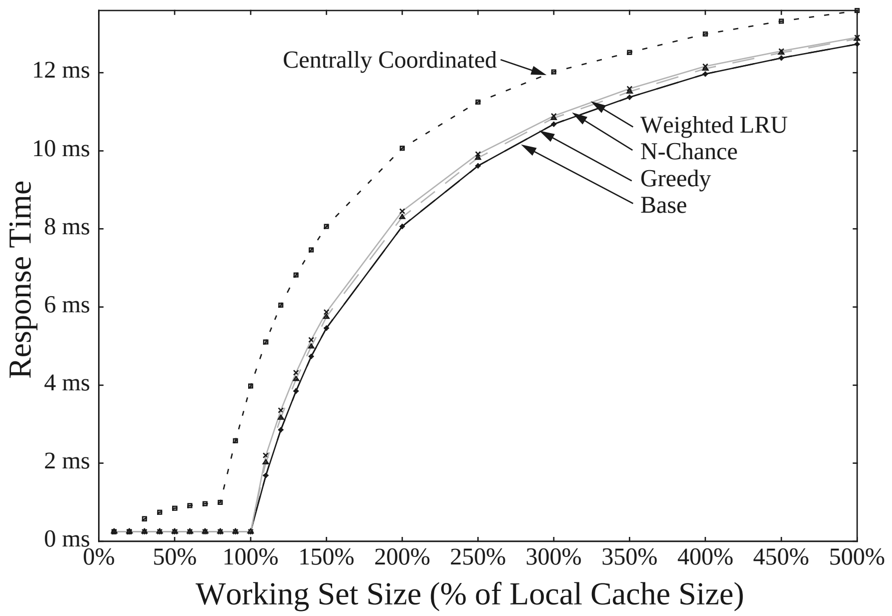


FIGURE 3-19. **Effect of working set size.** The effect of working set size on cooperative caching performance when sharing and burst-size are set to their worst case values -- no sharing and no gaps between bursts. The x axis indicates the amount of private data from which clients randomly select their accesses as a percentage of their local cache size, and the y axis indicates the response time. For instance the data points at x = 200% show the response time when each client touches twice as much data as will fit in its local cache.

Load Balance

Coordinated cooperative caching algorithms exploit load imbalances by using idle clients to supply data to active clients. Figure 3-20 illustrates the effect of varying the load balance in the otherwise worst-case situation where clients touch strictly private data with a working set 1.1 times the size of their local caches. When there is little or no load imbalance to exploit, the base case and Greedy algorithm do the best. However, as the inter-burst time increases, active clients can exploit the resources of idle clients, increasing the benefit of both the static and dynamic coordinated algorithms. Once the inter-burst time is equal to the burst time, the dynamic coordinated algorithms outperform the base and Greedy algorithms, even for what is otherwise the worst case scenario for those algorithms.

Sharing

Both the Greedy and coordinated algorithms exploit client sharing by allowing clients to access data from one another's caches. Figure 3-21 illustrates this effect by varying the fraction of requests that go to local data from 0% (no locality) to 100% (no sharing) with all machines active all of the time and with the workings set 1.1 times the size of a client's local cache. Near the left edge of the graph, widespread sharing allows all varieties of cooperative caching to outperform the base algorithm despite the lack of load imbalance and the awkward working set size. As locality

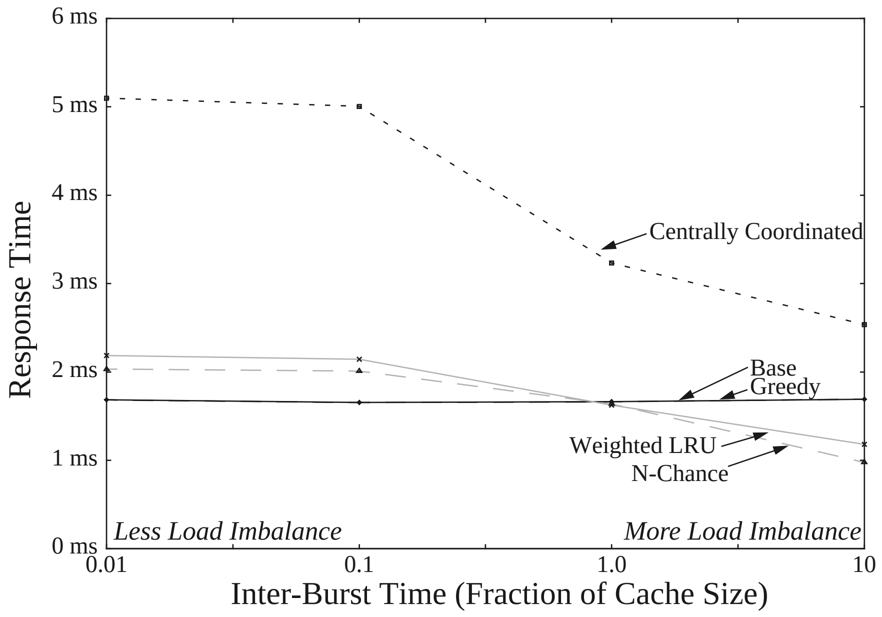


FIGURE 3-20. **Effect of load imbalance.** Each client waits for a random inter-burst time between 0.01 and 10 times the average cache size and then fires off a burst of requests. The size of the bursts is a random, uniform number between 0.1 and 10x the local cache size. Clients do not share data, and the working set size is 1.1 times the local cache size. Note that when the average burst is larger than the working set size, clients tend to revisit data during a burst.

increases towards the right of the graph, all algorithms except the Centrally Coordinated one show significant improvements in performance. When there is no sharing, the base and Greedy algorithms perform best; however, the Weighted LRU and N-Chance algorithms outperform them for all but the most insular workloads.

3.3.5. Summary of Simulation Results

N-Chance Forwarding is a relatively simple algorithm that appears to provide very good performance over a wide range of conditions. Weighted LRU provides similar performance but may be more complex to implement. Centrally Coordinated and Hash Distributed Caching can also provide very good performance, but they are more likely to degrade the performance of individual clients, they depend heavily on fast network performance to make up for the reduced local hit rates they impose, and they increase server load compared to the dynamic algorithms.

The Greedy Forwarding algorithm appears to be the algorithm of choice if simplicity is paramount. Although the Direct Cooperation algorithm is also simple, it is much less effective because it does not exploit the fact that clients share data.

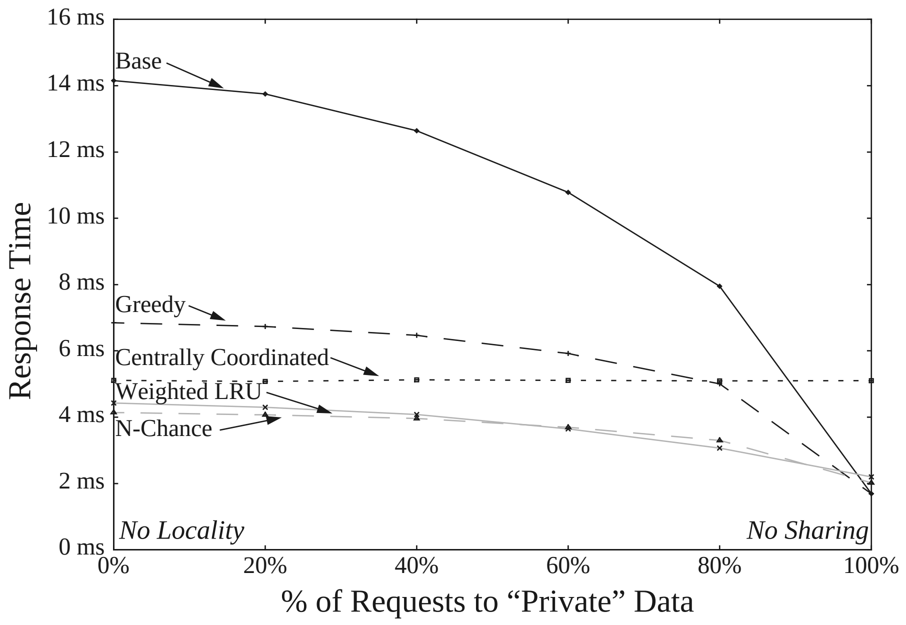


FIGURE 3-21. **Effect of sharing.** Each client accesses the indicated percentage of data from its private working set and randomly accesses blocks from all other clients’ private sets. In other respects the workload represents the worst case for the coordinated algorithms with no inter burst interval (all clients are always active) and with working set sizes 1.1 times the size of each client’s local cache.

Finally, consider the alternative to cooperative caching: physically moving more memory from clients to a traditional, central server cache. The results of this section show that distributed, dynamic cooperative caching is a more effective way to organize memory than static, traditional central server caching for at least six reasons.

• Cooperative caching provides better performance.

Central server caching is equivalent to the Centrally Coordinated algorithm and provides similar performance: moving 80% of client cache memory to the server yields improvements of 66% and 93% for the Sprite and Auspex workloads compared to the standard distribution of memory. These speedups are good, but they fall short of equalling the N-Chance algorithm because of their lower local hit rates of 64% and 44% (compared to 77% and 66% for N-Chance) resulting from smaller local caches.

• Dynamic cooperative caching is more fair to individual clients.

Static allocation of the global/local caches is more likely to provide bad performance for some individual clients as was seen in Figure 3-6 for Centrally Coordinated Caching.

• Dynamic cooperative caching provides better worst-case performance.

As indicated by the Centrally Coordinated algorithm’s lines in Figures 3-19 through 3-21, central server caching can perform poorly for a relatively wide range of workloads.

• Dynamic coordination is less sensitive to network speed.

A system with more cache memory at the server and less at the clients is sensitive to network speed as was seen in Figure 3-13 for Centrally Coordinated Caching. If the performance of networks falls compared to the performance of local memory, moving memory to the server becomes less attractive.

• Central server caching increases server load.

Reducing the size of local client caches can increase server load because the server then transfers more data from its memory to clients. The load for reads under a traditional caching system with the enlarged central cache is 50% higher than for N-Chance Forwarding under the Sprite workload.

• Large, central server caches stress server memory capacity.

Configuring servers with large amounts of memory may be less cost-effective than spreading the same amount of memory among the clients. For instance, 80% of the 16 MB of cache memory for the 237 clients in the Auspex trace would be 3 GB of memory, which would demand an extremely expandable and potentially expensive server.

3.4. Related Work

This chapter evaluates the performance benefits and implementation issues of cooperative caching. Its primary contributions are evaluating realistic management algorithms under real file system workloads and a systematic exploration of implementation options.

Leff et al. [Leff et al., 1991, Leff et al., 1993b, Leff et al., 1993a] investigate remote caching architectures, a form of cooperative caching, using analytic and simulation-based models under a synthetic workload. Two important characteristics of their workload were that the access probabilities for each object by each client were fixed over time and that each client knew what these distributions were. Leff found that if clients base their caching decisions on global knowledge of what other clients are caching, they could achieve nearly ideal performance, but that if clients made decisions on a strictly local basis, performance suffered.

The studies in this chapter differ from Leff's studies in a number of important ways. First, this chapter includes actual file system reference traces as a workload, allowing it to quantify the benefits of cooperative caching achievable under real workloads. A second major feature of this study is that it has focused on getting good performance while controlling the amount of central coordination and knowledge required by the clients rather than focusing on optimal replacement algorithms.

Franklin et al. [Franklin et al., 1992] examined cooperative caching in the context of client-server data bases where clients were allowed to forward data to each other to avoid disk accesses. The study used synthetic workloads and focused on techniques to reduce replication between the clients' caches and the server cache. The server did not attempt to coordinate the contents of the clients' caches to reduce replication of data among the clients. Their "Forwarding—Sending Dropped Pages" algorithm is similar to my N-Chance Forwarding algorithm, but they send the last copy of a block to the server cache rather than to another client.

Feeley et al. [Feeley et al., 1995] implemented unified buffer cache that implemented cooperative caching for both file system and virtual memory pages. They implemented a global, coordinated, dynamic algorithm similar to Weighted LRU, but used epochs to limit the amount of global knowledge required by the implementation. One set of microbenchmarks indicated that this algorithm provided better performance than N-Chance Forwarding when free memory was scarce and distributed unevenly across machines. Future work is needed to determine if a small amount of global knowledge can be added to N-Chance's randomized load balancing to retain the simplicity of the N-Chance algorithm while providing good performance in memory-constrained environments.

Blaze [Blaze, 1993] proposed allowing file system clients to supply hot data to each other from their local on-disk file caches. The focus of this work was on reducing server load rather than improving responsiveness. He found that the use of client-to-client data transfers allowed *dynamic hierarchical caching* and avoided the store and forward delays experienced by static hierarchical caching systems [Muntz and Honeyman, 1992].

The idea of forwarding data from one cache to another has also been used to build scalable shared memory multiprocessors. DASH hardware implements a scheme similar to Greedy Forwarding for dirty cache lines [Lenoski et al., 1990]. This policy avoids the latency of writing dirty data back to the server when it is shared. The same optimization could be used for a cooperative caching file system that uses delayed writes. Several "Cache Only Memory Architecture" (COMA) designs have also relied on cache-to-cache data transfers [Hagersten et al., 1992, Rosti et al., 1993].

Other researchers have examined the idea of using remote client memory rather than disk for virtual memory paging. Felten and Zahorjan [Felten and Zahorjan, 1991] examined this idea in the context of traditional LANs. Schilit and Duchamp [Schilit and Duchamp, 1991] scrutinized using remote memory paging to allow diskless portable computers, and Iftode et al. [Iftode et al., 1993] explored using memory servers in parallel supercomputers. Comer and Griffioen proposed a communications protocol for remote paging [Comer and Griffioen, 1992].

3.5. Conclusions

The advent of high-speed networks provides the opportunity for clients to work closely together to significantly improve the performance of file systems. This chapter examined the technique of cooperative caching and concluded that it can reduce read response times by nearly a factor of two for several of the workloads studied and that a relatively simple algorithm allows clients to efficiently manage their shared cache.

This analysis of cooperative caching algorithms suggests that coordinating the contents of client caches is vital to providing good global hit rates as well as good overall performance. This analysis further suggests that the N-Chance algorithm proposed here achieves such hit rates without hurting local hit rates and without requiring excessive server coordination.

4 Distributed Disk Storage

The serverless system's distributed disk storage subsystem stores all of the system's durable state, including file data blocks, metadata blocks, and data structures used internally by other parts of the system. The performance, availability, and scalability of the storage system is therefore crucial to the goals of the file system as a whole. To provide high performance, the disks must allow high bandwidth access to a single file or to multiple files by one or more clients, but the disks should also handle small file writes efficiently [Baker et al., 1991]. The system should therefore distribute data across multiple disks and machines for parallelism, but it should still support efficient small writes. To provide high availability, the system must allow continued access to the data when some machines fail, and it must provide support to help higher levels of the system recover after crashes. To do this, the system should store data redundantly, and it should provide some sort of reliable logging mechanism. To meet the scalability goals, the system should efficiently support hundreds or thousands of disks. It should therefore control the parallelism of disk striping independently from the number of disks, and it should allow the amount of redundant data in the system to increase as more disks are added.

Fast, scalable networks enable scalable, distributed storage systems by making it possible to harness disks from different machines in parallel, using the network as an I/O backplane. When disks are distributed across multiple machines connected by a fast network, a single client can write to the disks at a rate limited only by that client's network link, and multiple clients can access the system's disks at a rate limited only by the sum of the disks' bandwidths or the aggregate network bandwidth of the system. Also, by distributing disk storage across multiple machines and by storing data redundantly, the system can ensure that all data blocks are available, even if some of the machines or disks crash. Finally, fast networks make distributed disk storage scalable. Because the disks are located on multiple machines, no single machine's processor, memory, network link, or I/O bus limits the system's throughput. Even if all of the machines' I/O busses are saturated, for instance, the system can increase its raw disk bandwidth by connecting new machines and disks to the network.

Several disk storage systems that support subsets of the serverless storage system's goals have been built. As I discuss in Section 4.6, however, while these systems provide parallel disk storage, few provide scalability, high availability, or support for small writes.

One system, Zebra [Hartman and Ousterhout, 1995] does meet many of the serverless storage system's goals. Zebra's log-based, network striping combines the ideas of Redundant Array of Inexpensive Disks (RAID) [Patterson et al., 1988, Chen et al., 1994] and Log-structured File Systems (LFS) [Rosenblum and Ousterhout, 1992, Seltzer et al., 1993] and adapts them for distributed storage. The combination of redundant storage and log structured storage provides the high bandwidth and high availability of RAID with the straightforward crash recovery and good small-write performance of LFS. However, Zebra's scalability falls short of that needed by a serverless system because all client writes involve all of the system's disks, because Zebra relies on a single, sequential log cleaner to garbage collect free disk space to use for logging writes, and because Zebra relies on a single, sequential metadata manager.

This chapter describes a design that addresses the limitations of the Zebra system to make large-scale, log-based striping practical. To improve performance and availability, it employs stripe groups like those proposed for large RAIDs, and to prevent the log cleaner from throttling throughput it exploits distributed cleaning. Furthermore, the next chapter will describe the synergy between this design and the distributed manager. The distributed manager coordinates metadata without limiting the throughput of the disk storage system, and the distributed storage system's redundant, log-structured storage system enables distributed manager crash recovery.

The rest of the chapter proceeds as follows. First, Section 4.1 examines how Hartman's Zebra system combines RAID and LFS for distributed, log-based storage, and it compares log-based, network striping to alternative network disk architectures. The next two sections describe enhancements to Zebra to make that design scale well: Section 4.2 describes stripe groups, and Section 4.3 presents the design of a distributed cleaner. The next section investigates how to use the serverless system's large, cooperative cache to reduce cleaning overhead. Then, Section 4.5 focuses on issues related to fault tolerance. Then Section 4.6 describes previous work related to the disk subsystem design presented here. Finally, Section 4.7 highlights the key ideas behind the design and summarizes my conclusions.

4.1. Log-based Network Striping

This section describes the basic design of a log-based, striped storage system. It first reviews the concepts introduced by RAID and LFS, after which it outlines the Zebra design that combines these ideas and extends them to distributed systems. Next, it outlines the limitations of the basic Zebra design, and then it considers alternative designs of storage systems that do not use log-based storage. Based on this evaluation, I conclude that a modified Zebra approach provides a superior basis for scalable storage.

4.1.1. RAID

The serverless storage system exploits RAID-style disk striping to provide high performance disk storage that is highly available. As Figure 4-1 illustrates, a RAID partitions a *stripe* of data into N-1 data blocks and a parity block — the exclusive-OR of the corresponding bits of the data blocks; it then stores each data and parity block on a different disk. The parallelism of a RAID's multiple disks provides high bandwidth, while its parity storage provides fault tolerance — it can reconstruct the contents of a failed disk by taking the exclusive-OR of the remaining data blocks and the parity block. A generalization of this approach can withstand multiple-disk failures by using multiple-parity disks for each stripe [Blaum et al., 1994].

RAIDs, however, suffer from two limitations. First, the overhead of parity management can hurt performance for small writes; if the system does not simultaneously overwrite all N-1 data blocks of a stripe, it must first read the old parity and the old data from the disks to compute the new parity. Each small write thus requires four disk accesses: two to read the old data and old par-

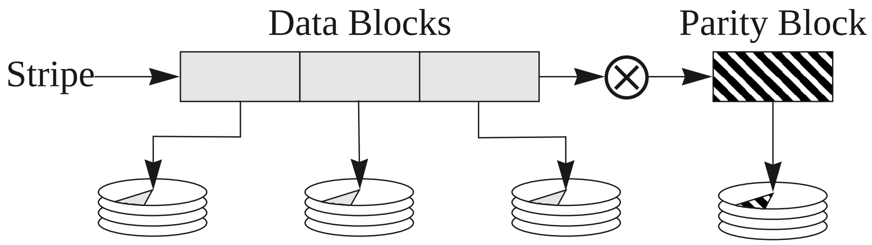


FIGURE 4-1. **RAID striping with parity.** The system stores each data stripe's block on a separate disk and stores the stripe's parity block on one additional disk.

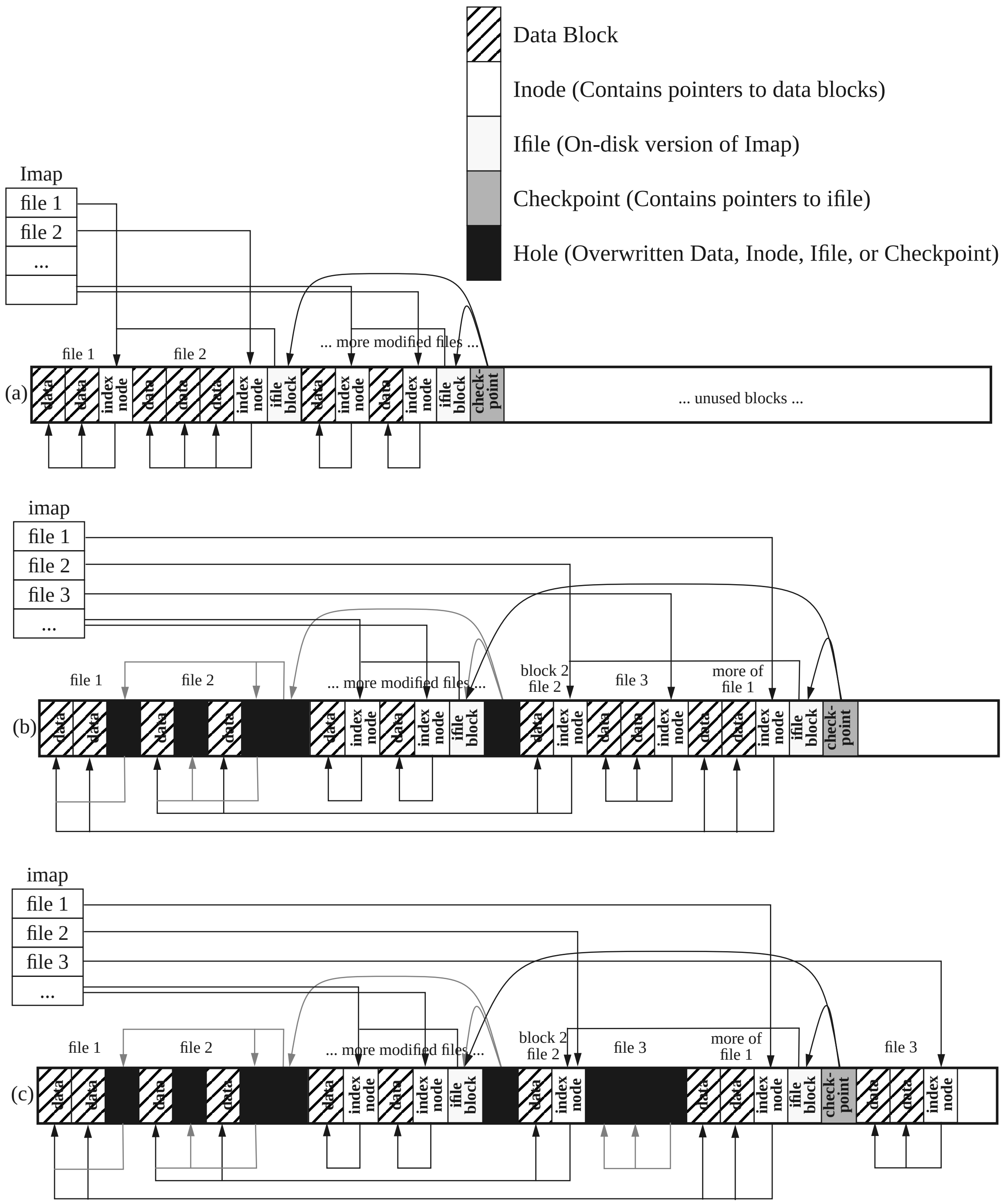
ity and two to write the new data and new parity. Unfortunately, small writes are common in many environments [Baker et al., 1991], and larger caches increase the percentage of writes in disk workload mixes over time. I would also expect cooperative caching — using workstation memory as a global cache as described in Chapter 3 — to further this workload trend by reducing the number of reads that access a disk. A second drawback of commercially-available, hardware RAID systems is that they are 2.5 to 10 times more expensive per megabyte than non-RAID commodity disks [Myllymaki, 1994]. RAIDs command this cost premium because they add special-purpose hardware to compute parity, because they have low sales volumes over which to amortize the added development costs to build this hardware, and because they are marketed at the high-end server market rather than the more cost-sensitive desktop market.

4.1.2. LFS

Distributed disk striping incorporates LFS because doing so addresses the problems of small writes in RAID systems while providing a flexible index to locate data for reads. LFS offers another advantage of particular value for scalable, distributed systems: it simplifies recovery compared to traditional disk organizations. However, LFS's log cleaner potentially limits the throughput of log-structured storage.

LFS provides high-performance writes by buffering many of them in memory and then committing them to disk in large, contiguous, fixed-sized groups called *log segments*. Batching writes into segments makes disk writes efficient by amortizing a single-disk seek and rotational latency over all of the writes in a segment. Thus, by relying on large writes, LFS exploits the technology trend whereby disk bandwidths are improving more rapidly than seek times or rotational latencies (see Table 2-1 on page 8.) LFS's log segments also address the RAID small-write problem by eliminating small writes; when LFS is used with a RAID, each segment of the log spans a RAID stripe and is committed as a unit.

Although log-based storage simplifies writes, it may complicate reads because a block could be located anywhere in the log, depending on when it was written. LFS's solution to this problem, illustrated in Figure 4-2, provides a general mechanism by which to handle location-independent data storage. LFS uses per-file *inodes* (index nodes), similar to those of the Fast File System (FFS) [McKusick et al., 1984], to store pointers to the system's data blocks. However, whereas FFS's inodes reside in fixed locations, LFS's inodes move to the end of the log each time they are modi-



Data Block
Inode (Contains pointers to data blocks)
Ifile (On-disk version of Imap)
Checkpoint (Contains pointers to ifile)
Hole (Overwritten Data, Inode, Ifile, or Checkpoint)
Imap
file 1
file 2
...
(a)
file 1
file 2
... more modified files ...
data
index node
ifile block
check-point
... unused blocks ...
imap
file 3
(b)
block 2 file 2
file 3
more of file 1
(c)

FIGURE 4-2. **A log-structured file system.** In figure (a), the log contains two new files, file 1 and file 2, as well as other modified files. The index node block following each file contains pointers to the file's data blocks, the imap contains pointers to the index nodes, and the ifile is an on-disk copy of the imap. The checkpoint contains pointers to the blocks of the ifile. In figure (b), the middle block of file 2 has been modified. A new version of it is added to the log, as well as a new version of its index node. Then file 3 is created, causing its blocks and metadata to be appended to the log. Next, file 1 has two more blocks appended to it. These two blocks and a new version of file 1's index node are appended to the log along with part of the ifile and a checkpoint. Finally, in figure (c), file 3 is overwritten; although this has occurred after the most recent checkpoint, if a crash occurred, the modification could be recovered during roll-forward from the previous checkpoint. When a data block, index node, or checkpoint is overwritten, the old version becomes obsolete, which creates a hole in the log that must be garbage collected later.

fied. When LFS writes a file's data block, moving it to the end of the log, the file system updates the file's inode so that it points to the new location of the data block; it then writes the modified inode to the end of the log as well. LFS locates the mobile inodes by adding a level of indirection, called an *imap*; in the next chapter, I will describe how to modify LFS's imap to distribute disk metadata across multiple managers. LFS stores the imap, containing the current log pointers to the system's inodes, in memory and periodically writes modified portions of the imap to disk in a file called the *ifile*. To support recovery, LFS occasionally writes *checkpoints* to the disk; the checkpoints contain pointers to the blocks of the ifile.

Checkpoints form a basis for LFS's efficient crash-recovery procedure because they allow LFS to recover by reading just the last few segments it wrote. After a crash, LFS reads backwards from the end of the log until it reaches the last checkpoint, and then it uses the checkpoint to read the ifile and recover the imap. To find the new location of inodes that were written since the last checkpoint, the system then *rolls forward*, reading the segments in the log after the checkpoint. When recovery is complete, the imap contains pointers to all of the system's inodes, and the inodes contain pointers to all of the data blocks. In contrast to LFS's efficient recovery procedure, traditional systems like FFS must scan the entire disk to ensure consistency after a crash because they have no way to isolate which areas of the disk were being modified at the time of the crash.

Another important aspect of LFS is its *log cleaner,* which creates free disk space for new log segments using a form of generational garbage collection [Lieberman and Hewitt, 1983]. When the system overwrites a block, it adds the new version of the block to the newest log segment, creating a "hole" in the segment where the data used to reside. The cleaner then coalesces old, partially-empty segments into a smaller number of full segments to create contiguous space in which to store new segments.

The overhead associated with log cleaning is the primary drawback of LFS. Although Rosenblum's original measurements found relatively low cleaner overheads, even a small overhead can make the cleaner a bottleneck in a distributed environment. Further, some workloads, such as transaction processing, incur larger cleaning overheads [Seltzer et al., 1993, Seltzer et al., 1995].

4.1.3. Zebra

Zebra provides a way to combine LFS and RAID so that both work well in a distributed environment: LFS's large writes make Zebra's writes to the network RAID efficient; Zebra's implementation of a software RAID on commodity hardware (workstation, disks, and networks) addresses RAID's cost disadvantage; and the reliability of both LFS and RAID allows Zebra to safely distribute data over the network.

LFS's solution to the small write problem is particularly important for Zebra's network striping because reading old data to recalculate RAID parity requires network communication in a distributed system. As Figure 4-3 illustrates, each Zebra client coalesces its writes into a private *per-client log*. It then commits the log to the disks using fixed-sized *log segments*, each made up of several *log fragments* that it sends to different storage server disks over the LAN. Log-based striping allows clients to calculate parity fragments efficiently using entirely local operations, and then to store parity fragments on an additional storage server to provide high data availability.

Zebra's log-structured architecture simplifies its failure recovery significantly because, like LFS, Zebra provides efficient recovery using checkpoint and roll forward. To roll the log forward, Zebra relies on *deltas* stored in the log. To allow the system to replay the modification during recovery, each delta describes a modification to a file system block, including the ID of the modified block and pointers to the old and new versions of the block. Deltas greatly simplify recovery by providing an atomic commit for actions that modify distributed state: each delta encapsulates a set of changes to a file system's state that must occur as a unit. Deltas also allow the system to chronologically order modifications to the same data by different clients.

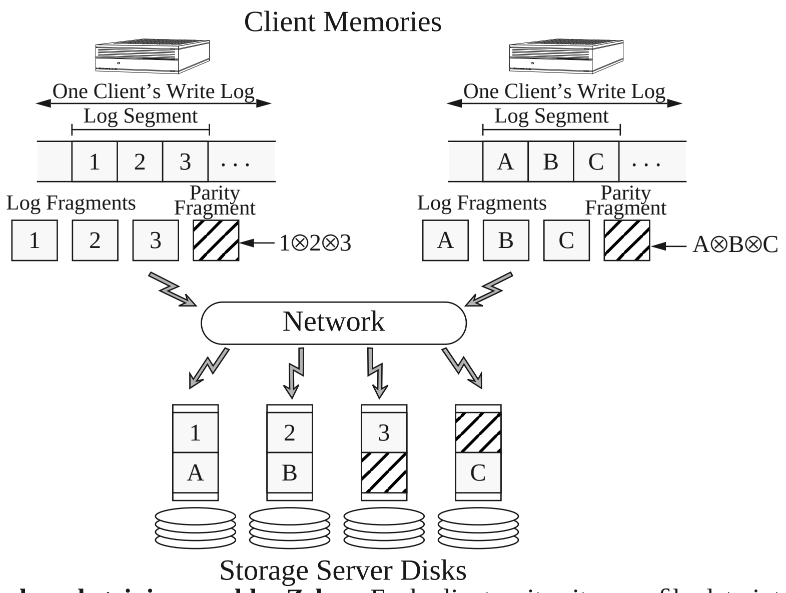


FIGURE 4-3. **Log-based striping used by Zebra.** Each client writes its new file data into a single append-only log and stripes this log across the storage servers. As a result, clients compute parity for segments, not for individual files.

4.1.4. Limits to Zebra's Scalability

Although Zebra points the way towards environments without servers, several factors limit Zebra's scalability. First, Zebra stripes all log segments over all storage servers. This restriction limits both performance and availability; Section 4.2 describes how to use *stripe groups* to address this problem. Second, Zebra, like LFS, relies on a single cleaner to create contiguous free disk space for new segments; as with LFS, a single cleaner limits Zebra's write throughput. Section 4.3 presents the design of a distributed cleaner that avoids this limit. Finally, a single *file manager* tracks where clients store data blocks in Zebra's log; the manager also handles all cache consistency operations. Chapter 5 describes how to adapt LFS's imap to distribute management duties across multiple machines and how to scale Zebra's recovery approach to handle this distributed management.

4.1.5. Alternatives to Log-based Striping

The earlier parts of this section suggested that log-based, redundant striping has two primary advantages over traditional striped-disk layout strategies that overwrite data in place. First, log-based striping provides superior small-write performance by coalescing writes; second, log-based striping simplifies recovery by restricting modifications to the end of the logs. Other approaches to solving these problems are possible; for instance, mirroring writes (RAID level 1) avoids the small-write problem encountered by parity striping (RAID level 5) because it duplicates data rather than calculate parity [Patterson et al., 1988]. Also, journaling file systems [Hagmann, 1987, Kazar et al., 1990] maintain some of the recovery advantages of LFS file systems while still overwriting data in place; journaling techniques might be extended to enable recovery for distributed striping.

4.1.5.1. RAID Level 1 Mirroring

A RAID level 1 storage organization *mirrors* duplicate data to two disks rather than computing parity over a larger numbers of disks. This approach avoids RAID level 5's small-write problem because it never needs to read the old data or the old parity to calculate the new parity. A RAID level 1 can also improve "degraded" read performance (when a disk fails). However, these improvements come at the price of increased storage cost. Thus, as Table 4-1 indicates, the cost/performance ratio of RAID level 1 is worse than an LFS-based RAID level 5 for writes, equal for standard reads, and equal for large reads when a disk has failed. Although RAID level 1 provides an advantage for small reads when a disk has failed [Hsiao and DeWitt, 1989, Lee, 1995], for many systems I would expect cooperative caching to satisfy most small-read requests, making small-read disk bandwidth under failure a relatively unimportant metric. Thus, the approach taken by RAID level 5 will usually be superior to a RAID level 1 approach. For systems that depend on small reads and that require real-time guarantees or for which caching is ineffective, a RAID level 1 approach may be worth considering.

4.1.5.2. AutoRAID

The primary drawback of a RAID level 1 system is the overhead, both in terms of capacity and bandwidth, to duplicate all data. AutoRAID [Wilkes et al., 1995] attempts to retain the small-write performance of RAID level 1 while reducing its space overhead. AutoRAID duplicates data that are actively being written using RAID level 1, but it automatically migrates less-active data to RAID level 5 to improve the system's storage capacity.

For systems with a high degree of write locality, AutoRAID can reduce storage overhead to approximate that of RAID level 5 while retaining the small-write performance of RAID level 1. Also, it retains the small-read failure-mode advantage of RAID level 1 for the subset of data that is

	RAID Level 1 2N Disks		**AutoRAID (Best Case) N + (N/G) Disks**		**RAID Level 5 (LFS) N+(N/G) Disks**		
	Performance	**Performance/ Cost**	**Performance**	**Performance/ Cost**	**Performance**	**Performance/ Cost**	**Advantage**
Capacity	N	1/2	N	G/(G+1)	N	G/(G+1)	Auto/RAID 5
Large-Write Bandwidth	N	1/2	(N+N/G)/2	1/2	N	G/(G+1)	RAID 5
Small-Write Bandwidth	εN	ε/2	ε(N+N/G)/2	ε/2	N*	G/(G+1)*	RAID 5*
Large-Read Bandwidth	2N	1	N+N/G	1	N + N/G	1	=
Small-Read Bandwidth	ε2N	ε	ε(N+ N/G)	ε	ε(N+ N/G)	ε	=
Large-Read BW (1 disk failed)	2N-1	1-1/(2N)	N+N/G-1	1-1/(N+N/G)	N	G/(G+1)	=
Small-Read BW (1 disk failed)	ε(2N-1)	ε(1-1/(2N))	ε(N+N/G-1)	ε(1-1/(N+N/G))	εN/2	ε(G/(G+1))/2	RAID 1/Auto

TABLE 4-1. **Performance and cost/performance.** This table indicates the relative performance and cost/performance of RAID level 1, AutoRAID, and RAID level 5 systems. This figure compares systems of constant capacity (each of the systems can store N disks worth of data plus redundant data.) The RAID level 1 system duplicates data while the LFS-based RAID level 5 system stores one parity fragment for each stripe group of G data fragments. The comparison for AutoRAID assumes the best case: that most of the system's data are stored in RAID level 5 to conserve space while most writes go to the RAID level 1 portion of the storage and that most small reads with one failed disk access RAID level 1 data. For small reads and writes, ε represents the efficiency of small accesses to disk: the small-access bandwidth (including seek and rotation overheads) divided by the large-access bandwidth. The asterisk (*) indicates that the results for the RAID Level 5 system assume that it is running LFS; without LFS, each small write in the RAID level 5 system requires four disk accesses, reducing the performance to N/4, the performance/cost to ε(1-1/G)/4, and giving the advantage to RAID level 1 and AutoRAID.

mirrored. However, because AutoRAID duplicates all writes, its total write bandwidth is less than that of a RAID level 5. Also note that AutoRAID is currently implemented as a traditional, centralized RAID controller. Adapting its data structures to a distributed system would require significant effort.

4.1.5.3. Journaling File Systems

Just as mirroring provides an alternative to RAID level 5 striping to provide data redundancy, journaling file systems [Hagmann, 1987, Kazar et al., 1990] provide an alternative to LFS deltas for metadata recovery. Journaling file systems maintain the update-in-place strategy used by traditional file systems, but they add an auxiliary data structure, called a journal, for logging changes before they are committed to the primary disk data structures. Before beginning to modify the file system's main data structures, the system atomically logs a list the changes it is about to make into the journal. If an operation that modifies multiple disk blocks in place is interrupted midway by a machine crash, it may leave the on-disk data structures in an inconsistent state. During recovery, however, the system can put the file system back in to a consistent state by comparing the main file system state with the modifications indicated in the journal's log and then completing any changes that were interrupted by the crash.

A journaling file system could thus be combined with RAID level 1 or AutoRAID to provide a serverless storage system with acceptable small write performance, high availability, and crash recovery. The main advantage of such a design would be the elimination of the LFS cleaner. However, a number of research issues must be addressed to make such an approach viable. First, the system would have to provide a method to introduce location-independence so that data could be moved between disks to balance load or when disks are added to or removed from the system. Second, the system should provide a distributed mechanism to locate and manage free disk space in the distributed system. Third, the system should provide policies for distributing files and directories across the disks to exploit locality and balance load. Developing these layout policies is likely to require substantial research efforts.

4.2. Stripe Groups

When using large numbers of storage servers, rather than stripe each segment to every storage server as in Zebra, scalable systems should implement *stripe groups* as have been proposed for large RAIDs [Chen et al., 1994] to improve performance and availability. A stripe group consists of a subset of the system's storage servers, and clients write their segments across stripe groups rather than across all storage servers. This section expands on the motivation for stripe groups, and then describes how the serverless design implements them.

4.2.1. Motivation for Stripe Groups

Stripe groups are essential to support large numbers of storage servers for at least four reasons.

1. Stripe groups allow large, efficient writes to disk.

By allowing clients to distribute each log segment to a limited set of storage servers, stripe groups allow clients to write larger fragments for a given segment size. By favoring large writes to each disk, stripe groups leverage the disk seek time and bandwidth technology trends that favor large disk accesses over small ones.

The alternative approach for providing large disk writes to many storage servers — increasing the segment size — is less desirable. First, this approach would increase the memory demands for file systems with many storage servers to allow clients to buffer these large segments. Further, because file systems periodically force data to disk using `fsync()` calls, clients would seldom fill larger segments before writing them to disk. For instance, Baker et al. [Baker et al., 1992] found that for eight production file systems, only 3% to 35% of all 512 KB segments were full when written to disk. Partial segment writes reduce the efficiency of writes by reducing the effective fragment size and by requiring expensive parity updates when segments are later filled.

2. Stripe groups reduce cleaning cost.

By limiting segment size, stripe groups make cleaning more efficient. This efficiency arises because when cleaners extract live data from a segment, they can skip completely empty segments, but they typically must read partially-full segments in their entirety. Large segments linger in the partially-full state longer than small segments, significantly increasing cleaning costs.

Table 4-2 illustrates this effect for a simulation using the Auspex trace described in Section 3.3.4.2 on page 56. Section 4.4, later in this chapter, describes my methodology for simulating cleaning.

As the table indicates, increasing the segment size from 64 KB to 512 KB increases the write cost by 56% for the Auspex trace because with larger segments the cleaner must read 47% of those segments cleaned while with smaller sizes only 7% of the segments cleaned must be read. Since the amount of data is the same no matter what the segment size, small segments mean less data need to be cleaned.

3. Stripe groups match network bandwidth to disk bandwidth.

Stripe groups match the aggregate bandwidth of the groups' disks to the network bandwidth of a client, using both resources efficiently. Thus, while one client writes to or reads form one stripe group at its full network bandwidth, another client can access a different group, also at the client's full network bandwidth.

4. Stripe groups improve availability.

Finally, stripe groups improve availability. Because each group stores its own parity, the system can survive multiple server failures if they happen to strike different groups, which is the most likely case in a large system with random failures. For instance, Figure 4-4 shows the amount of time that some data in a system are unavailable under organizations with zero, one, or two parity fragments per stripe as well as with or without stripe groups. Striping with a single parity fragment but without groups provides better availability than a single server system as long as there are fewer than 40 storage servers in the striped system. For larger systems, however, downtime increases rapidly. Using two parity fragments without stripe groups or one parity fragment with stripe groups allows the system to scale to approximately 150 storage servers. Two parity fragments per stripe group gives effectively unlimited scalability, suffering less than two minutes of

Segment Size	**Segments Written by File System**	**Segments Cleaned**	**Segments Read by Cleaner**	**Segments Written by Cleaner**	**Empty Segments Cleaned**	**Write Cost**
64 KB	61370	63357	4383	960	58974	1.087
512 KB	7671	8844	4166	1142	4678	1.693

TABLE 4-2. **Comparison of cleaning cost as segment size is varied.** Smaller segments are less expensive to clean because they are more likely to become completely empty. This simulation used the seven-day Auspex trace, and assumed that the disk was 80% full. The write cost [Rosenblum and Ousterhout, 1992] compares the overhead from the cleaner's disk accesses to the amount of data written by the system. A write cost of 1.0 is perfect, indicating no cleaner overhead, while a write cost of 2.0, for instance, indicates that the cleaner is responsible for as much disk activity as file system writes.

downtime per month with 1000 storage servers and less than 14 minutes per month with 10,000 storage servers.

4.2.2. Implementing Stripe Groups

The key data structure for implementing stripe groups is the stripe group map. As Figure 4-5 illustrates, this map provides a translation between a group identifier and the list of servers storing that group's fragments. When a client writes a segment, it selects one group from the map and distributes the segment and its parity to the members of the group indicated by the map. To read a segment, a client extracts the stripe group ID field from the segment ID and uses the stripe group ID to identify the machines storing the data. If a storage server is down when a client tries to read data from it, the client uses the stripe group map to identify the other storage servers in the group to reconstruct the data using parity.

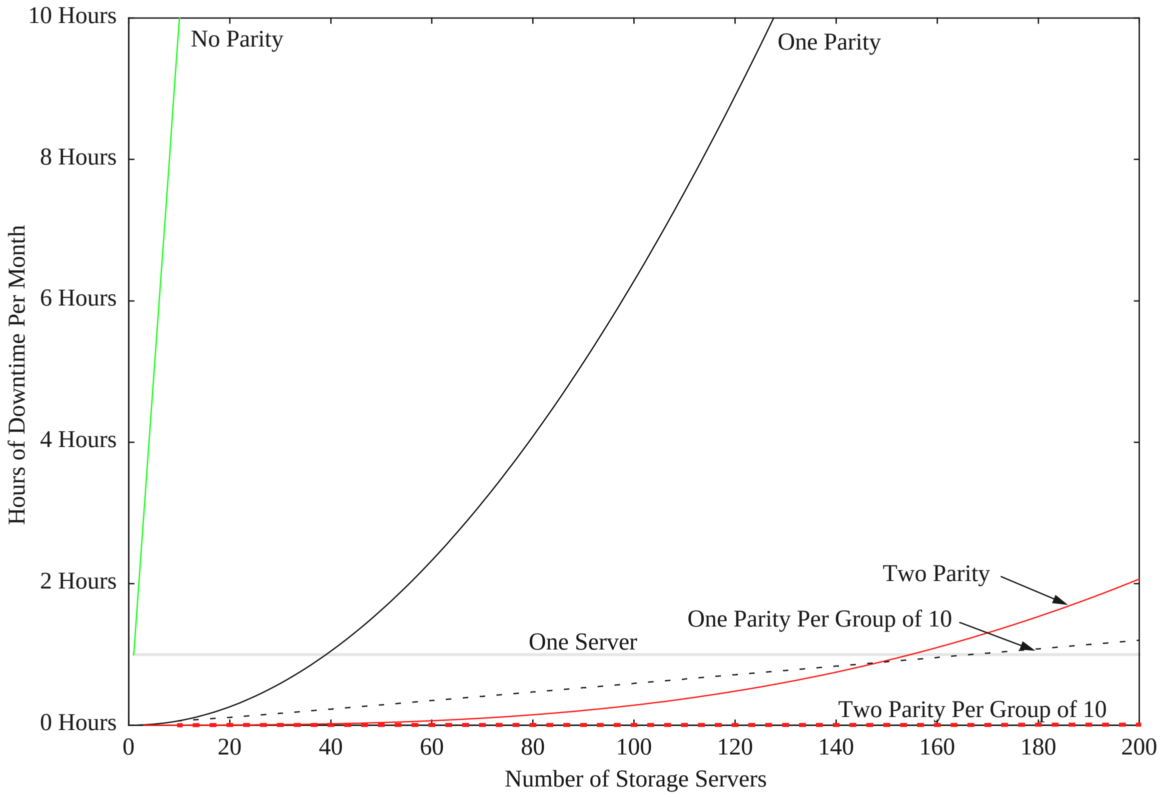


FIGURE 4-4. **Hours of downtime per month under different parity organizations.** This graph assumes that each storage server fails randomly for an average of one hour per month, and it assumes that the system is down when any data block is unavailable. The *No Parity* line shows the system's availability, assuming data are stored without redundancy; the *One Parity* and *Two Parity* lines show the system's availability assuming no stripe groups and one or two parity fragments protecting all of the system's disks. The *One Parity Per Group of 10* and *Two Parity Per Group of 10* lines assume that the system uses stripe groups of 10 machines with one or two parity fragments per group. For reference, the *One Server* line shows the availability of a single server under these assumptions.

The system replicates the stripe group map globally among clients so that any client can contact the storage servers that belong to any given stripe group. Global replication is reasonable because the stripe group map is small and because it seldom changes; the map contains a list of stripe groups and the groups' members, and it changes only when a machine enters or leaves the system — for instance, if a machine crashes.

Each storage server typically belongs to exactly one stripe group, but to handle map reconfiguration, storage servers can belong to multiple groups: one *current* stripe group and zero or more *obsolete* stripe groups. The current stripe groups listed in the map represent the groups of storage servers for the current configuration of machines; clients write only to current stripe groups. Obsolete stripe groups indicate mappings that existed at some time in the past, before the last reconfiguration of the stripe group map; clients read from both current and obsolete stripe groups. Leaving obsolete entries in the map allows clients to read data previously written to the obsolete groups without first transferring the live data from obsolete groups to current ones. Over time, the cleaner will move data from obsolete to current groups [Hartman and Ousterhout, 1995]; when the cleaner removes the last block of live data from an obsolete group, the system deletes the obsolete entry from the stripe group map to keep the map small.

When one or more storage servers enter or leave the system, the system must regenerate the stripe group map. To do so, it uses a global-consensus algorithm to identify storage servers that are active, to assign these servers to current stripe groups, and to distribute the resulting stripe group

Stripe Group Map

Stripe Group ID	**SS's In Group**	**Current/ Obsolete**
98	1,2,3,4	Current
99	5,6,7,8	Current
96	7,8,0,1	Obsolete
97	2,3,4,5,6	Obsolete

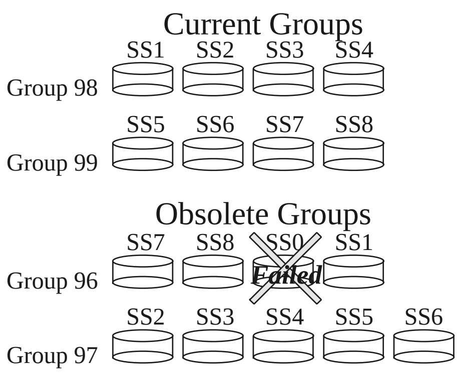


FIGURE 4-5. **A stripe group map.** The table on the left shows the contents of a stripe group map and the figure on the right shows the resulting logical relationships among disks. The system can write or read either of the current stripe groups, but it can only read the obsolete stripe groups. A storage server in one of the obsolete groups, Group 96, has failed. If a client were to read a block stored on the third storage server of Group 96, it would find that storage server 0 (SS0) is down, and it would then read the corresponding blocks from the other storage servers in the group to reconstruct the lost data from parity.

map of current groups; Section 4.5.2 discusses this process. To generate a map of obsolete groups, each storage server uses a local checkpoint to assemble a list of groups for which it stores fragments. The system combines the storage servers' lists and distributes the result to complete the stripe group map.

4.3. Cleaning

When a log-structured storage system appends data to its logs, it invalidates blocks in old segments, leaving "holes" that contain no live data. LFS systems use a *log cleaner* to coalesce live data from old segments into a smaller number of new ones, creating completely empty segments that can be used for full segment writes. Since the cleaner must create empty segments at least as quickly as the system writes new ones, a single, sequential cleaner would act as a bottleneck in a distributed system. A scalable architecture, therefore, must provide for a distributed cleaner.

An LFS cleaner, whether centralized or distributed, has three main tasks. First, the system monitors old segments' utilization status — how many holes they contain and how recently these holes appeared — to make wise decisions about which segments to clean [Rosenblum and Ousterhout, 1992]. Second, the system monitors the number of free segments and the level of system activity so that it can begin cleaning when space is needed or when the system is idle. Third, when the system does begin to clean, it proceeds in two steps: the cleaner examines the utilization information to select which segments to clean, and then, for segments that are not completely empty, it reads the live blocks from the old segments and writes those blocks to new, compact ones.

The rest of this section describes how to distribute cleaning across multiple machines. It first describes how to distribute the task of tracking segment utilization. It then outlines how the system monitors free space and system activity to activate the cleaners. Finally, it describes how the system distributes the tasks of selecting segments and moving live data from old segments to new ones. This section describes the mechanisms necessary to support distributed cleaning, and it explains why these mechanisms should provide good performance through locality. However, future work is needed to identify specific policies for activating cleaners.

4.3.1. Monitoring Segment Utilization

The distributed cleaner assigns the burden of maintaining each segment's utilization to the client that wrote that segment. The system stores this information in standard files, called *s-files*, to allow any machine to access any segment's status. Distributing bookkeeping to the writers of each segment provides parallelism and locality, and because clients seldom write-share data [Baker et al., 1991, Kistler and Satyanarayanan, 1992, Blaze, 1993], a client's writes usually affect the utilization status of local segments only. At the same time, s-files provide a globally accessible view of segment use by means of the file sharing mechanisms already in place in the system. The rest of this section examines the locality provided by distributing bookkeeping, and then it describes s-files in more detail.

4.3.1.1. Distributed Bookkeeping

To examine how well this distributed bookkeeping policy reduces the overhead of maintaining utilization information, I simulate its behavior on Auspex trace described in Section 3.3.4.2 on page 56. Because this simulation only considers blocks that are written multiple times, caching is not an issue, so I gather statistics for the full seven-day trace rather than using some of that time to warm caches. These simulations suggest that distributed cleaning reduces the total load to monitor segment utilization by over an order of magnitude, assuming that the load for network protocol processing dominates data structure manipulation. This approach further benefits scalability by distributing this reduced load across multiple machines.

Figure 4-6 shows the results of the simulation in more detail. The bars summarize the network communication necessary to monitor the segments' state under three policies: Centralized Pessimistic, Centralized Optimistic, and Distributed. Under the Centralized Pessimistic policy, clients notify a centralized, remote cleaner every time they modify an existing block. The Centralized Optimistic policy also uses a cleaner that is remote from the clients, but to account for the effect of the 30-second write delay buffer used by many systems, clients do not have to send messages when they modify blocks that were recently written. Because `fsyncs()` are not visible in the Auspex trace, the simulator optimistically assumes that all writes are buffered for 30 seconds, and it does not charge the Optimistic policy for local overwrites of data less than 30 seconds old. Finally, under the Distributed policy, each client tracks the status of blocks that it writes, so it needs no network messages when modifying a block for which it was the last writer.

During the seven days of the trace, roughly one million blocks were written by clients and then later overwritten or deleted. 33% of these were modified within 30 seconds by the same client and therefore require no network communication under the optimistic assumption. Nevertheless, the Distributed scheme reduces communication by a factor of eighteen for this workload compared to even the Centralized Optimistic policy.

4.3.1.2. S-Files

Although the system distributes the task of tracking segment utilization to the clients that wrote each segment, it allows any cleaner to clean segments written any client. This way, if a client is busy while another machine is idle, the second machine may clean segments written by the first. To allow universal access to utilization information, clients periodically write their bookkeeping information to *s-files*, standard files that may be read by any machine in the system.

For locality, each s-file contains utilization information for segments written by one client to one stripe group: clients write their s-files into per-client directories, and they write separate s-files in their directories for segments stored to different stripe groups.

The s-files not only localize information for segments written by each client to each group, they also provide a method by which different cleaners can concurrently work on segments stored

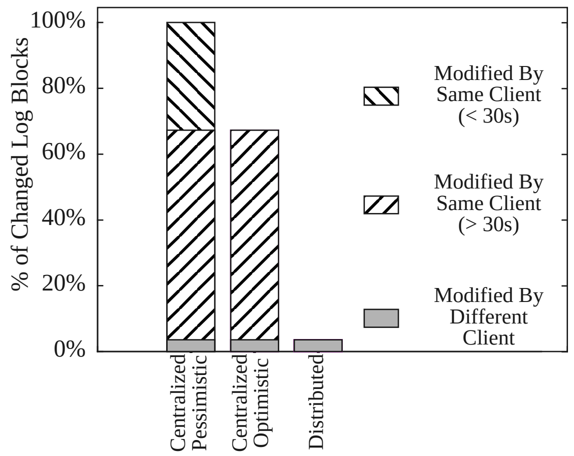


FIGURE 4-6. **Simulated network communication between clients and cleaner.** The distributed algorithm exploits locality to reduce the cleaner workload. Each bar shows the fraction of all blocks modified or deleted in the Auspex trace, based on the time and client that modified the block. Blocks can be modified by a client other than the original data writer, by the same client within 30 seconds of the previous write, or by the same client after more than 30 seconds have passed. The *Centralized Pessimistic* policy assumes every modification requires network traffic. The *Centralized Optimistic* scheme avoids network communication when the same client modifies a block it wrote within the previous 30 seconds, while the *Distributed* scheme avoids communication whenever a block is modified by its previous writer.

in the same stripe group: the system assigns each cleaner to a distinct set of s-files for that stripe group.

4.3.2. When to Activate Cleaners

The system cleans *on demand*, when the number of free segments in a stripe group of storage servers runs low, or *in the background*, when cleaning can take advantage of idle resources to reorganize data without delaying other requests to the file system [Blackwell et al., 1995]. Stripe groups can independently (and thus scalably) determine when to start cleaning.

To determine when to clean on demand, each storage server maintains a local count of free segments. If this count falls below a low-water mark, that storage server can initiate cleaning for the entire stripe group by informing its other members of the decision (so that they do not simultaneously initiate cleaning) and then activating cleaners by assigning to each a subset of the s-files from the stripe group.

Similarly, if a storage server is idle, it can initiate background cleaning by first asking the other members of the group if they are idle and then locating an idle cleaner. If other members are busy or if no idle cleaner can be found, the storage server aborts background cleaning.

A key question during this process is: which machines should clean? One simple policy assigns each client to clean the segments that it writes. An attractive alternative is to assign cleaning responsibilities to idle machines. Detailed policies for assigning s-files to cleaners should be investigated in the future.

4.3.3. Distributing Cleaning

The system activates a cleaner by assigning it a subset of s-files to examine and clean. Given this subset of segments, each of the distributed cleaners proceeds almost exactly as it would if it were the only cleaner in the system. The cleaning decisions, however, differ slightly from those of a global cleaner. In particular, because each member of the distributed cleaner examines only a subset of the segments, the ones that they choose to clean may not be globally optimal; by the same token, this approach allows the cleaners to make their decisions without a central bottleneck.

To clean the segments associated with a subset of s-files, the cleaner first reads the segment utilization state from those s-files. The second step is to choose segments to clean based on this utilization information. Cleaners choose segments based on a cost/benefit analysis that considers their utilizations and modification histories [Rosenblum and Ousterhout, 1992]. Finally, the cleaner reads the live data from the segments it is cleaning and writes that data to new segments, marking the old segments as clean.

4.3.4. Coordinating Cleaners

Like BSD LFS and Zebra centralized cleaners, a distributed cleaner uses optimistic concurrency control to resolve conflicts between cleaner updates and normal file system writes. Cleaners do not lock files that are being cleaned, nor do they invoke cache consistency actions. Instead, cleaners just copy the blocks from their old segments to their new ones, optimistically assuming that the blocks are not in the process of being updated somewhere else. If there is a conflict because a client is writing a block as it is cleaned, the manager will ensure that the client's update takes precedence over that of the cleaner [Hartman and Ousterhout, 1995]. Although the algorithm described here that distributes cleaning responsibilities never simultaneously asks multiple cleaners to clean the same segment, the same mechanism could be used to allow less strict (e.g. probabilistic) divisions of labor by optimistically resolving conflicts between cleaners.

4.4. Cleaning and Cooperative Caching

This section describes the synergy between cleaning and cooperative caching. The large cache provided by cooperative caching in a serverless file system significantly reduces cleaning costs by allowing a cleaner to read data from a cooperative cache rather than from disk.

Cooperative caching allows a serverless system to reduce the cost of log cleaning by reducing the need to read live data for segments being cleaned from disk. Reading live data blocks from disk is more expensive than reading them from cache for three reasons. First, the overhead for setting up a disk access is higher than for accessing cache memory. Second, disks, themselves, are much slower than main memory (see Table 2-1 on page 8.) Finally, caches support efficient random access, so the cleaner pays to read only the live data that it needs. In contrast, when reading a segment's live data from disk, the cleaner seeks to the segment, and then must often wait for the entire segment to rotate under the disk head, even if there are only a few live blocks scattered throughout the segment. Because cleaners typically clean segments that are mostly empty, this case happens frequently.

To quantify caching's effect on cleaning costs, I modify Rosenblum's cleaning simulator [Rosenblum and Ousterhout, 1992] to accept input from a trace file and to track cache information. For the sake of simplicity, I simulate a single, sequential cleaner. Each client appends data to its own, private 512 KB segment until the segment fills, but clients never overwrite data in pace, even when overwriting data from the current segment; this simplification increases the write cost by requiring the system to clean more data.

The simulation assumes that each client has 16 MB of cache, accessed and coordinated via cooperative caching using the N-Chance algorithm. The caching cleaner is allowed to read a segment from memory rather than disk only if all live blocks are cached; if even one live block is not cached, the cleaner reads the entire segment from disk just as the standard cleaner normally does. Both the caching and non-caching cleaners skip completely empty segments entirely.

For input, I use the seven-day Auspex trace described in Section 3.3.4.2 on page 56. I warm the simulator's state in two steps. First, to create a realistic layout of data in segments, the simulator runs through the entire trace with reads and caching turned off, but with writes and cleaning turned on. This pass gives all data an initial storage location, a pessimistic assumption since some writes in the trace would normally create new blocks rather than overwriting old ones. This assumption increases the cleaning cost because it creates more holes that must be cleaned than would occur in reality. The first pass also allows the rest of the simulation to run with an approximately stable disk utilization. In the second pass, I enable reads and caching (in addition to writes and cleaning), but, to ensure that the read caches were warm, I do not begin to gather statistics until the last half of the trace.

Figure 4-7 compares the *write cost* for two cleaning policies, one that always reads live segments from disk and a second that may read data from the client cooperative caches. I use Rosenblum's definition of write cost [Rosenblum and Ousterhout, 1992]:

$$WriteCost = (BytesWritten_{Data} + BytesRead_{Cleaner} + BytesWritten_{Cleaner}) / (BytesWritten_{Data})$$

In other words, the write cost expresses the average cost of writing new data as the total amount of new data written plus the amount of data read and written by the cleaner, divided by the amount of new data written to disk. If there were no cleaning overhead, the (ideal) write cost would be 1.0.

When the disk is 90% full, cleaning from the cache improves the write cost by 20% and reduces the cleaning overhead by 40% for this workload. At lower disk utilizations, the improvement is lower because longer segment lifetimes allow the segments to empty more completely before they are cleaned. When the disk is 60% full, caching reduces the write cost by 7% (a 40% improvement in the cleaning cost portion of the write cost).

Finally, note that even with the pessimistic assumptions about cleaning noted above, the write cost for this workload is low, supporting the decision to use LFS for file storage. The caching cleaner's write cost is less than 2.0, even when disk utilization reaches 90%. In contrast, Rosenblum estimates FFS's write cost to be between 4.0 and 10.0 [Rosenblum and Ousterhout, 1992]. This comparison suggests that LFS reduces write overheads by at least a factor of two for this workload compared to standard file system layouts.

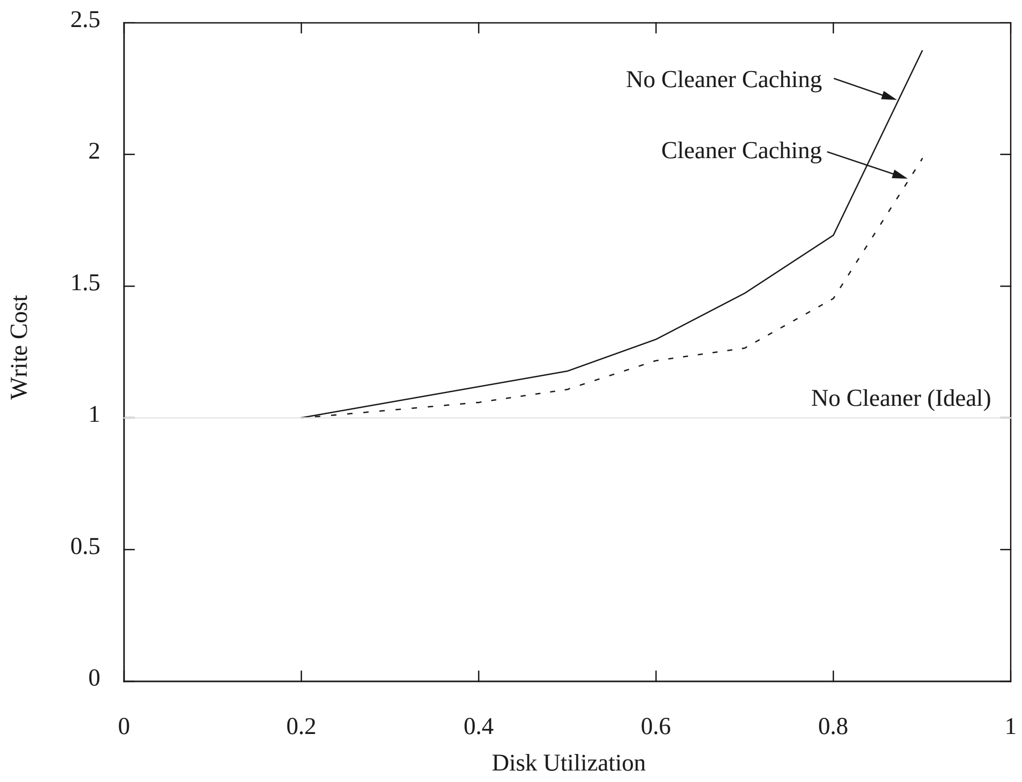


FIGURE 4-7. **Cooperative Caching and Cleaning.** The y-axis of this graph shows the write cost with and without the improvement of reading from the cache when cleaning. The x-axis indicates the peak disk utilization (the peak amount of live data divided by the total size of the disk) for the Auspex trace used for this simulation.

4.5. Availability

The storage system is a key building block for a serverless system’s reliability and availability. The rest of this section outlines how storage servers recover their local state after a crash, how they reconfigure the stripe group map in response to changing storage server status, how cleaners recover their state after a crash, and how storage servers generate data from parity to allow continued operation when a storage server fails. Section 5.4 on page 109 provides a more complete description of the serverless system’s recovery by discussing how the storage servers and managers interact to recover the system’s metadata.

4.5.1. Local Storage Server State

When a storage server boots, it first restores its internal data structures so that the rest of the system can access its on-disk logs. Each storage server maintains a local checkpoint that includes the mappings from logical fragment IDs to the fragments’ physical disk addresses, maps of local free disk space, and lists of the stripe groups for which the machine stores fragments. Each server also rolls forward from its local checkpoint to update its data to reflect fragments written after the checkpoint. During roll-forward, storage servers verify the checksums of any fragments that they wrote at about the time of the crash, discarding incomplete fragments. Each storage server recovers this information from data stored on its local disk, so this stage can proceed in parallel across all storage servers.

4.5.2. Stripe Group Map Reconfiguration

Whenever the number of storage servers in a system changes, either because a machine has been added, because a machine has failed, or because the system is booting, the system generates a new stripe group map and distributes it to all active storage servers, clients, and managers. Stripe group map generation proceeds in three steps.

In the first step, the storage servers use a consensus algorithm to identify all of the active storage servers and elect a leader from among those machines. The global consensus algorithm used can be one of several known algorithms [Ben-Or, 1990, Cristian et al., 1990, Cristian, 1991, Schroeder et al., 1991]. When a machine notices that another machine has left or wants to join the system, it initiates the consensus algorithm. When the algorithm completes, all of the machines will know the identities of all of the other machines, and they will agree on a machine to lead the next step.

Second, each storage server sends a list all of the stripe groups for which that storage server is currently storing fragments to the leader, and the leader assembles a full list of the stripe groups with live data. After recovery, the system will be able to read data from these potentially obsolete stripe groups, but it will not write data to them.

Third, the leader partitions the live storage servers into current stripe groups that the system will use for new writes after recovery, reusing as many old stripe groups as feasible. It combines the lists of current and obsolete stripe groups to form the complete stripe group map, and it distributes this stripe group map to all of the active storage servers.

4.5.3. Cleaner Recovery

The last piece of state that must be recovered for the storage subsystem is the cleaners' segment utilization information checkpointed in the s-files. Because the cleaners store this state in standard files, the same procedures that recover the rest of the systems standard files recover the s-files. Section 5.4 on page 109 describes these higher-level recover procedures.

4.5.4. Parity Reconstruction

Redundant data storage across storage servers allows the system's logs to be available even when one or more [Blaum et al., 1994] storage servers in a stripe group have crashed. When a client realizes that a storage server that it tries to access has failed, it looks at the stripe group map to find the list of storage servers in the same stripe group. The client then reads the data and parity fragments from those storage servers to reconstruct the fragment it was initially trying to read.

4.6. Related Work

Section 4.1 discussed several technologies that provide an important basis for the scalable, log-based, network striping described in this chapter. Redundant Arrays of Inexpensive Disks (RAID) [Patterson et al., 1988, Chen et al., 1994] demonstrate how to stripe data across multiple disks to get performance and availability. Log-structured File Systems (LFS) [Rosenblum and Ousterhout, 1992, Seltzer et al., 1993] provide a way to batch writes to improve performance and provide fast, simple recovery. Zebra [Hartman and Ousterhout, 1995] combines the ideas of RAID and LFS to provide efficient network striping. This section describes several other efforts to build decentralized storage systems and then describes several efforts to provide efficient writes in non-LFS systems.

4.6.1. Striped Disks

In contrast with Zebra's log-based striping, most efforts to distribute disk storage have used per-file striping in which each file is striped across multiple disks and parity is calculated on a per-file basis. This approach has two disadvantages compared to log-based striping. First, the update-in-place disk layout associated with per-file striping makes crash recovery difficult unless the system also supports journaling (see Section 4.1.5.3). Second, per-file striping with update-in-place makes parity calculation expensive for small files, small writes to large files, or concurrent writes to different parts of the same file.

As a result, many parallel disk file systems, such as CFS [Pierce, 1989], Bridge [Dibble and Scott, 1989], and Vesta [Corbett et al., 1993], distribute data over multiple storage servers, but do not attempt to provide availability across component failures. Such systems are typically used for parallel, scientific workloads where data availability is considered less important than maximum bandwidth for a given system cost.

Other parallel systems have implemented redundant data storage but restrict their workloads to large file accesses, where per-file striping is appropriate and where large file accesses reduce stress on their centralized manager architectures. For instance, Swift [Cabrera and Long, 1991] and SFS [LoVerso et al., 1993] provide redundant distributed data storage for parallel environments, and Tiger [Rashid, 1994] services multimedia workloads.

4.6.2. Delayed Writes

In addition to batching writes into large, contiguous segment writes, LFS exploits *delayed writes* to eliminate disk accesses for writes that are overwritten while still in the write buffer. Many other file systems, such as FFS [McKusick et al., 1984], Andrew [Howard et al., 1988], and Sprite [Nelson et al., 1988] use delayed writes to gain this benefit. Studies of such systems have shown that for office and engineering workloads, 35-50% of all bytes "die" — are overwritten or deleted — within 30 seconds [Srinivasan and Mogul, 1989, Baker et al., 1992, Ruemmler and Wilkes, 1993].

A disadvantage of delayed writes is that if a machine crashes, systems can lose data that has been written to the write buffer but not to disk. LFS limits the damage to the system by atomically committing groups of modifications to file system state when it writes each segment. In LFS, if a machine crashes before committing a segment in its write buffer, the modifications in the segment are lost, but the rest of the file system's data structures remain consistent. Echo [Birrell et al., 1993] and Quicksilver [Schmuck and Wyllie, 1991] also commit writes carefully to maintain consistency across crashes. In contrast, traditional file systems like FFS must scan all of the file system's on-disk data structures after a crash to fix any inconsistencies arising from partially-complete writes. As noted in Section 4.1.5.3, journaling supplements such systems with auxiliary logs to assist recovery [Hagmann, 1987, Kazar et al., 1990]. Other techniques attempt to avoid losing any of the write buffer's data when a machine crashes [Liskov et al., 1991, Baker et al., 1992, Baker, 1994]; these techniques could be applied to the serverless system's log-structured file system to further reduce the risks of delayed writes.

4.7. Conclusions

This chapter described how log based striping provides a solid basis for a scalable, distributed storage system. It presented several techniques to allow such a system to scale to large numbers of storage servers. In particular, stripe groups seem essential for providing availability and performance for large systems; the stripe group map abstraction provides a basis for implementing scalable stripe groups. Similarly, distributed cleaning and, when possible, cleaning from the cooperative cache prevent the cleaner from becoming a bottleneck. Finally, log-based, redundant striping combines the high availability of RAID with the simple, fast recovery of LFS to provide a solid basis for the rest of the serverless system's recovery procedures.

5 Distributed Management

A key design principle of the serverless file system is location independence: cooperative caching and network disk striping allow any block of data to be stored on any machine in the system, and they allow blocks' positions to change over time. As a result, tracking the locations of the system's data is a critical task. Furthermore, the location service, itself, must be dynamically distributed across the system's machines to provide scalable performance, to improve network locality, and to provide high availability. The serverless system's *distributed manager* provides this scalable, distributed location service.

Each manager machine in the system tracks the disk and cache storage locations of the blocks of some subset of the system's files. Using this location information, a manager can direct client requests to the correct storage locations and coordinate multiple clients' accesses to the same data.

The system distributes management across manager machines with a *manager map*, an array of machine identifiers that indicate which machines manage which portions of the file system. The system locates a file's manager by first hashing on the file's unique index number and then using the result as an index into the manager map. The manager map thus provides a level of indirection so that the management service, itself, is location independent; any manager can manage any file.

The distributed management design assumes a log-based, highly-available, distributed storage system as described in the previous chapter. In particular, the design distributes disk location information by distributing the LFS imap data structure. Additionally, the crash recovery design outlined here depends on two properties of the storage system. First, it requires that the storage system include a log to allow recovery based on checkpoint and roll forward. Second, it assumes that the storage system is redundant and highly available. If either of these assumptions were not met, building a reliable, distributed manager would be more difficult.

The rest of this chapter describes the distributed manager architecture in detail. Section 5.1 presents the distributed manager design and explains how the system fulfills the manager's two primary duties: tracking where data blocks are stored on disk and where data blocks are cached. Section 5.2 illustrates the operation of the system for reads and writes, and it explains how the system's cache consistency protocol works. Section 5.3 considers the performance impact of distributed management and, using simulation studies, examines different policies' impact on locality. Section 5.4 describes how crash recovery works, considering both how the system redistributes management duties when one or a few managers fail and how the system avoids bottlenecks when many machines must recover simultaneously. Section 5.5 discusses related work; although few file systems have implemented distributed management, several distributed shared memory computers have used related approaches to distribute their cache consistency state. Finally, Section 5.6 summarizes the chapter's main conclusions.

5.1. Distributed Management Design

The manager's primary task is to track the location of every block of data in the system, including copies stored in client caches and on storage server disks. Using this information, managers forward client read requests to storage locations that can satisfy them, and managers invalidate stale copies when clients overwrite or delete data.

A distributed manager partitions these tasks across multiple machines by assigning a portion of the index number space to each machine as Figure 5-1 illustrates. To locate a file's manager, a client hashes on the file's index number to generate a virtual manager number. It then uses the virtual manager number as an index into the manager map to determine which physical machine controls the file's cache consistency metadata and disk location metadata (the imap and cache of index nodes and indirect nodes.)

The manager map provides a level of indirection for the hash function, allowing the system to control the mapping from virtual managers to physical machines. For instance, if a manager crashes, the system can assign new physical machines to handle the affected virtual managers, or if a new machine joins the system, the system can assign that machine to manage some of the entries in the manager map. The manager map thus limits changes in management to isolated portions of the index number space, reducing service disruptions and preventing changes to one virtual manager mapping from interfering with the load distribution and locality efforts made for other mappings.

To support reconfiguration, the manager map should have at least an order of magnitude more entries than there are physical managers. This rule of thumb allows the system to balance load by assigning roughly equal portions of the index number space to each manager. Larger numbers of map entries allow the system to assign management on a more fine-grained basis and reduce load imbalance; however, larger tables increase the space overhead of storing the manager map.

The system globally replicates the manager map so that each manager and client has a local copy. This replication allows managers to know their responsibilities, and it allows clients to contact the correct manager directly given a file's index number. Even though the system distributes management over multiple machines, it requires no additional network latencies compared to a centralized manager; in fact, as Section 5.3 will show, distributed management can actually reduce average network latency per request by increasing locality. It is reasonable to globally replicate the manager map because (i) it is relatively small — even with hundreds of machines, the map with thousands of entries would be only tens of kilobytes in size — and (ii) because it changes infrequently — only to correct a load imbalance or when a machine enters or leaves the system.

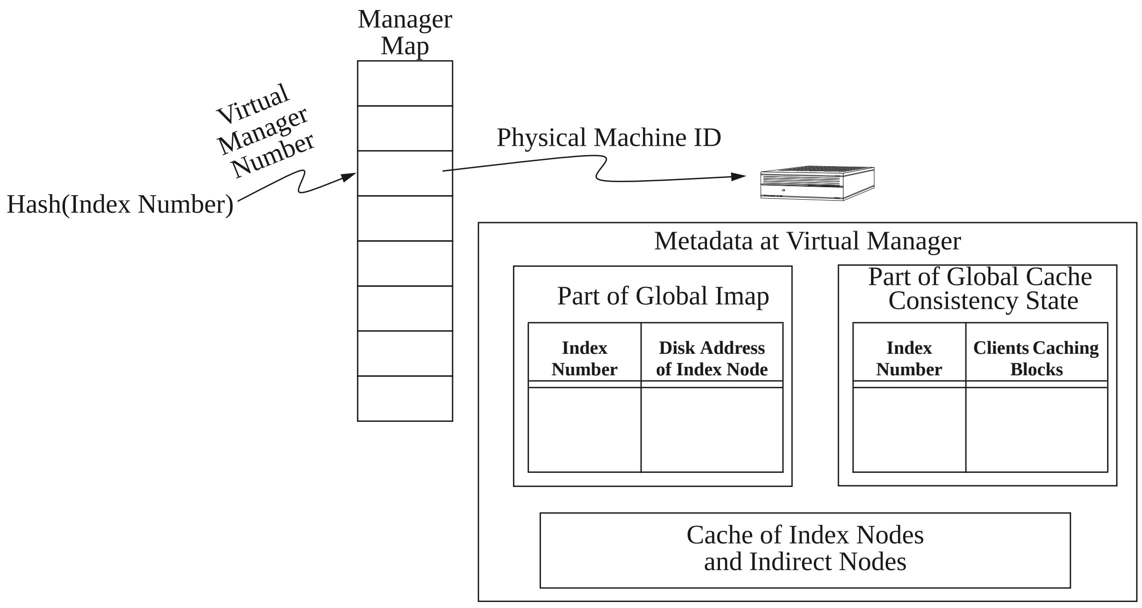


FIGURE 5-1. **The Manager Map.** The map specifies which physical machines manage which hash buckets of the index number space. The manager specified by the manager map controls the portion of the imap that lists the disk location of index nodes for the files managed by that virtual manager, and that manager keeps a cache of index nodes for those files. It also controls the cache consistency state (the list of all clients caching a file's blocks or with write ownership of the file's blocks) for those files.

5.1.1. Disk Storage Location Metadata

The system distributes the disk location metadata to managers by distributing the LFS imap. In LFS, the imap provides a mapping from each file's index number to the disk log address where the file's index node was last written. For a distributed manager, the imap contains the same translation, but rather than implement the imap as a single, centralized array, the system implements it as several, smaller arrays — one per virtual manager entry in the manager map. Each manager stores and controls the portion(s) of the imap indicated by the manager map, allowing that manager to read and write index nodes and indirect blocks for its files. The managers also cache recently accessed index nodes and indirect blocks.

As Figure 5-2 illustrates, the disk storage for each file can be thought of as a tree whose root is the imap entry for the file's index number and whose leaves are the data blocks. A file's imap entry contains the log address of the file's index node. xFS index nodes, like those of LFS and FFS, contain the disk addresses of the file's data blocks; for large files the index node can also contain log

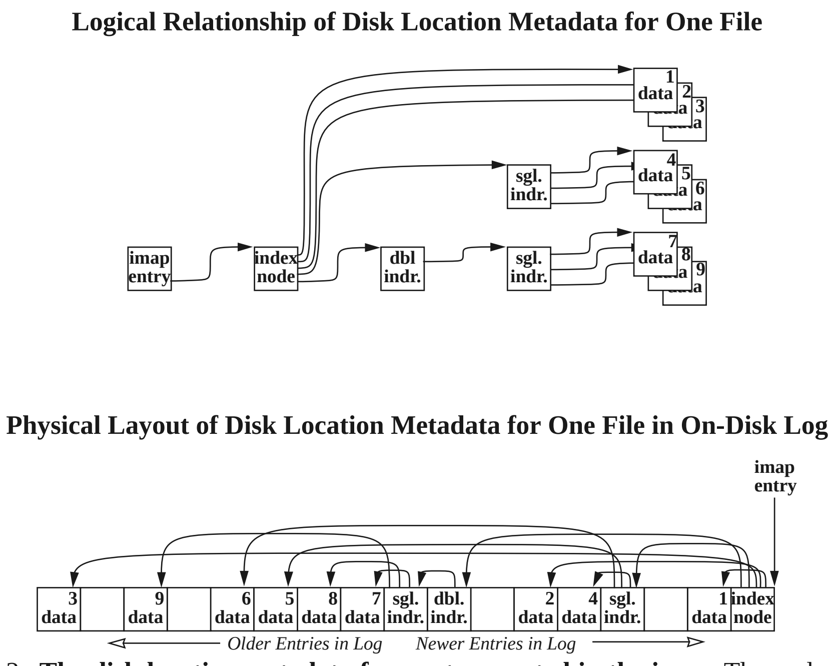


FIGURE 5-2. **The disk location metadata form a tree rooted in the imap.** The nodes of the tree are index nodes, data blocks, single-indirect blocks, double-indirect blocks, and (not shown) triple-indirect blocks. In the figure, the blocks are numbered to show the sequential order of the blocks; however, in a real system there are more blocks of each type than shown in this illustration, and the numbering would be correspondingly changed. This figure shows the simple case where a client on a single machine writes a file's data blocks, and a manager on the same machine writes the file's index node and indirect nodes. If the client and manager are on different machines, the index nodes and indirect nodes will be in different logs than the data blocks, but the nodes in the manager's log will still contain pointers to the data in the client's log.

addresses of indirect blocks that contain more data block addresses, double indirect blocks that contain addresses of indirect blocks, and so on. The manager stores the imap in its memory and periodically checkpoints it to the storage system to speed recovery. The lower levels of the tree are stored on storage server disks, although managers cache index nodes and (all levels of) indirect blocks, and clients cache data blocks.

5.1.2. Cache Consistency State

The same machine that handles a file's disk metadata also handles cache consistency state for the blocks of that file. For each block, this cache consistency state consists of a list of clients caching that block and a flag indicating whether that block is write-owned [Archibald and Baer, 1986]. Using this information, the manager forwards client read requests to other clients caching the blocks, and the manager maintains cache consistency by invalidating stale cached copies when a client writes or deletes a file.

5.2. System Operation

This section describes how distributed management works in conjunction with the log-based, distributed storage system described in the previous chapter. I describe the operation of the system for reads and writes and then detail how it manages index nodes and index numbers.

5.2.1. Reads and Caching

Figure 5-3 illustrates how the serverless system reads a block given a file name and an offset within that file. Although the figure is complex, the complexity in the architecture is designed to provide good performance. On a fast LAN, fetching a block out of local memory is much faster than fetching it from remote memory, which, in turn, is much faster than fetching it from disk.

To open a file, the client first reads the file's parent directory (labeled *1* in the diagram) to determine its index number. Note that the parent directory is, itself, a data file that must be read using the procedure described here. As with FFS, the serverless system breaks this recursion at the root; the kernel learns the index number of the root when it mounts the file system.

As the top left path in the figure indicates, the client first checks its local UNIX block cache for the block (*2a*); if the block is present, the request is done. Otherwise it follows the lower path to fetch the data over the network. The system first uses the manager map to locate the correct

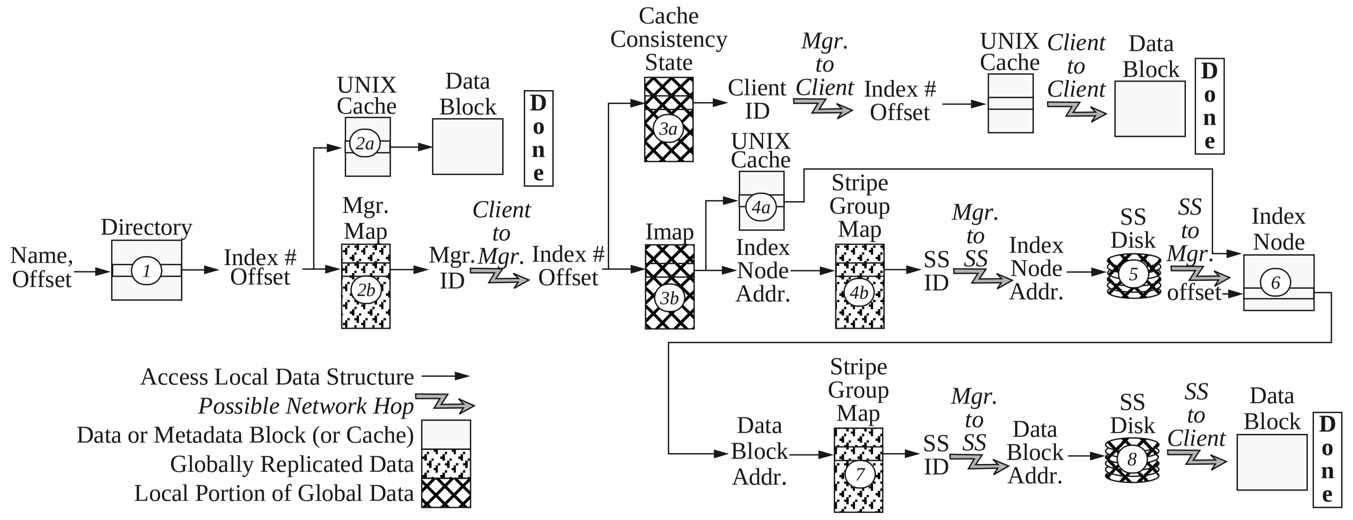


FIGURE 5-3. **Procedure to read a block.** The circled numbers refer to steps described in Section 5.2.1. The network hops are labelled as “possible” because clients, managers, and storage servers can run on the same machines. For example, The system tries to co-locate the manager of a file on the same machine as the client most likely to use the file to avoid some of the network hops. “SS” is an abbreviation for “Storage Server.”

manager for the index number (*2b*) and then sends the request to the manager. If the manager is not co-located with the client, this message requires a network hop.

The manager then tries to satisfy the request by fetching the data from some other client's cache. The manager checks its cache consistency state (*3a*), and, if possible, forwards the request to a client caching the data. That client reads the block from its UNIX block cache and forwards the data directly to the client that originated the request. The manager also adds the new client to its list of clients caching the block.

If no other client can supply the data from memory, the manager routes the read request to disk by first examining the imap to locate the block's index node (*3b*). The manager may find the index node in its local cache (*4a*) or it may have to read the index node from disk. If the manager has to read the index node from disk, it uses the index node's disk log address and the stripe group map (*4b*) to determine which storage server to contact. The manager then requests the index block from the storage server, who then reads the block from its disk and sends it back to the manager (*5*). The manager then uses the index node (*6*) to identify the log address of the data block. (I have not shown a detail: if the file is large, the manager may have to read several levels of indirect blocks to find the data block's address; the manager follows the same procedure in reading indirect blocks as in reading the index node.)

The manager uses the data block's log address and the stripe group map (*7*) to send the request to the storage server keeping the block. The storage server reads the data from its disk (*8*) and sends the data directly to the client that originally asked for it.

5.2.2. Writes

When a client writes a block, it interacts with the manager twice. First, to maintain cache consistency it acquires write ownership of the block if it has not already done so. Second, when a client commits the block to a storage server, it informs the block's manager of the block's new log address so that the manager can update the file's index node and, if the file is large, the file's indirect blocks. The consistency protocol and commit protocol interact when blocks are shared. To preserve consistency across failures, clients with exclusive access to modified data must flush all logged changes to stable storage before allowing other clients to read the data. This subsection details the activity related to cache consistency, commits, and ownership-loss writes.

5.2.2.1. Cache Consistency

The managers implement a token-based cache consistency scheme similar to Sprite [Nelson et al., 1988], Andrew [Howard et al., 1988], Spritely NFS [Srinivasan and Mogul, 1989], Coda [Kistler and Satyanarayanan, 1992], and Echo [Birrell et al., 1993] except that they manage consistency on a per-block rather than per-file basis to allow more efficient data sharing [Burrows, 1988]. Before a client modifies a block, it must acquire write ownership of that block, so the client sends a message to the block's manager. The manager then invalidates any other cached copies of the block, updates its cache consistency information to indicate the new owner, and replies to the client, giving permission to write. Once a client owns a block, the client may write the block repeatedly without having to ask the manager for ownership each time. The client maintains write ownership until some other client reads or writes the data, at which point the manager revokes ownership, forcing the client to stop writing the block, flush the current segment to the storage servers if it contains changes to the block, and forward the data to the new client.

The managers use the same state for both cache consistency and cooperative caching. The list of clients caching each block allows managers to invalidate stale cached copies in the first case and to forward read requests to clients with valid cached copies in the second.

5.2.2.2. Commit of Writes to Log

Clients buffer writes in their local memory until committed to a stripe group of storage servers. Because the storage system uses log-based striping, every write changes the disk address of the modified block. Therefore, after a client commits a segment to a storage server, the client notifies the modified blocks' managers; the managers then update their imaps, index nodes, and indirect blocks and periodically log these changes to stable storage. As with Zebra, the system does not need to "simultaneously" commit index and indirect nodes with their associated data blocks because the client's log includes a *delta* that allows reconstruction of the manager's data structures in the event of a client or manager crash. I discuss deltas in more detail in Section 5.4.

As in BSD LFS [Seltzer et al., 1993], each manager stores its portion of the imap in memory, checkpointing it to disk in a special file called the *ifile*. The system treats the ifile like any other file with one exception: the ifile has no index nodes. Instead, when managers boot, they locate the blocks of the ifile using manager checkpoints described in Section 5.4.

5.2.2.3. Writes Due to Loss of Ownership

To facilitate recovery, the protocol requires clients to commit modified data to disk before releasing ownership. By guaranteeing that clients commit modified data before allowing other clients to observe that data, the system ensures that a modification that depends on another modification will not be committed before the state change on which it depends. For instance, if client A makes directory “foo,” and then client B tries to modify directory “foo” to write file “foo/bar,” and then one or both clients crash, upon recovery the system should be in one of three states

1. Neither “foo” nor “foo/bar” exist.
2. “Foo” exists but “foo/bar” does not.
3. Both “foo” and “foo/bar” exist.

The system would be inconsistent if, after recovery, “foo/bar” exists, but “foo” does not. By forcing “foo” to stable, reliable storage before allowing client B to modify “foo” to create “foo/bar,” the protocol guarantees that either case 2 or case 3 prevails after recovery.

Writing data to disk before releasing ownership will hurt performance if there is significant write sharing. Each time a client loses write ownership of a block, it may have to write a partial segment to disk, reducing disk efficiency [Baker et al., 1992]. Additionally, the client that is attempting to access the new data must endure the latency of a synchronous disk write before it can begin writing. For many workloads, little write sharing occurs, so this delay in revoking ownership will have little effect on performance [Thompson, 1987, Baker et al., 1991, Blaze, 1993]. Further, by maintaining consistency on a per-block rather than a per-file basis, the consistency protocol reduces ownership-loss writes caused by false sharing. On the other hand, ownership-loss writes could delay parallel workloads that have fine-grained write sharing; I have not yet determined how significant such delays will be. If they are significant, a number of techniques could reduce the delay while maintaining acceptable crash-recovery semantics. For instance, rather than commit pending changes to disk when losing ownership, a client could send those changes over the network to the client whose access caused write ownership to be revoked [Liskov et al., 1991]; that client would add those changes to its log, guaranteeing that those changes make it to disk before any modifications that depend on them, even if the first client crashes. As an optimization, once either client successfully commits the duplicated part of the log to disk, it could “cancel” the duplicate writes at the other client.

5.2.3. Index Node and Indirect Node Access

One important design decision was to cache index nodes and indirect nodes only at managers and not at clients. Although caching these nodes at clients would allow them to read multiple blocks from storage servers without sending a request through the manager for each one, doing so has four significant drawbacks. First, by reading blocks from disk without first contacting the manager, clients would lose the opportunity to use cooperative caching to avoid disk accesses. Second, although clients could sometimes read a data block directly, they would still need to notify the manager of the fact that they are now caching the block so that the manager knows to invalidate the block if it is later modified. Third, routing all read requests through a manager allows managers to enforce security rather than relying on clients; in contrast with Zebra's design, where clients can access any block stored on a storage server by constructing the proper block pointer [Hartman and Ousterhout, 1995], clients never read data directly from storage servers, allowing managers to enforce security on client reads. Similarly, managers can enforce security on client writes; even though the clients write directly to storage servers, the system will ignore the writes unless the manager accepts the new location of the blocks as legitimate. Finally, this approach simplifies the design by eliminating client caching and cache consistency for index nodes and indirect nodes — only the manager handling an index number directly accesses these structures.

5.2.4. Assigning Files to Managers

The system can control how files are assigned to managers in two ways, by manipulating files' index numbers or by manipulating the manager map. Controlling index numbers allows fine-grained control while manipulating the manager map provides a method to simultaneously change many files' managers.

5.2.4.1. Assigning and Changing Index Numbers

Because a file's index number determines its manager, the system can control the assignment of files to managers by controlling the use of index numbers. The simplest mechanism for doing this is to control the choice of index numbers when files are created; systems must already choose an index number for every file they create, so modifying this choice to control management assignments is straightforward. A more complicated, but more flexible, mechanism is to change file index numbers after the files have been created. Section 5.3 quantifies the benefits of algorithms that use each approach.

Although assigning index numbers when creating files requires little or no modification of file system interfaces, allowing the systems to dynamically change file index numbers complicates matters. As Figure 5-4 shows, to change a file's index number, a system changes the file's directory to reflect the new mapping from name to index number, it removes the old imap entry for that file, and it adds a new imap entry that contains a pointer to the file's index node. Because these three changes must be atomic, the system must support a new type of log delta to encapsulate these changes. Further, because this functionality requires the system to change the directory entry for the file, it does not support multiple hard links to a single file (soft links must be used instead). Multiple hard links would mean that several directory entries all contain references to the same imap entry, but the system does not provide a mechanism to locate all of the aliases to the same file.

5.2.4.2. Changing the Manager Map

The system can assign groups of files to new managers by changing the manager map. This mechanism allows the system to remap many files with a single action and without accessing each file individually. It is thus suited for relatively coarse-grained adjustments such as assigning all of the files previously managed by one machine to other machines when the first machine crashes. In that case, fast recovery is more important than precise control over where the system assigns individual files.

The system might also use this mechanism for balancing load when no machines have crashed. For instance, if one machine becomes heavily loaded, this mechanism would allow the

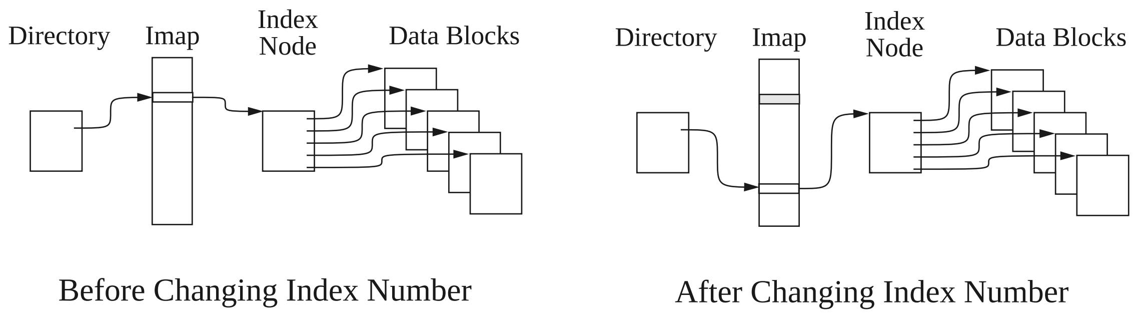


FIGURE 5-4. **Changing a file's index number.** Changing a file's index number requires three simultaneous modifications to the file system's state. First, that file's directory entry must be modified to map the file's name to its new index number. Second, the imap entry for the old index number must be released. Third, the imap entry for the new index number must be updated to include a pointer to the file's index node and data blocks. Note that the file's index node and data blocks are not changed.

system to quickly distribute a portion of that machine’s load to other machines. Unfortunately, the coarse control provided by this mechanism forces the system to reassign many files at once, so while the system may benefit from improved load balance, it is also likely to suffer reduced locality as “private” files managed by the overloaded machine are reassigned along with the rest of the files. Studying policies that balance locality against load distribution remains future work.

5.3. Management Locality

The previous sections described mechanisms to distribute management across multiple machines, but the policies used to do so will determine distributed management’s impact on performance. Distributing management can improve performance in two ways: by improving locality and by distributing load. This section examines the issue of locality by using trace-driven simulation studies to examine the impact of several distributed management policies on locality. Because the file system traces to which I have access have relatively little concurrency, I defer a quantitative evaluation of load distribution until Section 7.2.8 on page 144, where I examine the performance of distributed management in the prototype implementation using a synthetic workload.

This section compares three policies. The Centralized policy uses a single, centralized manager that is not co-located with any of the clients to provide a baseline for comparison. Under the First Writer policy, when a client creates a file, the system chooses an index number that assigns the file’s management to the manager co-located with that client. Finally, under the Last Writer policy, the system dynamically assigns files to be managed by the manager co-located with the last client to begin writing the files’ first blocks.

Based on the simulation studies, I conclude that co-locating a file’s management with the client using that file can significantly improve locality and reduce network communication, particularly for writes. For the workloads I looked at, the First Writer policy is sufficient to reap most of the performance gains. The more complicated Last Writer policy provides only a small additional improvement. Although these workloads do not appear to justify the added complexity of the Last Writer policy, other workloads might. For instance, in a NOW, parallel programs, process migration, and batch jobs may increase the amount of data that are written and shared by multiple clients and that would benefit from this policy. As NOW workloads become available, this policy should be evaluated in that context.

5.3.1. Methodology

I simulate the system's behavior under two sets of traces. The first set consists of four two-day traces gathered by Baker from the Sprite file system. This set of traces was described in more detail in Section 3.2 on page 35, and is analyzed in detail elsewhere [Baker et al., 1991]. The second workload consists of a seven-day trace of 236 clients' NFS accesses to an Auspex file server in the Berkeley Computer Science Division. Section 3.3.4.2 on page 56 described this trace in more detail. For each trace, the simulator warms the simulated caches through the first day of the trace and gathers statistics through the remainder.

The finite length of the traces introduces a bias into the simulation which may reduce the apparent benefits of distributed management. The finite trace length does not allow the simulator to determine a file's "First Writer" with certainty for references to files created before the beginning of the trace. For files that are read or deleted in the trace before being written, I assign management to random managers at the start of the trace; when and if such a file is written for the first time in the trace, I move its management to the first writer. Because write sharing is rare — 96% of all block overwrites or deletes are by the block's previous writer — this heuristic should yield results close to a true "First Writer" policy for writes, although it will give pessimistic locality results for "cold-start" read and write misses.

The simulator counts the network messages needed to satisfy client requests, assuming that each client has 16 MB of local cache and that there is a manager co-located with each client, but that storage servers are always remote.

An artifact of the Auspex trace affects read caching for that trace. Because the trace was gathered by snooping the network, it does not include reads that resulted in local cache hits. By omitting requests that resulted in local hits, the trace inflates the average number of network hops needed to satisfy a read request. I therefore report the number of network hops per read miss rather than per read when reporting results for this trace. Because I simulate larger caches than those of the traced system, this factor does not significantly alter the total number of network requests for each policy [Smith, 1977], which is the relative metric used for comparing policies.

5.3.2. Results

Figure 5-5 and Figure 5-6 show the impact of the simulated policies on locality for the Sprite traces and the Auspex trace. The First Writer policy reduces the total number of network hops needed to satisfy client requests by 11% to 50% for the different traces compared to the centralized policy. Most of the difference comes from improving write locality; the algorithm does little to improve locality for reads, and deletes account for only a small fraction of the system's network traffic. The dynamic Last Writer policy improves locality only slightly compared to the First Writer policy for all of the traces.

Figure 5-7 and Figure 5-8 illustrate the average number of network messages to satisfy a read block request, write block request, or delete file request. Despite the large number of network hops that can be incurred by some requests (see Figure 5-3 on page 98), the average per request is quite low. For instance, in the first two-day Sprite trace under the First Writer policy, 65% of read requests in the trace are satisfied by the local cache, requiring zero network hops. An average local

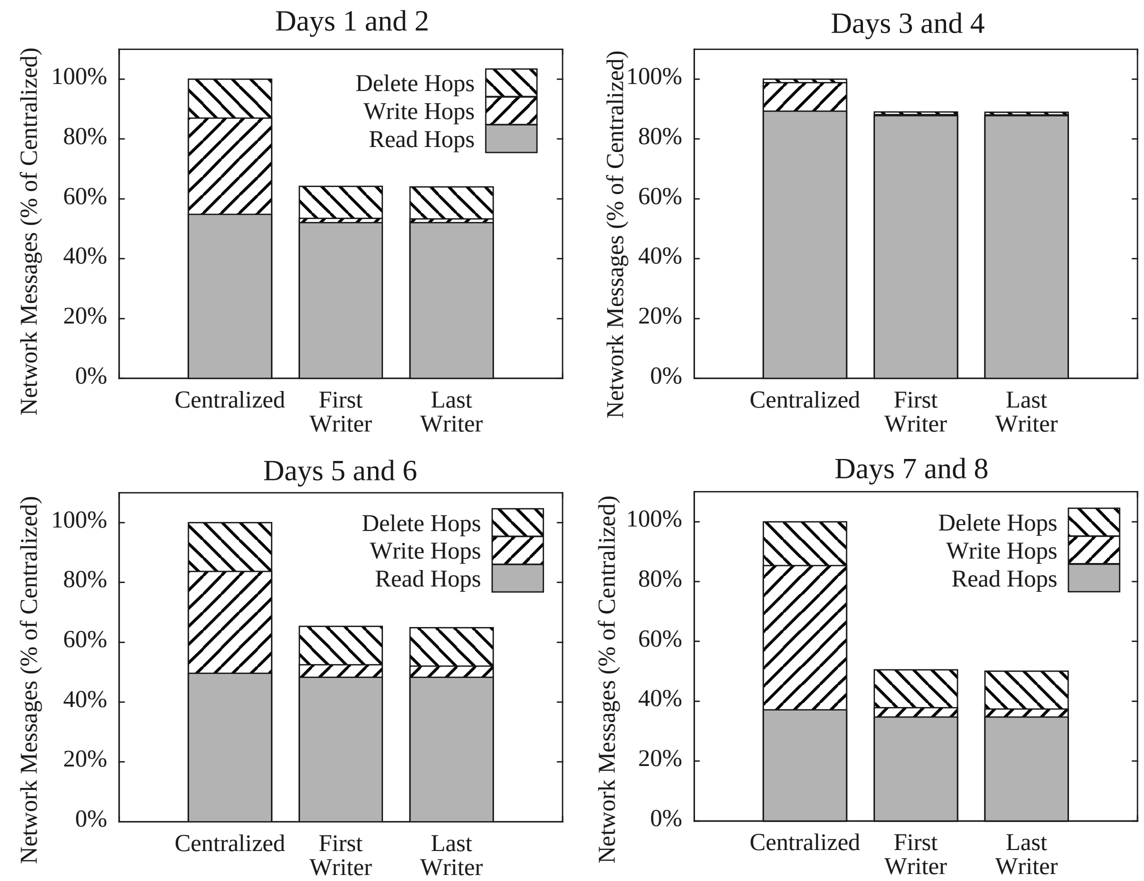


FIGURE 5-5. **Comparison of locality.** Locality for the four two-days Sprite traces is indicated by the reduced network traffic of the First Writer and Last Writer policies compared to the Centralized policy. The y axis indicates the total number of network messages sent under each policy as a fraction of the messages sent by the Centralized policy.

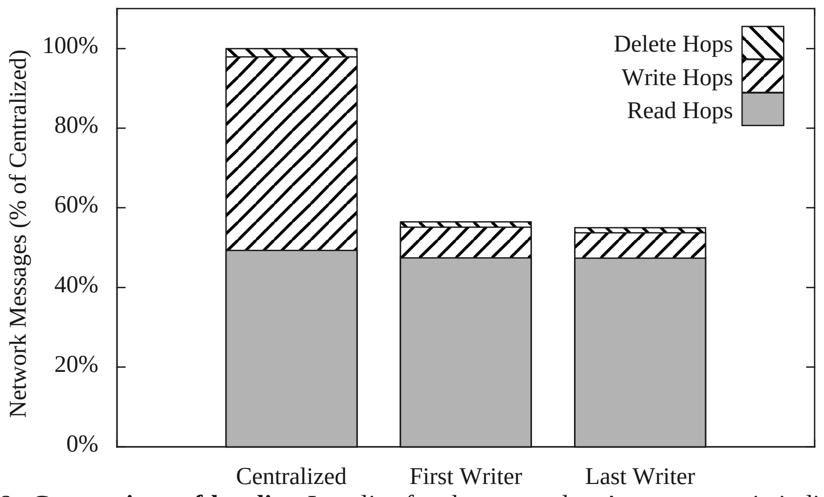


FIGURE 5-6. **Comparison of locality.** Locality for the seven-day Auspex trace is indicated by the reduced network traffic of the First Writer and Last Writer policies compared to the Centralized policy. The y axis indicates the total number of network messages sent under each policy as a fraction of the messages sent by the Centralized policy.

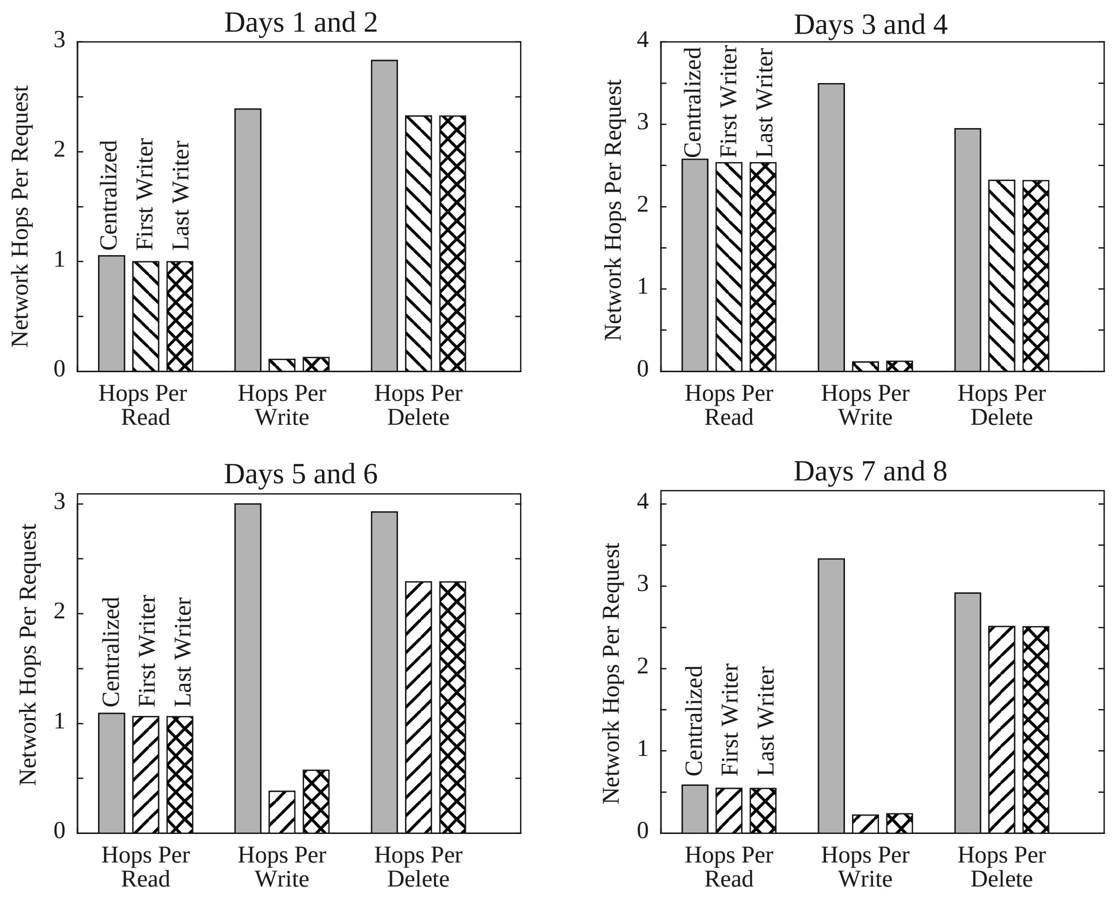


FIGURE 5-7. **Network messages per request.** Average number of network messages needed to satisfy a read block, write block, or delete file request under the Centralized and First Writer policies. The Hops Per Write column does not include a charge for writing the segment containing block writes to disk because the segment write is asynchronous to the block write request and because the large segment amortizes the per block write cost.

read miss costs 2.85 hops; a local miss normally requires three hops (the client asks the manager, the manager forwards the request, and the storage server or client supplies the data), but 16% of the time it can avoid one hop because the manager is co-located with the client making the request or the client supplying the data. Under all of the policies, a read miss will occasionally incur a few additional hops to read an index node or indirect block from a storage server. The other traces and the Last Writer policy have similar characteristics.

Writes benefit more dramatically from locality. For the first Sprite trace under the First Writer policy, of the 37% of write requests that require the client to contact the manager to establish write ownership, the manager is co-located with the client 96% of the time. When a manager invalidates stale cached data, one-third of the invalidated caches are local. Finally, when clients flush data to disk, they must inform the manager of the data's new storage location, a local operation 97% of the time in this trace. Again, the other traces and the Last Writer policies are similar, although the Last Writer policy provides slightly more locality for writes.

Deletes, though rare, benefit modestly from locality. For the first Sprite trace and the First Writer policy, 29% of file delete requests go to a local manager, and 83% of the clients notified to stop caching deleted files are local to the manager. The other Sprite traces have similar characteristics, and the Auspex trace has somewhat better delete locality.

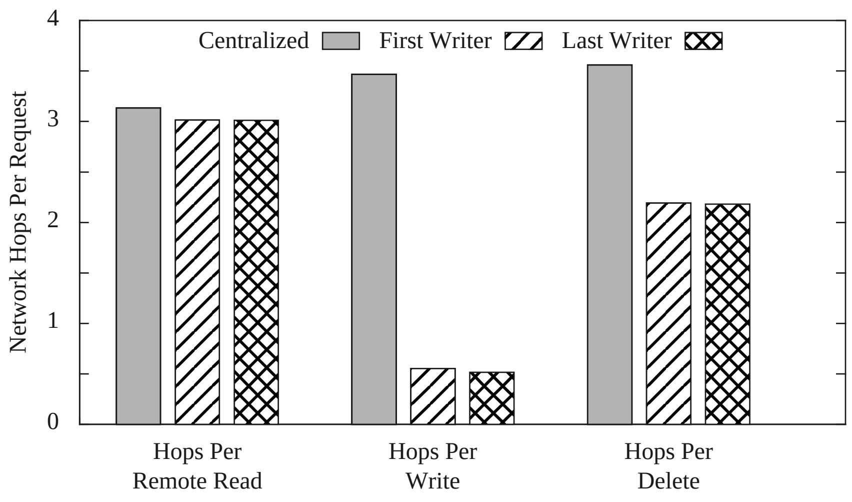


FIGURE 5-8. **Network messages per request.** Average number of network messages needed to satisfy read block (considering only reads that miss in the local cache), write block, or delete file request unde the Centralized and First Writer policies. The Hops Per Write column does not include a charge fc writing the segment containing block writes to disk because the segment write is asynchronous to th block write request and because the large segment amortizes the per block write cost.

5.4. Recovery and Reconfiguration

Because there can be large numbers of managers in a serverless system, the system must continue to function if some of the managers fail, and it must adjust when managers are added. To accomplish this, managers implement a reconfiguration algorithm that allows one manager to take over another's functions. If there is a crash, one of the remaining managers recovers the lost manager's state and then performs its duties. Similarly, if a new manager joins the system, it takes over some of the established managers' state using the recovery procedures. When the system as a whole reboots, all of the managers take part in recovery to rebuild the system's entire management state.

As the rest of this section describes, manager recovery proceeds in three stages to assemble three sets of state. First, all active managers participate in a global consensus algorithm to agree on a new manager map. This step happens first because it assigns different parts of the index number space to different managers for the later steps. Second, managers recover their disk-location metadata for the files they manage. This step exploits the redundant, log-structured storage system described in the previous chapter. Third, managers recover the cache consistency state for their files by polling clients to determine which clients are caching which files. After describing these three phases of recovery, this section reviews cleaner recovery, which relies on file metadata recovery, and then it evaluates the scalability of these recovery procedures.

5.4.1. Manager Map Reconfiguration

When the number of managers in the system changes, either because a machine has been added, because a machine has failed, or because multiple machines have rebooted, the system generates a new manager map and distributes it to all active manager machines. The system builds and distributes the manager map using a distributed consensus algorithm that both identifies all of the machines that will act as managers under the new mapping and elects a leader from among those machines. Fortunately, several distributed consensus algorithms are known [Ben-Or, 1990, Cristian et al., 1990, Cristian, 1991, Schroeder et al., 1991].

Once the system chooses a leader, the leader creates the new manager map. To balance load across managers, it assigns roughly equal numbers of manager map entries to each active manager machine. As an optimization, it may use the old map as a starting point when generating the new map; this approach allows the system to limit the number of manager map entries changed and thus maintains most of the locality between managers and the files that they manage. Once the manager has generated a map, it distributes it to the active manager machines.

5.4.2. Recovery of Disk Location Metadata

The managers then recover the disk location metadata so that they can locate all of the data blocks stored in the system's storage servers. To recover this information, the managers build their global, distributed imap. This imap contains pointers to on-disk copies of the system's index nodes, which, in turn, point to the data blocks on disk. The system recovers this information using the same approach as other LFS systems [Rosenblum and Ousterhout, 1992, Seltzer et al., 1993, Hartman and Ousterhout, 1995]: first it recovers a checkpoint of the imap from the logs stored on storage server disks and then it rolls forward the logs to account for blocks that were written to the log since the last checkpoint.

The recovery design relies on two facets of the storage server architecture described in the previous chapter. First, it takes advantage of the high availability provided by redundant, striped storage. Thus, the procedures described here assume that the underlying storage system is highly reliable, and they do not have to explicitly deal with lost data; lower levels of the recovery protocol do that. As long as no more than one storage server per stripe group has failed, manager recovery can proceed. If more than one storage server in a stripe group is down, manager recovery is stalled until it can proceed (multiple parity fragments per group [Blaum et al., 1994] would allow recovery to continue in the face of multiple storage server failures.) Second, disk metadata recovery uses the system's log-structured storage to focus recovery efforts on only the segments written at about the time of the crash, and it uses log records, called deltas, that make it easy to determine what operations were in progress at the time of the crash. In contrast, without a log, the system would have to scan all disk storage to discover any partially-completed operations that were in progress at the time of the crash. Without reliable, log-structured storage, it would be much more difficult to build a reliable, distributed manager.

The procedure described here draws heavily on the recovery design used in Zebra [Hartman and Ousterhout, 1995]. Zebra demonstrates how a single manager can recover all of a system's disk location metadata from multiple clients' logs. To adapt this approach to handle multiple managers, I divide recovery so that different machines recover different parts of the disk location metadata by reading different parts of the systems' logs. This approach works because the disk location metadata for different files are largely independent. The following subsections discuss distributed checkpoint recovery, distributed roll-forward, and consistency issues that arise from independent manager checkpoints.

5.4.2.1. Distributed Checkpoint Recovery

The managers' checkpoints allow the system to recover the state of the disk storage as it existed a short time before the reconfiguration without scanning the logs from beginning to end. The system stores important disk metadata in its logs and stores pointers to this metadata in log checkpoints that it can locate and read quickly during recovery. For scalability, managers write and recover their checkpoints independently.

During normal operation, managers keep the system's imap, which contains the log addresses of the index nodes, in memory. Periodically, however, they write modified parts of the imap to their logs so that the on-disk ifile contains a nearly up-to-date copy of the imap. Managers also write checkpoints to their logs. A checkpoint is, in effect, the index node of the ifile in that it contains pointers to the on-disk blocks of the ifile [Seltzer et al., 1993]. Each manager writes checkpoints for the portion of the imap that it manages, and collectively the checkpoints contain the disk addresses of all of the blocks of the ifile. Checkpoints differ from standard index nodes in that they are tagged in the log so that they may be easily located when the ends of the logs are scanned during recovery.

Figure 4-2 on page 72 illustrates the logical relationship among data blocks, index nodes, the imap, the ifile, and checkpoints and shows how they might be arranged in the log. This figure illustrates two important things about the checkpoints. First, checkpoints refer only to ifile blocks that were written to the log before the checkpoint. Second, checkpoints do not always reflect the most recent modifications to the file system; the system updates the imap to reflect more recent changes during roll forward, as the next section describes. Checkpoints thus provide a snapshot of the system's state, although the picture may not be completely current.

During recovery, managers read their checkpoints independently and in parallel. Each manager locates its checkpoint by first querying storage servers to locate the newest segment written to its log before the crash and then reading backwards in the log until it finds the segment with the most recent checkpoint. Next, managers use this checkpoint to recover their portions of the imap. Although the managers' checkpoints were written at different times and therefore do not reflect a globally consistent view of the file system, the next phase of recovery, roll-forward, brings all of the managers' disk-location metadata to a consistent state corresponding to the end of the clients' logs.

5.4.2.2. Distributed Roll Forward

To account for modifications since the time of the most recent checkpoint, the system *rolls forward* the clients' logs. To roll a log forward, a client reads forward in time, using deltas [Hartman and Ousterhout, 1995] to replay the operations that occurred between the time of the checkpoint and the time of the crash.

Each delta contains enough information to identify the changes to the blocks of a file. A delta contains five pieces of information:

1. The *index number* of the modified file.
2. The *file offset* of the block that was modified in the file.
3. The *block version number* that indicates when the client that wrote the block acquired ownership of it. Version numbers allow ordering of deltas from different clients' logs that modify the same block.
4. The *old block address* gives the block's previous log address. After a crash, the system uses the old block pointers to recover the segment utilization information used by cleaners, and the managers use this field to detect and correct races caused by the simultaneous cleaning and modification of a file.
5. The *new block address* is the block's new log address.

These fields are like those used in Zebra's deltas with one exception. To support per-block cache consistency and write sharing, these deltas use per-block version numbers rather than per-file versions. If the system implemented per-file version numbers like those of Zebra, the system could allow only one client to write a file at any time; in contrast, per-block version numbers allow different clients to simultaneously modify a file, as long as they are accessing different blocks. Managers supply block version numbers to clients when they grant write ownership — the version number is simply the time stamp when the client acquired write ownership from the manager.

A simple, though inefficient, way for managers to roll forward from their checkpoints is for each manager to begin scanning each client's log starting from the time of the manager's checkpoint. A manager would ignore deltas pertaining to files managed by other managers, but it would apply the deltas that refer to its files to its imap. This approach is inefficient because it reads each client's log from disk multiple times. The problem with having managers read the clients' logs is that each log can contain modifications to any file managed by any manager, so all managers must inspect all clients' logs.

A small change restores efficiency. Instead of having managers read client logs from disk, clients read the logs and send the deltas to the managers that need them. When recovery starts, the system assigns clients to roll forward each log using the consensus procedure described in Section 5.4.1. Normally, each client recovers the log that it wrote before the crash, but if a client is down when the system starts roll forward, the system can assign a different client to roll forward that log. To initiate roll forward, the recovering managers use information in their checkpoints to determine the earliest segment that each client must roll forward, and they transmit this information to the clients. Each client then reads backwards in time from the tail of its log until it locates the earliest segment requested by any manager. Clients then begin reading forward in the log, sending each delta to the manager indicated by the index number in the delta.

5.4.2.3. Consistency of Disk Location Metadata

Distributing management across multiple machines raises the issue of consistency across checkpoints written by different managers. Checkpoints written by different managers at different times do not generally correspond to the same point in a client's log. Therefore, after checkpoint recovery, different portions of the index number space will reflect different moments in time, possibly leading to inconsistencies. For instance, the directory entry for a newly created file might exist, but the file itself might not if the directory index number was recently checkpointed but the file's part of the index number space was not checkpointed since the file was created.

Two solutions are possible. First, the system can allow inconsistent manager checkpoints and use roll-forward to restore consistency. Second, the system can synchronize checkpoints across managers so that they reflect a consistent view of the world.

Allowing inconsistent checkpoints but then rolling forward to a consistent state is the more efficient solution to this problem because managers can then write checkpoints independently. Roll-forward results in a consistent global state because it brings all managers' metadata to the same state corresponding to the end of each client's log. Note that the state reflected by the clients' logs must be consistent, even if the clients' logs end at slightly different times, because each segment can only depend on segments already written to disk. As Section 5.2.2.3 explained, before a client reads data that another client wrote, the client that wrote the data forces its dirty data to disk as it loses write ownership.

Another approach to consistent recovery is to synchronize checkpoints so that they reflect a globally consistent state. The advantage of this strategy is that it does not require roll forward, potentially simplifying implementation. The disadvantage of this approach is that it requires a global barrier at the time of a checkpoint, limiting scalability: the system must designate the point in all of the clients' logs at which the checkpoint will occur, and it cannot allow managers to modify their state to reflect any later client modifications until after the managers write their checkpoints.

5.4.3. Cache Consistency State

After the managers have recovered and rolled forward the imap, they must recover the cache consistency state associated with the blocks that they manage. This state consists of lists of clients caching each data block plus a flag indicating whether the client caching a block holds write ownership for that block. The Sprite file system demonstrates how the managers can recover this information by polling clients for the state of their caches [Nelson et al., 1988]. With a distributed manager, each manager can recover its portion of the index number space by polling the clients. The parallelism across managers actually reduces the risk of recovery storms possible in the Sprite system [Baker, 1994] because the managers are less likely to overwhelmed by the clients. However, the reverse problem can occur if the multiple managers all ask a single client for the different parts of its cache state; to avoid this problem, managers randomize the order in which they poll clients.

5.4.4. Cleaner Recovery Revisited

As Section 4.5.3 on page 90 indicated, the storage servers' cleaners store their persistent state in standard files called s-files. The procedures described earlier in this section recover the s-files along with all of the other data files in the system. However, the s-files may not reflect the most recent changes to segment utilizations at the time of the crash, so s-file recovery also includes a roll forward phase. Each client rolls forward the utilization state of the segments tracked in its s-files by asking the other clients for summaries of their modifications to those segments that are more recent than the s-files' checkpoints. To avoid scanning their logs twice, clients can gather this segment utilization summary information during the roll-forward phase for manager disk location metadata.

5.4.4.1. Scalability of Recovery

Two facets of the recovery design allow it to scale to large amounts of storage and to large numbers of clients, storage servers, and managers. First, log-structured storage allows the system to examine only the most recently written segments; in contrast, after a crash other disk layouts, such as FFS, scan all blocks stored on disk because any write can affect any area of the disk (auxiliary journals [Hagmann, 1987, Kazar et al., 1990] could be employed by these systems to avoid scanning the disk during recovery). Second, the system distributes recovery so that each client or manager log is read only once, allowing different machines to recover different portions of the system's state in parallel. While these properties of the architecture suggest that it will scale well, future work is required to measure the performance of recovery in practice.

The system's log-structured storage provides the basis for scalable recovery by restricting modifications to the tails of the system's logs. During recovery, the system examines only the most recently written segments; managers read their logs back to the most recent checkpoint, and clients read the segments written since the manager checkpoints. The system can reduce recovery time by reducing the interval between checkpoints, although this increases the overhead for writing checkpoints during normal operation.

Distributing recovery across managers and clients provides scalability by parallelizing recovery; although increasing the size of a system may increase the amount of state to recover after a crash, different clients and managers can independently recover their own state so that recovery time is proportional to the amount of state recovered per client or manager rather than to the total amount of state recovered.

Disk metadata recovery proceeds in four steps, each of which progresses in parallel. First, clients and managers locate the tails of their logs. To enable machines to locate the end of the logs they are to recover, each storage server tracks the newest segment that it stores for each client or manager. A machine can locate the end of the log it is recovering by asking all of the storage server groups and then choosing the newest segment. While this procedure requires $O(N^2)$ messages (where N corresponds to the number of clients, managers, or storage servers) to allow each client or manager to contact each storage server group, each client or manager only needs to contact N storage server groups, and all of the clients and managers can proceed in parallel, provided that they take steps to avoid recovery storms where many machines simultaneously contact a single storage server [Baker, 1994]; randomizing the order that machines contact one another accomplishes this goal.

The second step of disk metadata recovery also proceeds in parallel. Each manager scans its log backwards from the tail until it locates and reads the segment that contains its last checkpoint. Because managers can scan their logs independently, the time for this step depends primarily on the interval between checkpoints, assuming that there are enough storage servers relative to managers so that manager requests to read segments do not significantly interfere with one another.

In the third step of recovery, managers inform clients of where to start roll forward. Each manager sends each client a message that indicates the last segment in that client's log that manager had processed at the time of the checkpoint; each client begins roll forward from the earliest segment requested by any manager. Again, managers contact clients independently and in parallel, so that this phase requires N steps, assuming that it avoids recovery storms.

Next, clients roll forward their logs by reading segments written since the manager checkpoints and sending information from the deltas to the managers. The time for this phase depends primarily on the amount of data scanned per client, which is determined by the checkpoint interval rather than the size of the system.

Finally, managers recover their cache consistency state by polling the clients. As in the other phases of recovery, the managers proceed in parallel to make this phase efficient.

5.5. Related Work

This chapter describes a distributed metadata management system that implements management as a separate module from data storage. This separation of function leads to a clean manager interface that may easily be distributed across multiple machines. The Mass Storage Systems Reference Model [Coyne and Hulen, 1993] also recommends separate data and control paths for file systems, and Zebra implements file systems using modules with similar functions to those described here [Hartman and Ousterhout, 1995], although Zebra's manager ran on a single machine.

A small number of file systems have distributed their metadata across multiple machines. However, a larger number of massively parallel, distributed shared memory machines have distributed their cache consistency metadata using related techniques. In either case, the approaches are based on one of two basic data structures. The first set of approaches, which distribute data by hashing, are similar to the architecture described in this chapter. The second set of approaches base their distribution on hierarchical trees.

5.5.1. Hash-based Metadata Distribution

This chapter described an approach to metadata distribution built around a distributed hash table. Similar approaches have been taken by a network virtual memory system, by multiprocessor cache consistency systems, and by one other file system that I know of.

Most of these previous systems have used a fixed mapping from data address to manager. In such systems, if one machine fails, a fraction of the system's metadata becomes unavailable until the machine recovers. Furthermore, it is difficult to add new machines to such systems or to change the mapping of addresses to machines to balance load, because doing so requires all metadata to be redistributed according to a new hash function. In contrast, Feeley's network virtual memory system [Feeley et al., 1995] uses a level of indirection similar to the serverless file system's manager map to allow reconfiguration of management responsibilities.

The Vesta file system [Corbett et al., 1993] distributes metadata by hashing on a file's path name to locate the machine that controls the file's metadata. However, metadata in Vesta is much simpler than in the serverless system described in this chapter; Vesta's file layout on disk is restricted to variations of round-robin mappings across storage servers. Additionally, Vesta uses a fixed mapping from file names to manager addresses, so it cannot easily reconfigure management responsibilities.

File systems' cache consistency protocols resemble the directory-based multiprocessor memory schemes [Tang, 1976, Censier and Feautrier, 1978, Archibald and Baer, 1984, Yen et al., 1985, Agarwal et al., 1988]. In both cases, the cache consistency state allows the consistency manager to prevent nodes from caching stale data. The DASH multiprocessor demonstrates how to provide scalability by distributing the consistency directory [Lenoski et al., 1990]; each DASH node manages a subset of the system's cache consistency state. Unfortunately, in DASH and related, distributed, multiprocessor cache consistency schemes [Chaiken et al., 1991, Kuskin et al., 1994], the distribution of state across managers is fixed, limiting their availability and ability to reconfigure.

5.5.2. Hierarchical Metadata Distribution

Several MPP designs have used dynamic hierarchies to avoid the fixed-home approach used in traditional directory-based MPPs. The KSR1 [Rosti et al., 1993] machine, based on the DDM proposal [Hagersten et al., 1992], avoids associating data with fixed home nodes. Instead, data may be stored in any cache, and a hierarchy of directories allows any node to locate any data by searching successively higher and more globally-complete directories. Although this approach could be adapted to file systems [Blaze, 1993, Dahlin et al., 1994], a manager-map-based approach to distribution is superior for three reasons. First, it eliminates the “root” manager that must track all data; such a root would bottleneck performance and reduce availability. Second, the manager map allows a client to locate a file’s manager with at most one network hop. Finally, the manager map approach can be integrated more readily with the imap data structure that tracks disk location metadata.

5.6. Conclusions

This chapter described the design of a distributed management policy in which each manager controls the metadata and cache consistency state for a subset of the system’s files. A level of indirection called the manager map provides flexibility to reassign files to managers as the system’s configuration changes, and on-disk checkpoints and deltas allow the system to be resilient to failures when used in conjunction with the redundant, log-structured storage architecture described in Chapter 4.

Surprisingly, this distribution and indirection does not increase the number of network hops needed to contact the manager compared to a central-server approach. In fact, by co-locating a file’s manager with the client using that file, distributed management actually increases locality.

6 Security

A serverless system derives its scalability from distributing a central server's responsibilities among a large number of machines. This approach raises security issues. If any of a serverless system's machines are compromised, the intruder may be able to read or modify the file system's data without authorization. Although similar security issues arise in client-server architectures, trust is a particular concern in a serverless system because serverless systems place more responsibility and trust on more nodes than do traditional, central-server protocols. Distributing responsibilities improves performance, but it may increase the opportunity for a mischievous machine to interfere with the system.

The serverless design presented in this dissertation was designed for uniform security environments where all machines are administered as a unit. The serverless design will work best in such an environment, where the system can take full advantage of all of the machines' resources by using all machines as clients, managers, and storage servers.

One example of such an environment is a Network of Workstations (NOW) [Anderson et al., 1995]. Another is a non-NOW cluster of workstations that is administered uniformly. This level of trust might physically span an office, lab, floor, or building or might follow an organization's functional divisions, existing within groups, departments, or across an entire company. A serverless file system can also be used within a massively parallel processor (MPP) such as a Thinking Machines CM-5, Cray T3D, Intel Paragon, or IBM SP-2. Each node of these machines resembles a workstation in a NOW, with a processor and a significant amount of memory; some nodes also have disks and can be used as storage servers. Further, the level of trust among these nodes is typically even higher than among NOW workstations. A serverless design may also be an effective way to supply file service across the "scalable servers" currently being researched [Lenoski et al., 1990, Kubiatowicz and Agarwal, 1993, Kuskin et al., 1994]. Although these scalable servers try to bridge the gap between MPPs and bus-based shared-memory multiprocessors by adding hardware to support directory-based shared memory abstractions, their nodes otherwise resemble the nodes of an MPP or NOW in both capabilities and trust.

Note that while serverless systems place a high level of trust on the kernels of all of their nodes, they do not need to trust all users. Like traditional systems, a serverless system trusts the kernel to enforce a firewall between untrusted user processes and kernel subsystems such as the file system, allowing user processes to communicate with kernel subsystems only through well-defined and protected interfaces. The system’s storage servers, managers, and clients can then enforce standard file system security semantics. Examples of operating systems that provide this type of protection include Unix, VMS, and Windows NT. This approach, however, would not work for operating systems such as DOS and Windows95 that do not provide address space protection.

Although the complete serverless system can be used in many environments, some environments do not trust all of their machines. For instance, in a file system serving students’ personal computers in a dormitory, because users have complete control over the administration of their personal machines, they might trust only their own machines and a set of secure file servers administered by the university. Central server architectures deal with such environments by trusting servers and distrusting client machines.

By altering the serverless protocol, a serverless architecture can also be used in these mixed-trust environments. However, as machines trust one another less, it becomes more difficult for them to share their resources, and performance is reduced. This chapter describes how to extend the serverless protocol to allow untrusted clients; however, the design still requires trusted managers and storage servers. The “core” of managers and storage servers act as a scalable, high-performance, and highly-available file server for the “fringe” untrusted clients.

The rest of this chapter examines the security issues that arise when using the serverless design to provide file system service to untrusted machines. First, it discusses the security assumptions made in the serverless design and compares those assumptions to those of two central-server network file systems, NFS [Sandberg et al., 1985] and AFS [Howard et al., 1988]. Then, it describes modifications of the basic serverless design that allow the approach to be used in settings where some machines are trusted and some are not.

6.1. Security Assumptions

Security is a complex issue involving trade-offs between usability and security, and no system can be completely secure [Denning and Denning, 1979]. Rather than try to address all aspects of security in this chapter, I compare the security assumptions of the serverless design to those of two traditional, centralized network file systems, NFS and AFS.

Table 6-1 compares the security assumptions of a serverless system with those of NFS and AFS. There are three basic resources that can be compromised in any of these systems. If a user circumvents the kernel's protection model on a machine (for instance, by learning the root password), that user gains complete control over that machine. If a user compromises the network (for instance, by compromising the kernel of a machine on the network or attaching an unauthorized machine to the network), that user may be able to read or even change any unencrypted network packets. The impact of a compromised network depends on the network topology — bus-based networks such as Ethernet are more vulnerable than switch-based networks such as ATM. Finally, users with physical access to a machine may be able to compromise its security by, for instance,

		Kernel Compromised			**Network Link Compromised**			**Broadcast Network Compromised**			**Physical Security Compromised**		
		Local Users	**Remote Users**	**Remote Root**	**Local Users**	**Remote Users**	**Remote Root**	**Local Users**	**Remote Users**	**Remote Root**	**Local Users**	**Remote Users**	**Remote Root**
NFS	Client	✗	✗		✗	✗		✗	✗	✗			
	Server	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
AFS	Client	✗			✗			✗	✗	✗			
	Server	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Serverless	Client	✗	✗	✗	✗	✗	✗	✗	✗	✗			
	Manager	✗	✗	✗	✗	✗	✗	✗	✗	✗			
	Storage Server	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Untrusted Client	✗			✗			✗	✗	✗			

TABLE 6-1. **Direct security impact.** Summary of direct security impact of cracking the kernel, network link, broadcast network, or physically accessing the machine. An ✗ in a column indicates that the specified entity is compromised if the resource is compromised. For instance, if an NFS client's kernel is compromised, then all non-root users' files in the exported file system are compromised, but the remote root files are not. Note that compromising one of these resources often makes it easy to compromise others. The Serverless Untrusted Client line indicates the vulnerability assuming the modifications described in Section 6.2.

physically removing the disk that stores file system data and attaching that disk to another machine for unrestricted access.

Although I have listed the three resources separately, they are closely related; compromising one often makes it easy to compromise others. For instance, if I can physically access a computer, I can often compromise its kernel by attaching a new disk and booting my kernel or compromise its network by attaching a new computer to its network tap. If I can compromise a machine's kernel, I can compromise its network by snooping or inserting packets. If I compromise a machine's network, I can compromise its kernel by supplying modified executables over the network [Brewer et al., 1995].

Table 6-1 summarizes the direct impacts of different types of break-in. First, some attacks gain no advantage; for instance, physically removing an NFS client's disk does not directly compromise NFS data, because the system does not store data on client disks. Second, attacks on client machines often compromise data accessed by that client or the data of any user that logs onto that client machine because that machine can issue commands to the rest of the system using those users' credentials. Third, an attack can compromise all but a privileged subset of the file system's data. For instance, NFS can be configured so that client machines cannot access files owned by the server's root account, so compromising an NFS client compromises all files except those owned by root. Finally, some attacks such as compromising the kernel on an NFS server, AFS server, or serverless manager lay essentially the entire file system open for abuse.

6.1.1. Compromising the Kernel

Most operating systems enforce a protection "firewall" between users' processes and privileged subsystems such as the file system, allowing the kernel to control users' access to system resources. A broad range of attacks designed to circumvent this barrier exist, including password guessing, trojan horses, modifying the kernel by compromising the network or disk, and taking advantage of kernel bugs. Denning and Denning [Denning and Denning, 1979] and Bellovin [Bellovin, 1992] provide overviews of some general techniques, and Haynes and Kelly [Haynes and Kelly, 1992] examine the issues in the context of file system security.

6.1.1.1. Compromising Client Kernels

If the kernel on a client machine is compromised in any of the file systems, the data of any user that runs processes on that machine are at risk because the malicious kernel can issue spurious commands using the credentials provided by the user.

AFS limits vulnerability to users that access the compromised machine by preventing a compromised client from issuing file system requests in the name of users who have not logged on to that system. The file server only honors requests made by *authenticated users* [Satyanarayanan, 1989]. In an NFS or serverless system, in contrast, a compromised client can issue requests in the name of any user in the system, although NFS protects files owned by root from this type of attack by remapping root identifiers at clients to an unprivileged account at the servers. Both NFS and serverless protocols can be modified to incorporate AFS-style authentication to improve security [Steiner et al., 1988, Haynes and Kelly, 1992].

The serverless client architecture raises three additional security concerns because clients write data to storage servers, read data from storage servers, and participate in cooperative caching. Section 6.2.2 discusses the impact of these aspects of the architecture and explains how to make the serverless clients' security properties as good as those of traditional clients.

6.1.2. Compromising Server, Manager, and Storage Server Kernels

The correct operation of the kernels of the NFS and AFS servers as well as the kernels of the serverless managers and storage servers are critical to security. If a kernel is compromised, the intruder gains essentially unlimited access to all file system state.

In NFS and AFS the central server controls all of the file system's data and all client accesses to data. Therefore, if the central server's kernel is compromised, the entire file system is jeopardized.

In a serverless system the managers and storage servers play roles similar to the NFS and AFS servers. The managers control the file system's metadata and enforce access restrictions on clients. A damaged manager might therefore allow anyone to read or write any data in the system. The storage servers store the system's persistent state on their disks, so a modified storage server can read or write any data stored on its disks.

A potential advantage of NFS and AFS is that only one machine (the central server) must be trusted, while in a serverless system all managers and storage servers must run on trusted machines. However, a serverless system allows installations to vary the number of machines with this level of trust and thereby vary the level of performance in the system; as a serverless system trusts more machines to act as storage servers and managers, its performance increases. Conversely, if only a few machines can be trusted, the system can use only those machines to act as storage servers and managers.

6.1.3. Compromising the Network

Attacking the network is an effective way to compromise all three network file systems. The impact of such an attack depends on the network topology, which determines which machines are compromised when different portions of the network are compromised. If a link is compromised in a point-to-point, switched network such as ATM or Myrinet, packets to or from the machines that use that link are vulnerable. In a broadcast network such as Ethernet, all packets are in danger if an intruder has access to the broadcast medium. The simplest way for an intruder to gain access to a network is to compromise a machine attached to the network; another approach is to attach a new machine to the network.

An intruder that has compromised the network can passively read the packets sent across the network. Because none of the file systems examined encrypts data sent across the network, all data crossing the compromised section of the network are vulnerable.

A more active intruder can forge file system requests or replies to actively read data from the system or to modify data. By forging client requests, the intruder can read or write any data to which that client has access. By forging server replies, the intruder can falsify any data accessed by clients.

Although AFS and Kerberized NFS attempt to solve these problems, for performance reasons their authentication schemes do not protect the contents of packets. They do not encrypt data blocks sent over the network, so network snooping can still read all data sent over the network. They do not encrypt the contents of RPC requests or replies, so a compromised network can still forge requests or replies. They do, however, reduce the range of requests that can be forged because their servers verify that the user sending a request from a particular machine is actually logged onto that machine. Forgeries must, therefore, appear to come from a current user/machine pair. This level of security can be breached, but it provides a useful safeguard against “casual” intruders who might otherwise access other users’ files by using their personal machines to issue commands in other users’ names.

Although NFS and AFS protect their systems’ most sensitive files by disallowing all privileged root-account accesses from the network, this safeguard adds less to security than it would first appear. A compromised network can forge any data read by a client, even if the version of the data stored at the server remains unmodified. An intruder that can forge network packets can, therefore, modify even these protected files when they are read over the network by clients [Brewer et al., 1995]. From a client’s point of view, the entire file system, including protected root-owned files, is compromised. For example, an intruder can capture passwords by providing a fake version of the login executable to the client.

Further, as software implementations of encryption become faster, it will become feasible to encrypt all data sent over the network. This will reduce the vulnerability of all of these network file systems to these types of attack.

6.1.4. Physical Access to a Machine

In addition to making it easier to compromise the kernel or network of a machine (with the consequences described above), physical access to a machine also allows access to its disks. A malicious user can remove a disk to damage the file system’s data or to connect the disk to a new machine in order to read or modify the file system’s data.

Because NFS clients, serverless clients, and serverless managers do not access their local disks, this peril does not directly affect them. Conversely, NFS servers, AFS servers, and serverless storage servers store their system’s data on their disks and must be concerned with this type of attack. Further, AFS clients cache data on their local disks, making the cached data vulnerable to this type of assault.

6.2. Untrusted Clients in Mixed-Trust Environments

In some environments, not all machines will be trusted to perform all of the functions required by a serverless system. This section first explores allowing untrusted clients to use traditional client protocols such as NFS, Kerberized NFS, and AFS to access data stored by trusted, serverless machines. Section 6.2.2 then examines a solution with higher performance: modifying the client portion of the serverless protocol to make cooperative caching and log reconstruction safe.

6.2.1. Using the Serverless System as a Scalable Server

When a system uses protocols that do not require that clients be trusted and restricts storage servers and managers to a subset of trusted machines, the trusted machines act as a "scalable server" for the clients. The architecture resembles a traditional client-server one with a group of serverless machines acting as a traditional — though scalable, reliable, and cost effective — file server. If the storage servers and managers run only on trusted machines that are managed like traditional servers, such a system provides the same level of security as a traditional, centralized file system.

The approach offers two additional advantages. First, it allows commodity workstation clients to use the industry-standard NFS protocol to benefit from many of the advantages of serverlessness without changing any part of their operating systems. Second, the NFS and AFS protocols may work better than the serverless protocols for clients with slow (e.g. Ethernet-speed) network connections, because they do not require tight cooperation among clients. An economical use of this technology might be to build fast, serverless cores using high-end machine-room networks but to leave desktop machines connected to slower networks.

Figure 6-1 illustrates an installation in which a serverless core of machines exports file service to untrusted fringe clients via the NFS protocol. To use this system, an NFS client employs the same procedures it would use to mount a standard NFS partition, but instead of contacting a traditional NFS server, it contacts any one of the serverless clients. The serverless client then acts as an NFS server for the NFS client, providing high performance by employing the remaining core

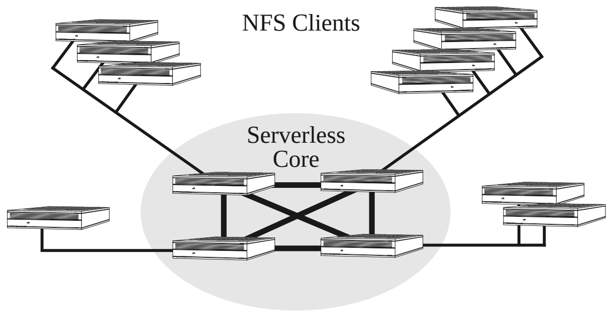


FIGURE 6-1. **A serverless core.** The serverless core acts as a scalable file server for unmodified NFS clients.

machines to satisfy requests not satisfied by its local cache. Multiple NFS clients can utilize the core as a scalable file server by having different NFS clients mount the file system from different core clients to avoid bottlenecks. Because the serverless system provides single-machine sharing semantics, it appears to the NFS clients that they are mounting the same file system from the same server. The NFS clients also benefit from the serverless core's high availability because they can mount the file system using any available core client.

Figure 6-2 illustrates a similar scenario, where several administrative *cells* work together to service a single, global file system hierarchy. Different machines from the same cell fully trust one another, but machines from different cells have less trust. Such a situation might exist among different research groups within the Berkeley Computer Science Department, for instance. The NOW group's machines all have the same administrator and root password, and the members of the NOW research group trust those machine as file servers. Similar levels of trust exist within the Daedalus group, the Plateau group, and the Tenet group. Each of these cells can use its machines to provide serverless file service and can use the full serverless protocol within cells for best performance. Users from one group might sometimes use machines from other groups, so each machine in a cell mounts the other cells' file systems using NFS or some other untrusted-client protocol so that all machines present a uniform, global name space.

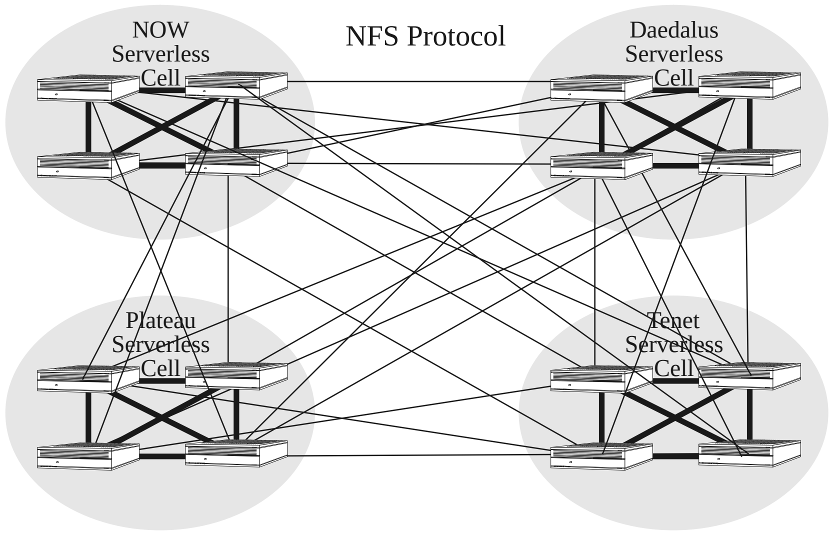


FIGURE 6-2. **Four administrative cells.** Each uses the serverless protocol for file access within the cell, but uses the NFS protocol for access to other cells' files.

6.2.2. Safe Serverless Client Protocol

Although traditional client protocols such as NFS and AFS can be used to access a serverless core, the NFS and AFS client protocols reduce performance compared to the serverless client protocol for two reasons. First, there are additional forwarding delays because the NFS and AFS clients access the serverless system by using a serverless core client as an NFS or AFS server. This client adds a delay because all data transmissions between the traditional clients and the serverless core are staged through that serverless client. For instance, if an NFS client reads a block that is not located in the cache of the serverless client that is acting as its NFS server, the serverless client reads the block from a storage server or other serverless client, and then it forwards the block to the NFS client. The second disadvantage is that the serverless system can not exploit the memory of the traditional clients for use in the cooperative cache. The rest of this section describes how to adapt the serverless client protocol for use by untrusted clients so that they can realize nearly the same performance as trusted serverless clients.

Serverless clients differ from traditional clients in three ways that affect security. First, they write data to logs that are stored directly on the storage servers. Second, they read data from the storage server logs during normal operation and recovery. Third, they participate in cooperative caching. The rest of this section discusses issues raised by each of these aspects of the design and describes how to maintain security while retaining the serverless architecture.

6.2.2.1. Client Writes to Storage Server Logs

Although serverless clients write data directly into the storage server logs, this capability does not compromise security because managers prevent unauthorized writes from being observed by any but the compromised client. If a client tries to modify a file’s data by writing a new version of the file to the log without permission, the manager refuses to update the disk-location metadata for that file, and the new version of the data has no effect on the system. The manager also issues a “reject delta” to its log to allow the cleaners to free the space consumed by the bogus data [Hartman and Ousterhout, 1995].

Just as the managers in a serverless system control the data-location metadata for security, they also control the file-attribute metadata such as file ownership and access permissions. Although performance arguments can be made for storing file-attribute metadata in either the file directories (written by clients) or in index nodes (written by managers), the latter approach is more secure. Manager control of file-attribute metadata prevents a compromised client from, for instance, creating a dangerous executable and then changing its attributes to be “set-uid root” so that it is executed with all of the capabilities of the root account.

6.2.2.2. Client Reads from Storage Servers

Clients read data from storage servers in three different ways. First, during normal operation, they read blocks from storage servers. Second, when a storage server fails, they read blocks by reading other blocks and parity from the same segment. Finally, during recovery, clients roll-forward logs to replay reads that occurred between the time of a checkpoint and the crash.

During normal operation, clients ask managers for data and the managers forward the clients’ reads to the correct storage servers. Because clients never access storage servers directly, they cannot read data without authorization. This approach provides better security than the Zebra system in which clients issue arbitrary read requests directly to storage servers.

If a client reads data stored on a failed storage server, instead of supplying the data that a client requests, the system supplies other blocks of data and parity so that the client can reconstruct the missing block. This is a potential security hole because the client may receive blocks that it does not have permission to read. A better approach, which I plan to implement in the future, is to reconstruct lost data at one of the remaining storage servers rather than at the client.

Finally, clients read the logs directly during roll-forward. This raises two concerns. First, if a client rolls forward a log that was written by some other client or one containing multiple clients’ writes combined together by a cleaner, it may read data without permission. To avoid this, the system can restrict roll forward to trusted machines only. Second, the deltas in the log may contain unauthorized modifications to the system. Managers should, therefore, check permissions for actions requested during roll-forward just as they verify client requests during normal operation.

6.2.2.3. Cooperative Caching

Cooperative caching introduces two concerns about client security. First, a client might modify a block before forwarding it to another client. Second, a client might allow unauthorized local reads to data that have been forwarded to it. These risks can be addressed in either of two ways: by restricting which clients cache data cooperatively or by using encryption-based techniques. Combinations of these techniques may be the most practical approach.

Restricting Cooperative Caching to Trusted Machines

A simple strategy to make cooperative caching safe is to only use trusted clients' memories for cooperative caching. Untrusted clients could still benefit from cooperative caching when blocks are forwarded to them, but because they would never forward blocks themselves, they could not modify data being sent to other clients. Further, because the system would not forward data to such clients for cooperative caching storage, they would not be able to issue unauthorized reads. Of course, the system could not benefit from the untrusted clients' memory capacity.

Another strategy that protects only against unauthorized read is to restrict untrusted clients to greedy cooperative caching. In that case, the contents of untrusted clients' memories only include data that they have read on behalf of authorized users via the traditional client interface. For full protection, this strategy for protecting reads can be combined with the encryption-based strategy described below for protecting writes.

Encryption-Based Cooperative Caching Security

Cooperative caching protected via encryption-based techniques can exploit untrusted memories, paying additional CPU overheads to prevent clients from transgressing. As Table 6-2 shows, modern CPUs can encrypt data (to guard against unauthorized reads) or compute digests (to guard against unauthorized writes) quickly; because these operations are CPU limited, the technology trends discussed in Chapter 2 will make this approach even more attractive in the future. In addi-

		Digests		**Encryption**	
	SPECInt92	**MD4**	**MD5**	**DES**	**WAKE**
HP 715/80	65	8.2 MB/s	5.9 MB/s	3.7 MB/s	9.5 MB/s
HP 735/99	80	8.6 MB/s	6.0 MB/s	4.5 MB/s	11.4 MB/s
HP 9000/J200	139	10.6 MB/s	7.6 MB/s	4.9 MB/s	12.27 MB/s
SUN SS2	22	2.5 MB/s	1.9 MB/s	0.8 MB/s	N/A
SUN SS10/51	65	6.3 MB/s	4.7 MB/s	1.6 MB/s	8.2 MB/s
SUN SS20/51	77	6.4 MB/s	4.7 MB/s	1.6 MB/s	8.6 MB/s
DEC AXP 3000/400	75	7.4 MB/s	5.1 MB/s	N/A	N/A

TABLE 6-2. **Encryption and message digest performance.** Performance was measured for several algorithms on several machines. All performance figures indicate the bandwidth to encrypt or compute the message digest of an 8 KB block of data; if larger blocks were used, all bandwidths would be significantly higher. The digest algorithms are Rivest's MD4 [Rivest, 1992a] and MD5 [Rivest, 1992b], the Digital Encryption Standard algorithm (DES), and Wheeler's WAKE [Wheeler, 1993].

tion to protecting cooperative caching, widespread, fast encryption may protect other aspects of distributed file systems such as network communication.

Encryption prevents an unauthorized client from reading data stored in its cooperative cache. Figure 6-3 illustrates how a client can encrypt and forward data to another client's cache as it would for the N-Chance algorithm. When a client makes room in its cache by forwarding data to another client, it first encrypts the data using a private key; clients issue a different key for each block they encrypt. In addition to forwarding the encrypted data to the remote cache, the client sends the key to the manager, which stores the key with the block's cache consistency state. If the manager forwards a read request to the client holding the encrypted data, that client sends the encrypted data to the reader, and the manager sends the key to the reader. The reader can decrypt the data using the key and then store the data in its own cache.

Another encryption-based technique, message digests, can protect against unauthorized data modifications. Message digests provide a secure checksum for data: given a block and its message digest, it is computationally infeasible to devise another block with the same digest. As Figure 6-4 illustrates, when a client loses write ownership of a block it has modified, it computes a new digest

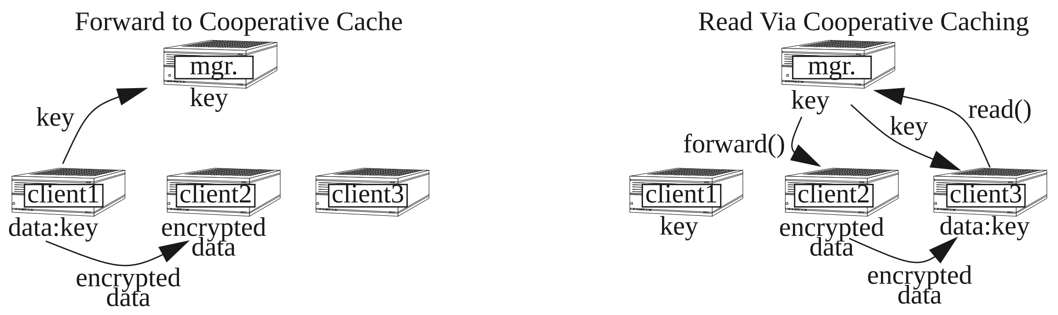


FIGURE 6-3. **Encryption.** Cooperative caching when client 2 is not authorized to read a block of data. The left picture indicates how client 1 encrypts data before sending it to client 2. The right picture shows how client 2 forwards this encrypted data, which, when combined with the encryption key sent by the server, satisfies client 3's read request.

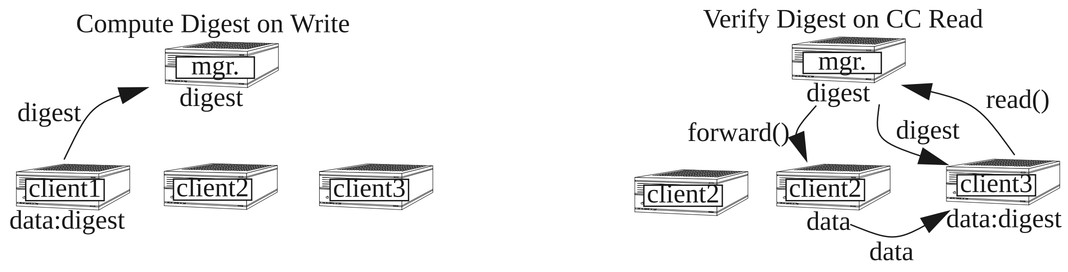


FIGURE 6-4. **Message digests.** The security protocol uses message digests to verify the integrity of data supplied via cooperative caching. Clients compute digests and send them to their managers when they lose write ownership of a block. Later, when another client reads the block via cooperative caching, it verifies the data by computing its digest and comparing the digest to the digest supplied by the manager.

and sends its digest to the manager. The managers thus always have current digests for all cache blocks that are forwarded via cooperative caching. Digests can be combined with encryption or greedy caching to provide protection from both unauthorized reads and unauthorized writes.

6.3. Summary

The serverless design will be most effective when machines trust one another equally. In that case, machines can take full advantage of one anothers' resources to provide file service. If not all machines are trusted, more restrictive versions of the serverless protocol can be used. In these protocols, the serverless clients are replaced with clients with the same security requirements as traditional file system clients, and the serverless managers and storage servers execute only on trusted machines, in the same way that traditional systems' servers do.

7 xFS Prototype

To investigate the pieces of serverless design described in the previous chapters, this chapter examines xFS, a prototype serverless network file system. The construction and measurement of the xFS prototype has been a joint project with rest of the Berkeley xFS group: Jeanna M. Neefe, Drew S. Roselli, and Randolph Y. Wang.

The xFS prototype integrates cooperative caching, serverless storage, and serverless management to realize its goal of location independence: the ability to put "anything, anywhere." It seeks to distribute all data, metadata, and control throughout the system and to allow them to be dynamically migrated during operation. It attempts to exploit this location independence to improve performance by taking advantage of all of the system's resources — CPUs, DRAM, and disks — to distribute load and increase locality. Finally, it aims to use location independence to provide high availability by allowing any machine to take over the responsibilities of a failed component after recovering its state from the redundant log-structured storage system.

This chapter first describes how cooperative caching, distributed storage, and distributed management fit together to form xFS. Next, it describes the xFS prototype and presents initial performance results. Finally, it summarizes the conclusions that can be drawn from these preliminary results and discusses future directions.

7.1. Serverless File Service

The xFS prototype brings together cooperative caching, serverless storage, and serverless management to replace the functionality of a traditional central server. In a typical, centralized system, the central server has four main tasks:

1. The server stores all of the system's data blocks on its local disks.
2. The server manages disk location metadata that indicate where on disk the system has stored each data block.

3. The server maintains a central cache of data blocks in its memory to satisfy some client misses without accessing its disks.
4. The server manages cache consistency metadata that lists which clients in the system are caching each block. It uses this metadata to invalidate stale data in client caches.

The xFS system performs the same tasks, but it builds on the ideas discussed in this dissertation to distribute this work over all of the machines in system. xFS replaces the server cache with cooperative caching that forwards data among client caches under the control of the managers as described in Chapter 3. Similarly, to provide scalable disk storage, xFS uses log-based network striping with distributed cleaners as Chapter 4 described. Finally, to provide scalable control of disk metadata and cache consistency state, xFS uses serverless management techniques like those discussed in Chapter 5. In xFS, four types of entities — the clients, storage servers, cleaners, and managers cooperate to provide file service as Figure 7-1 illustrates.

7.2. xFS Prototype

The xFS prototype implementation runs on a cluster of 32 SPARCStation 10's and 20's. A small amount of code runs as a loadable module for the Solaris kernel. This code provides xFS's interface to the Solaris v-node layer and kernel buffer cache. The remaining pieces of xFS run as daemons outside of the kernel address space to facilitate debugging [Howard et al., 1988]. If the xFS kernel

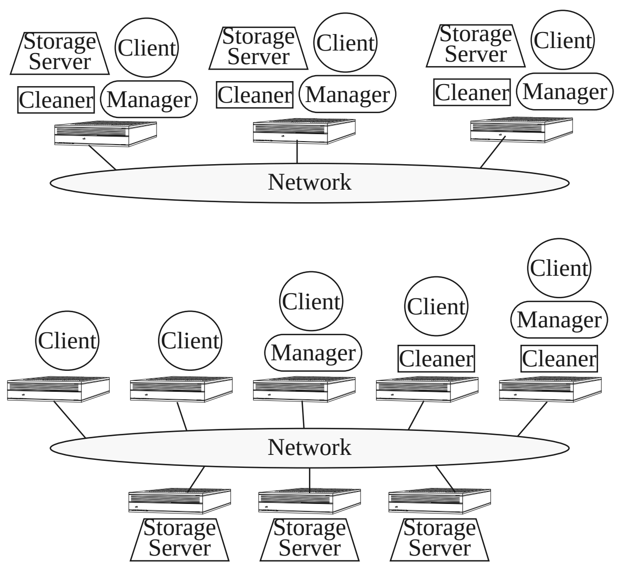


FIGURE 7-1. **Two simple xFS installations.** In the first, each machine acts as a client, storage server, cleaner, and manager, while in the second each node only performs some of those roles. The freedom to configure the system is not complete; managers and cleaners access storage using the client interface, so all machines acting as managers or cleaners must also be clients.

module cannot satisfy a request using the buffer cache, then it sends the request to the client daemon. The client daemons provide the rest of xFS’s functionality by accessing the manager, storage server, and cleaner daemons over the network.

The rest of this section summarizes the status of the prototype as of October 1995 and describes the prototype’s hardware and software environment.

7.2.1. Prototype Status

The xFS prototype implements most of the key features of a serverless system, including distributed management, cooperative caching, and network disk striping with single parity and multiple groups. Several key features, however, remain to be implemented. The most glaring deficiencies are in crash recovery and cleaning. Although the implementation supports storage server recovery, including automatic reconstruction of data from parity, it does not implement manager state checkpoint and roll forward; also, it does not include the consensus algorithms necessary to calculate and distribute new manager maps and stripe group maps; the system currently reads these mappings from a non-xFS file and cannot change them. Additionally, the system does not have a cleaner yet. As a result, xFS is still best characterized as a research prototype, and the results in this chapter should thus be viewed as evidence that the serverless approach is promising, not as “proof” that it will succeed.

7.2.2. Test Environment

The testbed includes a total of 32 machines: eight dual-processor SPARCStation 20’s, and 24 single-processor SPARCStation 10’s. Each of the machines has 64 MB of physical memory. Uniprocessor 50 MHz SS-20’s and SS-10’s have SPECInt92 ratings of 74 and 65, and can copy large blocks of data from memory to memory at 27 MB/s and 20 MB/s, respectively. For the xFS tests, all machines act as storage servers, managers, and clients unless otherwise noted. For experiments using fewer than 32 machines, I always include all of the SS-20’s before starting to use the less powerful SS-10’s.

Each xFS storage server stores data on a 256 MB partition of a 1.1 GB Seagate-ST11200N disk. These disks have an advertised average seek time of 10.5 ms and rotate at 5,411 RPM. I measured a 2.7 MB/s peak bandwidth to read from the raw disk device into memory. For all xFS tests, the system uses a log fragment size of 64 KB, and unless otherwise noted it uses storage server groups of eight machines — seven for data and one for parity; all xFS tests include the overhead of parity computation.

A high-speed, switched Myrinet network [Boden et al., 1995] connects the machines. Although each link of the physical network has a peak bandwidth of 80 MB/s, RPC and TCP/IP protocol overheads place a much lower limit on the throughput actually achieved [Keeton et al., 1995]. The throughput for fast networks such as the Myrinet depends heavily on the version and patch level of the Solaris operating system used. For my xFS measurements, I use a kernel that I compiled from the Solaris 2.4 source release. I measured the TCP throughput to be 3.2 MB/s for 8 KB packets when using this source release. The binary release of Solaris 2.4, augmented with the binary patches recommended by Sun as of June 1, 1995 provides higher performance; the TCP test achieves a throughput of 8.4 MB/s for this setup. Alas, I could not get sources for the patches, so my xFS measurements are penalized with a slower effective network than the NFS and AFS measurements described below. RPC overheads further reduce network performance.

7.2.3. NFS and AFS Environments

I use the same hardware to compare xFS with two central-server architectures, NFS [Sandberg et al., 1985] and AFS (a commercial version of the Andrew file system [Howard et al., 1988]). I use NFS as my baseline system for practical reasons — NFS is mature, widely available, and well-tuned, allowing easy comparison and a good frame of reference — but its limitations with respect to scalability are well known [Howard et al., 1988]. Since many NFS installations have attacked NFS's limitations by buying shared-memory multiprocessor servers, I would like to compare xFS running on workstations to NFS running on a large multiprocessor server, but such a machine was not available to me, so my NFS server runs on essentially the same platform as the clients. I also compare xFS to AFS, a more scalable central-server architecture. However, AFS achieves most of its scalability compared to NFS by improving cache performance; its scalability is only modestly better compared to NFS for reads from server disk and for writes.

For my NFS and AFS tests, one of the SS-20's acts as the central server, using a larger and somewhat faster disk than the xFS storage servers: a 2.1 GB DEC RZ 28-VA with a peak bandwidth of 5 MB/s from the raw partition into memory. These servers also use a Prestoserve NVRAM card that acts as a buffer for disk writes [Baker et al., 1992]. The xFS machines did not use NVRAM buffers, but their log buffers provide similar performance benefits.

For local disk caches, the AFS clients use a 100 MB partition of the same Seagate ST11200N disks used by the xFS storage servers.

The NFS and AFS tests run on the unmodified Solaris kernel, so I use the patched-binary Solaris release for them. Under this kernel release, the network TCP tests indicate a maximum 8.4 MB/s throughput between one client and the server for the Myrinet network.

7.2.4. Performance Results

This section presents preliminary performance results for xFS under a set of microbenchmarks designed to stress file system scalability and under an application-level benchmark. Although these results are preliminary and although I expect future tuning to significantly improve absolute performance, they suggest that xFS has achieved its goal of scalability. For instance, in one of the microbenchmarks 32 clients achieved an aggregate large file write bandwidth of 13.9 MB/s, close to a linear speedup compared to a single client's 0.6 MB/s bandwidth. The other tests indicated similar speedups for reads and small file writes.

As noted above, several significant pieces of the xFS system — manager checkpoints and cleaning — remain to be implemented. I do not expect checkpoints to limit performance. Thorough future investigation will be needed, however, to evaluate the impact of distributed cleaning under a wide range workloads; other researchers have measured sequential cleaning overheads from a few percent [Rosenblum and Ousterhout, 1992, Blackwell et al., 1995] to as much as 40% [Seltzer et al., 1995], depending on the workload.

Also, the current prototype implementation suffers from three inefficiencies, all of which will be addressed in the future:

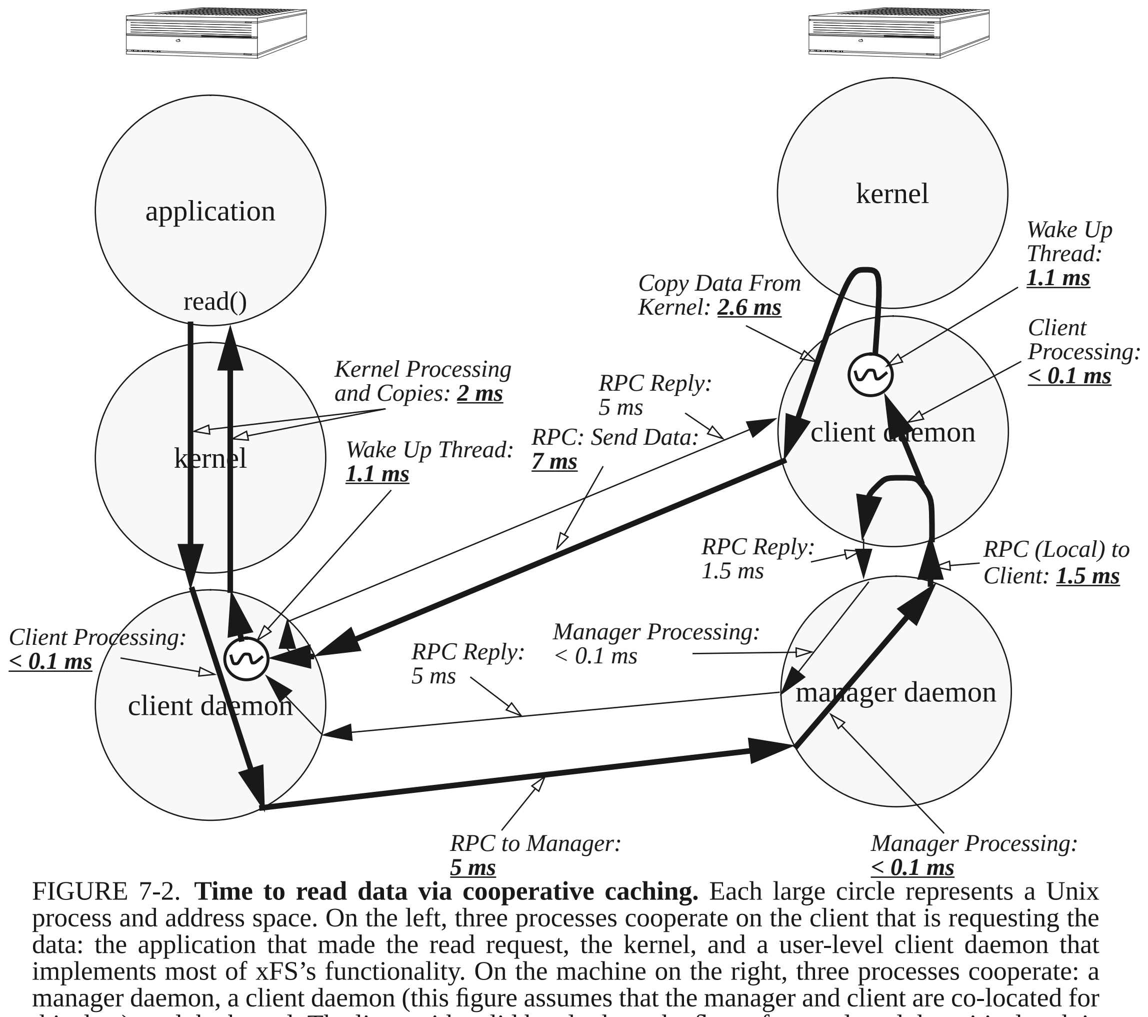
1. xFS is currently implemented by redirecting v-node calls to a set of user-level processes. This indirection hurts performance because each user/kernel space crossing requires the kernel to schedule the user level process and copy data to or from the user process's address space. The fix for this limitation is to move xFS into the kernel. (Note that AFS shares this handicap.)
2. RPC and TCP/IP overheads severely limit xFS's network performance. The fix for this limitation is to port xFS's communications layer to a faster communication system, such as Active Messages [von Eicken et al., 1992].
3. Once the first two limitations have been addressed, the system must be systematically profiled and tuned to identify and fix any other major inefficiencies.

As a result, the absolute performance is much less than I expect for the (hypothetical) well-tuned xFS. As the implementation matures, I expect a single xFS client to significantly outperform an NFS or AFS client by benefitting from the bandwidth of multiple disks and from cooperative caching. The eventual performance goal is for a single xFS client to be able to read and write data at a rate near that of its maximum network throughput, and for multiple clients to realize an aggregate bandwidth approaching the system's aggregate local disk bandwidth.

To quantify the performance of the prototype, I examine the performance of cooperative caching in detail. I then examine its scalability using a series of microbenchmarks. These microbenchmarks measure read and write throughput for large files and write performance for small files. Finally, I use Satyanarayanan's Andrew benchmark [Howard et al., 1988] as a simple evaluation of application-level performance. In the future, I plan to compare the systems' performance under more demanding applications.

7.2.5. Performance of the Prototype

Figure 7-2 details the performance of the prototype for a read request satisfied via cooperative caching. These measurements illustrate the limitations of the current implementation that were discussed above. As the summary in Table 7-1 indicates, network protocol processing causes most of the latency for cooperative caching; context switches and copies between kernel and user-level



application
read()
kernel
client daemon
kernel
client daemon
manager daemon
Kernel Processing and Copies: 2 ms
Wake Up Thread: 1.1 ms
Client Processing: < 0.1 ms
RPC Reply: 5 ms
RPC: Send Data: 7 ms
RPC Reply: 5 ms
Copy Data From Kernel: 2.6 ms
Wake Up Thread: 1.1 ms
Client Processing: < 0.1 ms
RPC Reply: 1.5 ms
RPC (Local) to Client: 1.5 ms
Manager Processing: < 0.1 ms
RPC to Manager: 5 ms
Manager Processing: < 0.1 ms

FIGURE 7-2. **Time to read data via cooperative caching.** Each large circle represents a Unix process and address space. On the left, three processes cooperate on the client that is requesting the data: the application that made the read request, the kernel, and a user-level client daemon that implements most of xFS's functionality. On the machine on the right, three processes cooperate: a manager daemon, a client daemon (this figure assumes that the manager and client are co-located for this data), and the kernel. The lines with solid heads show the flow of control, and the critical path is drawn with thick lines. The complete read() takes about 20 ms.

daemons are also significant factors. Once these overheads have been reduced, further improvements may require restructuring the code to avoid switching between threads, since each of the two signals to wakeup a new thread takes over a millisecond.

Although the performance of the current prototype falls short of its ultimate goals, these measurements identify the limiting factors and point the way towards improved performance. Feeley et al. [Feeley et al., 1995] implemented a global virtual memory system that provides performance similar to what I expect from a more mature cooperative cache implementation. This system's architecture is similar to xFS's: when one machine has a local miss, it sends a message to a global-cache-directory (GCD) that is similar to xFS's managers. The GCD forwards the request to the machine with the data, and that machine sends the data to the first machine. The implementation differs from xFS, however, in that it uses a custom lightweight communications protocol, runs in the kernel, and is more carefully tuned (although the authors note several remaining opportunities to improve performance through further tuning.) The system runs on several 225 MHz DEC Alphas running OSF/1 that communicate over a 155 Mbit/s ATM network. This implementation allows the system to fetch data from a remote client's memory in 1.5 ms.

7.2.6. Scalability

Figures 7-3 through 7-5 illustrate the scalability of xFS's performance for large writes, large reads, and small writes. For each of these tests, as the number of clients increases, so does xFS's aggregate performance. In contrast, just a few clients saturate NFS's or AFS's single server, limiting peak throughput.

	Time
Network Protocols	13.5 ms
Kernel/User Space Crossings	4.6 ms
Thread Switching	2.2 ms
Other	<0.5 ms
Total	**20 ms**

TABLE 7-1. **Breakdown of time for cooperative caching read**. Network processing accounts for the largest fraction of the latency for a cooperative caching request. The next largest component is the cost of communication between user-space and kernel-space; this cost includes the time for the kernel to intercept the vnode call, copy data between the address spaces, and schedule and activate a user-level process. The third source of inefficiency is the implementation's use of threads; each remote hit switches between user-level threads in the client daemon twice for a total cost of 2.2 ms. Manipulating data structures and other processing is a relatively minor source of delay.

Figure 7-3 illustrates the performance of the disk write throughput test, in which each client writes a large (10 MB), private file and then invokes sync() to force the data to disk (some of the blocks stay in NVRAM in the case of NFS and AFS.) A single xFS client is limited to 0.6 MB/s, about one-third of the 1.7 MB/s throughput of a single NFS client; this difference is largely due to the extra kernel crossings and associated data copies in the user-level xFS implementation as well as high network protocol overheads. A single AFS client achieves a bandwidth of 0.7 MB/s, limited by AFS's kernel crossings and overhead of writing data to both the local disk cache and the server disk. As the number of clients increases, NFS's and AFS's throughputs increase only modestly until the single, central server disk bottlenecks both systems. The xFS configuration, in contrast, scales up to a peak bandwidth of 13.9 MB/s for 32 clients, and it appears that if the prototype had more clients available for these experiments, it could achieve even more bandwidth from the 32 xFS storage servers and managers.

Figure 7-4 illustrates the performance of xFS and NFS for large reads from disk. For this test, each machine flushes its cache and then sequentially reads a per-client 10 MB file. Again, a single NFS or AFS client outperforms a single xFS client. One NFS client can read at 2.8 MB/s, and an AFS client can read at 1.0 MB/s, while the current xFS implementation limits one xFS client to 0.9 MB/s. As is the case for writes, xFS exhibits good scalability; 32 clients achieve a read throughput of 13.8 MB/s. In contrast, two clients saturate NFS at a peak throughput of 3.1 MB/s and 12 clients saturate AFS's central server disk at 1.9 MB/s.

While Figure 7-4 shows disk read performance when data are not cached, all three file systems achieve much better scalability when clients can read data from their caches to avoid interacting

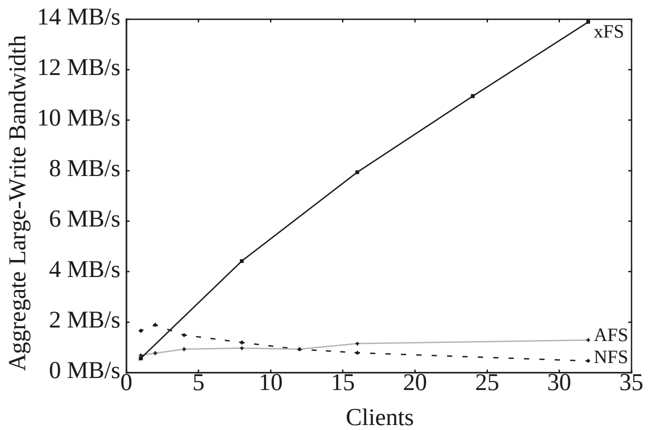


FIGURE 7-3. **Aggregate disk write bandwidth.** The x axis indicates the number of clients simultaneously writing private 10 MB files, and the y axis indicates the total throughput across all of the active clients. xFS uses four groups of eight storage servers and 32 managers. NFS's peak throughput is 1.9 MB/s with 2 clients, AFS's is 1.3 MB/s with 32 clients, and xFS's is 13.9 MB/s with 32 clients.

with the server. All three systems allow clients to cache data in local memory, providing scalable bandwidths of 20 MB/s to 30 MB/s per client when clients access working sets of a few tens of megabytes. Furthermore, AFS provides a larger, though slower, local disk cache at each client that provides scalable disk-read bandwidth for workloads whose working sets do not fit in memory; our 32-node AFS cluster can achieve an aggregate disk bandwidth of nearly 40 MB/s for such workloads. This aggregate disk bandwidth is significantly larger than xFS's maximum disk bandwidth for two reasons. First, as noted above, xFS is largely untuned, and I expect the gap to shrink in the future. Second, xFS transfers most of the data over the network, while AFS's cache accesses are local. Thus, there will be some workloads for which AFS's disk caches achieves a higher aggregate disk-read bandwidth than xFS's network storage. xFS's network striping and better load balance, however, provides better write performance and will, in the future, provide better read performance for individual clients via striping. Additionally, once cooperative caching runs under a faster network protocol, accessing remote memory will be much faster than going to local disk, and thus the clients' large, aggregate memory cache will further reduce the potential benefit from local disk caching.

Figure 7-5 illustrates the performance when each client creates 2,048 files containing 1 KB of data per file. For this benchmark, xFS's log-based architecture overcomes the current implementation limitations to achieve approximate parity with NFS and AFS for a single client: one NFS, AFS, or xFS client can create 51, 32, or 41 files per second, respectively. xFS also demonstrates good scalability for this benchmark. 32 xFS clients generate a total of 1,122 files per second, while NFS's peak rate is 91 files per second with four clients and AFS's peak is 87 files per second with four clients.

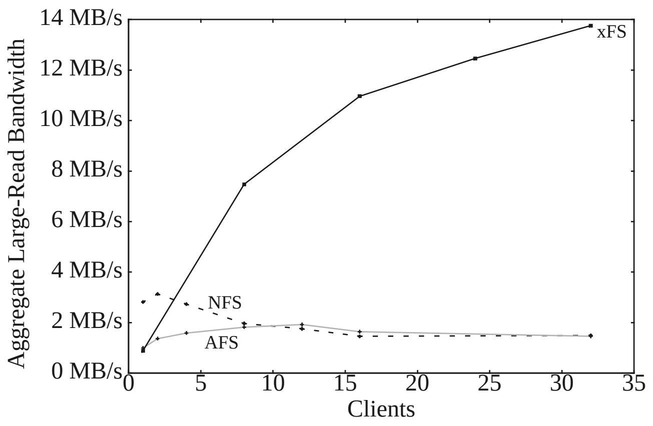


FIGURE 7-4. **Aggregate disk read bandwidth.** The x axis indicates the number of clients simultaneously reading private 10 MB files and the y axis indicates the total throughput across all active clients. xFS uses four groups of eight storage servers and 32 managers. NFS's peak throughput is 3.1 MB/s with two clients, AFS's is 1.9 MB/s with 12 clients, and xFS's is 13.8 MB/s with 32 clients.

Figure 7-6 shows the average time for a client to complete the Andrew benchmark as the number of clients varies for each file system. This benchmark was designed as a simple yardstick for comparing application-level performance for common tasks such as copying, reading, and compiling files. When one client is running the benchmark, NFS takes 64 seconds to run and AFS takes 61 seconds, while xFS requires somewhat more time — 78 seconds. xFS's scalability, however, allows xFS to outperform the other systems for larger numbers of clients. For instance, with 32 clients simultaneously running independent copies of the benchmark, xFS takes 117 seconds to complete the benchmark on average, while increased I/O time, particularly in the copy phase of the benchmark, increases NFS's time to 172 seconds and AFS's time to 210 seconds. A surprising result is that NFS outperforms AFS when there are a large number of clients; this is because in-memory file caches have grown dramatically since this comparison was first made [Howard et al., 1988], and the working set of the benchmark now fits in the NFS clients' in-memory caches, reducing the benefit of AFS's on-disk caches.

7.2.7. Storage Server Scalability

In the above measurements, I used a 32-node xFS system where all machines acted as clients, managers, and storage servers and found that both bandwidth and small write performance scaled well. This section examines the impact of different storage server organizations on that scalability. Figure 7-7 shows the large write performance as I vary the number of storage servers and also as I change the stripe group size.

Increasing the number of storage servers improves performance by spreading the system's requests across more CPUs and disks. The increase in bandwidth falls short of linear with the

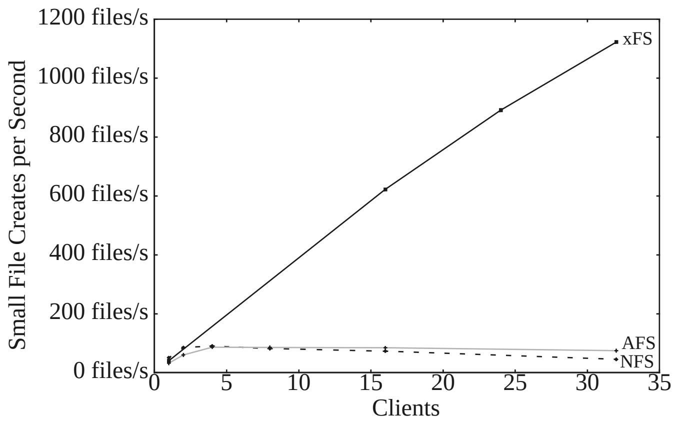


FIGURE 7-5. **Aggregate small write performance.** The x axis indicates the number of clients, each simultaneously creating 2,048 1 KB files. The y axis is the average aggregate number of file creates per second during the benchmark run. xFS uses four groups of eight storage servers and 32 managers. NFS achieves its peak throughput of 91 files per second with four clients, AFS peaks at 87 files per second with four clients, and xFS scales up to 1,122 files per second with 32 clients.

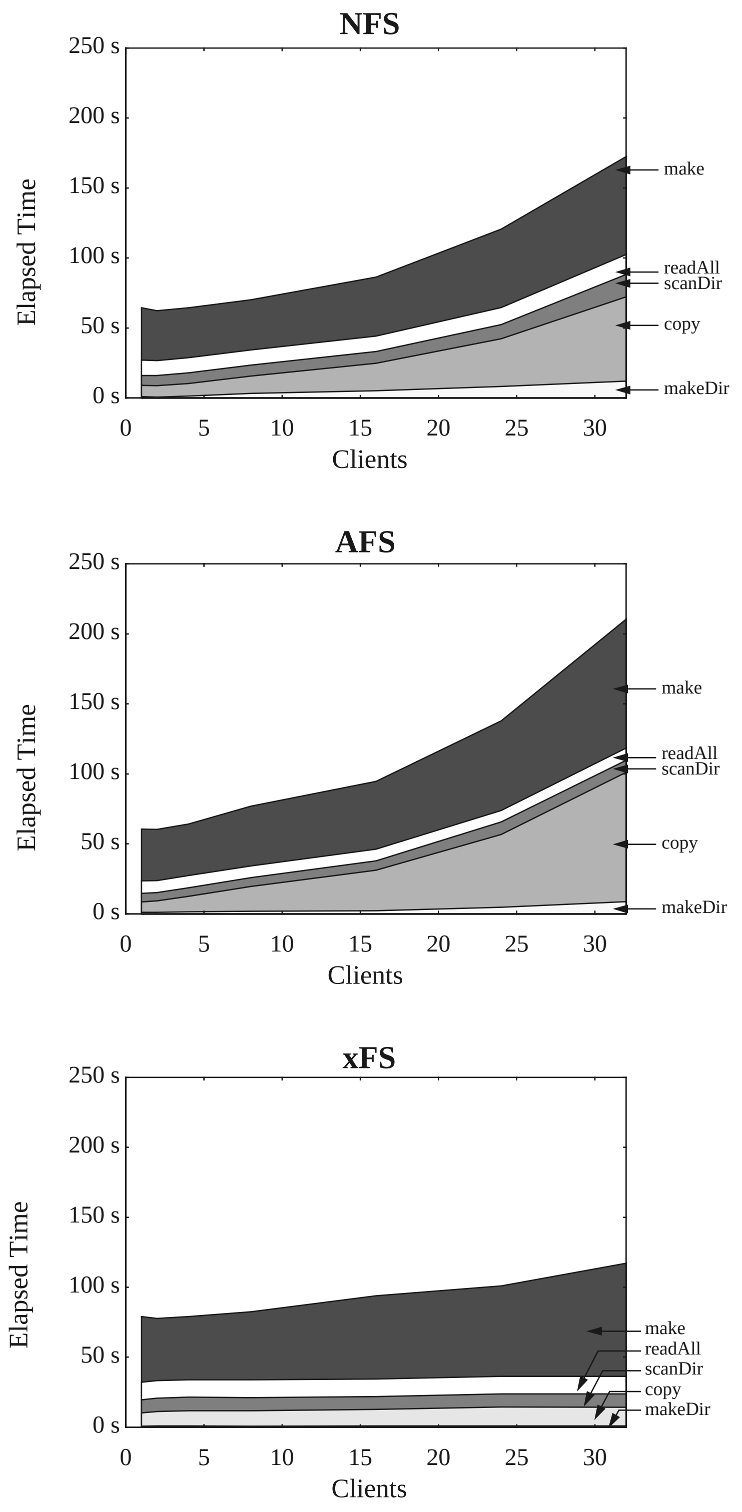


FIGURE 7-6. **Average time to complete the Andrew benchmark.** The three graphs show results for NFS, AFS, and xFS as the number of clients simultaneously executing the benchmark varies. The total height of the shaded areas represents the total time to complete the benchmark; each shaded area represents the time for one of the five phases of the benchmark: makeDir, copy, scanDir, readAll, and make. For all of the systems, the caches were flushed before running the benchmark.

number of storage servers, however, because client overheads are also a significant limitation on system bandwidth.

Reducing the stripe group size from eight storage servers to four reduces the system's aggregate bandwidth by 8% to 22% for the different measurements. I attribute most of this difference to the increased overhead of parity. Reducing the stripe group size from eight to four reduces the fraction of fragments that store data as opposed to parity. The additional overhead reduces the available disk bandwidth by 16% for the system using groups of four servers.

7.2.8. Manager Scalability

Figure 7-8 shows the importance of distributing management among multiple managers to achieve both parallelism and locality. It varies the number of managers handling metadata for 31 clients running the small write benchmark (due to a hardware failure, I ran this experiment with three groups of eight storage servers and 31 clients.) This graph indicates that a single manager is a significant bottleneck for this benchmark. Increasing the system from one manager to two increases throughput by over 80%, and a system with four managers more than doubles throughput compared to a single manager system.

Continuing to increase the number of managers in the system continues to improve performance under xFS's First Writer policy. This policy assigns files to managers running on the same machine as the clients that create the files; Section 5.3 on page 104 described this policy in more detail. The system with 31 managers can create 45% more files per second than the system with four managers under this policy. This improvement comes not from load distribution but from locality; when a

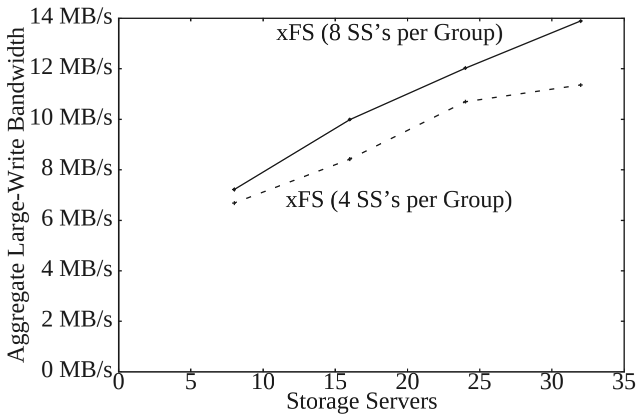


FIGURE 7-7. **Storage server throughput.** Large write throughput as a function of the number of storage servers in the system. The x axis indicates the total number of storage servers in the system and the y axis indicates the aggregate bandwidth when 32 clients each write a 10 MB file to disk. The 8 SS's line indicates performance for stripe groups of eight storage servers (the default), and the 4 SS's shows performance for groups of four storage servers.

larger fraction of the clients also host managers, the algorithm is more often able to successfully co-locate the manager of a file with the client accessing it.

The Nonlocal Manager line illustrates what would happen without locality. For this line, I altered the system's management assignment policy to avoid assigning files created by a client to the local manager. When the system has four managers, throughput peaks for this algorithm because the managers are no longer a significant bottleneck for this benchmark; larger numbers of managers do not further improve performance.

7.3. Conclusions and Future work

This chapter described how cooperative caching, serverless storage, and serverless management combine to form a completely serverless file system, and it provided an overview of a prototype system called xFS. The goal of a serverless system is to eliminate all file system bottlenecks to provide scalability, high performance, and high availability, and the initial prototype provides evidence in support of this approach.

Full validation of this approach, however, will have to wait for a more complete implementation of the prototype. The remaining work on the prototype can be divided into three major efforts:

1. Performance Improvements

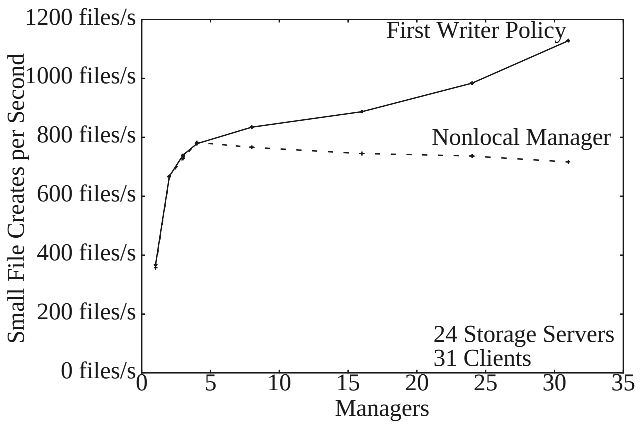


FIGURE 7-8. **Manager throughput.** Small write performance as a function of the number of managers in the system and manager locality policy. The x axis indicates the number of managers. The y axis is the average aggregate number of file creates per second by 31 clients, each simultaneously creating 2,048 small (1 KB) files. The two lines show the performance using the First Writer policy that co-locates a file's manager with the client that creates the file, and a Nonlocal policy that assigns management to some other machine. Because of a hardware failure, I ran this experiment with three groups of eight storage servers and 31 clients. The maximum point on the x-axis is 31 managers.

My hypothesis is that single client in a serverless system should be able to realize I/O bandwidth limited only by its network interface and that the system as a whole should be limited only by its aggregate disk bandwidth. Figure 7-9 compares the prototype's current performance to the ultimate performance I hope to achieve. Clearly, much remains to be done on this front. I plan to pursue three approaches to achieve this goal: replacing RPC communication with Active Messages, moving the xFS implementation into the kernel address space, and more general performance profiling and tuning.

2. Dynamic Reconfiguration

A mature serverless system will use dynamic reconfiguration both to provide high availability and to balance load. The current xFS prototype, however, does not implement reconfiguration after machine failures and it does not use reconfiguration of management to balance load. Future work will therefore be needed to test the hypotheses that a serverless system can provide better availability than a central-server system and that a serverless system can, for a wide range of loads, avoid performance bottlenecks caused by hot-spots.

3. Distributed Cleaning

The distributed cleaner design presented in Chapter 4 was designed to prevent the cleaner from bottlenecking throughput. To evaluate the effectiveness of this design, a distributed cleaner must be implemented for xFS, and that cleaner must be tested under a wide range of workloads.

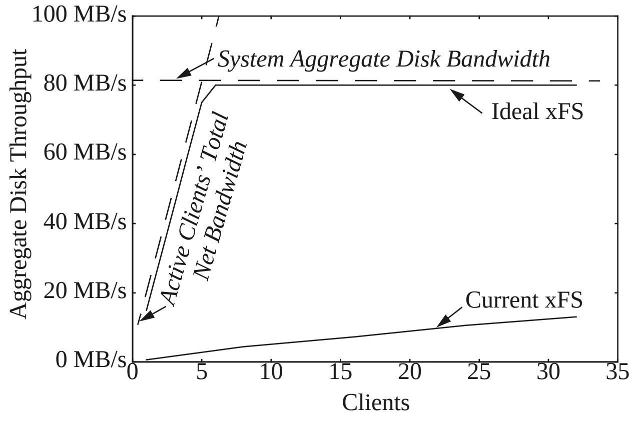


FIGURE 7-9. **Ideal performance goals.** Comparison of current xFS performance for large writes against ideal performance goals on the current experimental hardware configuration. With small numbers of clients, the system throughput should be limited only by the clients' network throughput, and with large numbers of clients, the throughput should be limited only by the system's aggregate disk throughput.

8 Conclusions

My thesis is that serverless network file systems — fully distributed file systems consisting of cooperating commodity workstations — can eliminate file system bottlenecks and scale to meet rapidly increasing demands on file systems. This dissertation presents a design that distributes the services handled by a traditional, central server by first functionally decomposing a central server's responsibilities, and then distributing each of these functions. A cooperative cache distributed among clients replaces central-server caching; log-based, redundant, network storage replaces server disks; and a distributed management protocol both provides cache consistency and locates the blocks stored on the disks.

The goals of this design are improved performance, availability, and scalability. I have evaluated the design using both simulation and measurements of a prototype called xFS.

• **Goal: Improved Performance**

The cooperative cache replaces the relatively small central-server cache with a potentially much larger cache that exploits all clients' memories. Simulation results indicate that cooperative caching improves read response time by 30% to 150% for office and engineering workloads.

The network-striped disks allow parallel data transfers for reads and writes; the goal is to provide enough bandwidth to an individual client to saturate its network bandwidth and to provide aggregate bandwidth when servicing multiple clients approaching the aggregate bandwidth of the disks in the system. Initial measurements indicate that the prototype falls short of these goals; however, it appears that the architecture will support higher performance as limitations of the implementation are addressed.

Distributed management improves performance both by eliminating the bottleneck that would be present with a single manager and by improving locality by co-locating managers with the clients using the files they manage. Simulation results suggest that locality is a larger performance factor than load distribution for office and engineering workloads, but benchmarks run on the prototype reveal the importance of both factors for some workloads.

• **Goal: Improved Availability**

The architecture bases high-availability on location independence and redundant logging of durable state. If a machine fails, another takes over its duties by recovering its state from the redundant logs. I have completed the design of serverless recovery, but future work is still needed to fully implement and validate this design. Beyond this basic approach, the use of stripe groups increases the number of storage servers over which the system can store its data by improving availability and limiting segment size.

• **Goal: Improved Scalability**

A key aspect of both the system's performance and availability goals is scalability — the system's performance should improve as the number of machines increases, and its availability should not decline. The design addresses these goals by trying to eliminate all centralized bottlenecks, and initial performance results are promising. For instance, in the 32-node prototype with 32 active clients, each client receives nearly as much read and write bandwidth as it would see if it were the only active client. As the prototype's absolute performance increases, continuing to avoid bottlenecks will be more challenging. Evaluating the prototype's scalability with respect to availability is also future work.

8.1. Lessons Learned About Research

The xFS project represents computer systems research "in the large." The complexity of this system makes it difficult to evaluate the design. On one hand, simulation alone cannot be used, because the simulator is unlikely to capture all of the important factors in the system. On the other hand, the system's complexity also makes it hard to isolate the effect of different aspects of the design when the system is running. My approach has been to use both simulation and evaluation of the prototype.

I found simulation to be valuable when comparing different policy choices for specific points of the design; however, there is a limit on how much can be learned from simulations alone, so they should be carefully designed to minimize the effort needed to examine a particular issue.

There are two dangers. First, it is tempting to put unnecessary details into a simulation. Second, it is easy to neglect important details in a simulation. Although these statements may appear to be contradictory, they are not because they apply to different aspects of the simulation. Designers must guard against “tunnel vision” where, on one hand, they spend all of their time working on details of the simulator that they understand well, while on the other, they ignore details that they don’t. Otherwise, simulation complexity can begin to approach implementation complexity, but its accuracy will not! The positive lesson is to consider exactly what one wants to learn from a simulation, build the simplest simulator that can examine the issues, and to validate the simulation results against the real world.

My first major simulation study [Dahlin et al., 1994] (which does not appear in this dissertation) is a good example of the dangers of careless simulator design. In that study, I built an event-driven simulator that not only modeled CPUs, disks, and networks, but also modeled queueing for those resources. This simulator was far too complex — the event-driven approach introduced logical concurrency to the simulator, forcing me to worry about locking data structures and avoiding race conditions. Although this complexity made it appear that the simulator captured a lot of details, the details were probably meaningless because my choices of workload and my simplifying assumptions about the network, operating system, and disk all were more significant factors in the final results than the queuing effects.

Evaluating a prototype, however, is not simple either. The performance results that appear in Chapter 7 required myself and the rest of the xFS group to engage in literally hundreds of hours of tedious “baby-sitting” the machines while the simulations ran. At least part of this problem came from engineering errors on our part: we neglected practical pieces of the implementation that would have made our lives easier because they seemed to be boring compared to the “real research” parts of the design where we spent out time. For instance, restarting a client in a running xFS system required rebooting the machine on which the client was running and then killing and restarting all of the storage servers and managers in the system.

Beyond problems with engineering, however, evaluating a serverless network file system is inherently complex. Performance is affected by at least the following factors:

- the number of clients
- the number of managers
- the number of storage servers

• the number of fragments in a stripe

• the fragment size

• the size of files being accessed

• the ratio of reads to writes

• the degree of sharing

• the state of the caches

• the raw network bandwidth

• the raw disk bandwidth

• the prefetching policy

• the cache replacement policy

• ...

Because so many different factors affect performance, some design decisions and measurement priorities must be based on engineering judgement; there are too many factors to implement them all, let alone systematically investigate all of their impacts on performance.

8.2. Future Directions

The work described in this dissertation has addressed many of the basic questions about building a serverless network file system, but it leaves some questions unanswered and raises several new issues. This section first describes future work that will help evaluate this approach in more detail. It then discusses broader research issues raised by the project.

8.2.1. Serverless File System Issues

The simulation studies presented in this thesis and the measurements of the xFS prototype provide a basis for evaluating the serverless approach to building file systems. Future work is called for to flesh out the prototype, undertake additional simulation studies to address unanswered questions, and to extend the xFS design to handle other types of workloads.

The most immediate issues involve implementing significant pieces of the design described in this dissertation but not yet included in the xFS prototype. Three key efforts are to improve the performance of the prototype, implement dynamic reconfiguration, and implement parallel cleaning.

In addition, several other useful file system services should be added to xFS both to provide important services and to exploit the state of the art in file system design. For instance, recent advances in prefetching would allow the system to exploit its scalable bandwidth to reduce latency [Griffioen and Appleton, 1994, Cao et al., 1995, Patterson et al., 1995]. Improved cache replacement policies [Cao et al., 1994, Karedla et al., 1994] and cache implementation techniques [Braunstein et al., 1989, Stolarchuk, 1993] can improve performance, as can improved disk request scheduling [Teorey and Pinkerton, 1972, Seltzer et al., 1990, Worthington et al., 1994]. On-line compression can reduce storage costs [Burrows et al., 1992] and increase performance [Douglis, 1993]. The system could also use its cleaner to reorganize data stored on its disks to improve read performance [McKusick et al., 1984, Rosenblum and Ousterhout, 1992, Smith and Seltzer, 1994, Seltzer et al., 1995].

Another service needed to make xFS comparable to commercial file systems is backup. While serverless systems do not directly introduce new problems for backup subsystems, they rely on some form of scalable backup to prevent backup from limiting scalability. The serverless architecture does, however, provide facilities that may help construct scalable backup systems. First, because serverless systems already store redundant data, backup systems may worry less about disk head crashes and more about restoring past versions of individual files; this mode of operation would place a premium on random access to the archive as opposed to the ability to restore an entire volume quickly (although the system must still retain the ability to recover the entire file system to enable recoveries from catastrophes affecting the entire file system.) Second, the manager module abstracts location service and already tracks data through different levels of the storage hierarchy (cache and disk). The manager might be extended to track data stored in the backup archive as well. Finally, the LFS cleaner might be adapted to migrate and backup data to the archive as it cleans segments, or it might be modified to avoid cleaning recently deleted or overwritten blocks to provide a user-accessible "undelete" or "time-travel" function such as that found in the Network Appliance *snapshots* [Hitz, 1993].

In addition to these implementation issues, a number of detailed simulation and measurement studies will help improve the understanding of different aspects of the serverless design. As Section 3.1.7 on page 34 discussed, there are several interesting variations of cooperative caching algorithms that should be studied. Also, several aspects of the distributed cleaner design should be examined; for instance, as Section 4.3 on page 82 indicated, a specific algorithm must be developed to balance locality and load when activating cleaners. Furthermore, in Section 5.3 on page 104, I hypothesized that parallel workloads might benefit more from distributed management than the sequential workloads I studied; to examine this hypothesis, the management strategies should be studied under a wider range of workloads. Finally, Section 6.2 on page 125 suggested a "core-fringe" protocol for dealing with mixed-security environments; the performance of this approach, however, needs to be examined.

Once the basic system is operational, one track of research will be evaluating a wide range of workloads and making any necessary changes to support them. For instance, large data base systems might benefit from xFS's scalable I/O, but for best performance, these systems need to have more precise control over resource usage than is provided by the Unix file system interface on which xFS is built. One possible addition to the interface is to allow a single client to write to multiple logs, each of which contains a specific subset of the database's files and each of which the database stores on a specific set of storage servers. A second enhancement would be to add additional storage server architectures; for instance as described in Chapter 4, a RAID level 1 mirrored storage system could be combined with write-ahead logging to support efficient update-in-place often needed by the random update patterns of large databases.

Parallel programs might also benefit from being able to specify particular assignments of data to disks [Corbett et al., 1993] or from new storage server architectures. For instance, some parallel programs might prefer to log data without parity to increase their performance even at the risk of lost data. Systems that support such unreliable writes, however, must ensure that no other data can be lost through file system inconsistencies.

Multimedia workloads should also benefit from xFS's scalable bandwidth, but the system may need to be modified to support real-time scheduling for best effect.

Finally, other issues arise when adapting a file system for wide area network (WAN) workloads. Although the serverless protocol assumes that machines are tightly coupled, a hybrid system could be used in which the serverless protocol is used within clusters of machines connected by a LAN, while clusters share data with one another using a different, WAN protocol [Sandhu and Zhou, 1992]. An earlier version of the xFS protocol [Wang and Anderson, 1993, Dahlin et al., 1994] would be an effective WAN protocol.

8.2.2. Other Research Issues

The serverless network file systems design attempts to distribute a complex system across multiple machines in a way that provides scalable performance and reliability. Many of the issues raised in this research have broader applications — to other complex, distributed, scalable, high-performance, and highly-available systems.

The experience of evaluating the performance of the xFS prototype has highlighted the difficulties of measuring and tuning complex computer systems. Developing methodologies to evaluate and tune complex systems is an interesting research question of immense practical concern. Because even commercial systems are becoming quite complex, such methodologies would have applications beyond leading-edge research systems like xFS. The goal is make these systems self-diagnosing or self-tuning or both. Ideally, for instance, a system would configure its parameters in response to changing workloads or hardware configurations to achieve as good performance as possible, even when it is run by someone who does not fully understand its design.

A related issue is balancing competing resource demands in a Network of Workstations (NOW) environment, where different distributed or serverless systems make demands for resources. In the Berkeley NOW project, for instance, many different subsystems compete to use memory: the Inktomi world wide web search engine application uses memory as a cache for data it stores on disk; Network RAM allows workstations to page virtual memory to remote client memory rather than to their local disks; xFS uses cooperative caching to cache files in client memories; and, of course, users run programs that require RAM memory. To complicate matters further, the operating system dynamically balances the amount of memory used for file cache or virtual memory on each machine. Each of these consumers of memory can deliver marginally better performance to its user if it is allowed to use more physical memory, but the amount of memory that the system has to distribute among these applications is finite. Research is needed to determine how best to allot limited resources to such diverse demands not only for memory capacity, but also for CPU cycles, disk accesses, and network accesses.

File systems researchers would benefit from a standard file system programming interface that covers a broader range of issues than the vnode layer. While vnodes standardize the interface that allows the operating system to access file system services, it leaves out other interfaces important to file system implementation. These omissions cause two problems. First, they make it difficult to integrate a new file system into multiple operating system platforms. For instance, the interface to the buffer cache differs across different Unix platforms, forcing a file system developer to develop and maintain multiple sets of code. Second, the lack of well-defined interfaces to different functions forces file system designers to “reinvent the wheel” for many subsystems of each system. For instance, as noted above, the xFS prototype has not yet implemented sophisticated prefetching despite the wealth of recent research in the area [Kotz and Ellis, 1991, Cao et al., 1995, Patterson et al., 1995]. Standard interfaces would make it easier for researchers to benefit from one another’s innovations. xFS’s functional division of the file system into distinct modules for storage, caching, and management may provide a starting point for this investigation, as may recent work in *stackable* file system interfaces [Khalidi and Nelson, 1993, Heidemann and Popek, 1994].

The serverless file system design addresses the issue of recovery in a distributed system, demonstrating one approach to the problem, but a systematic investigation of the range of approaches would be useful. The serverless design represents one end of the spectrum: it achieves availability by logging batches of state changes to highly-available storage. Conversely, systems like Isis [Birman and Cooper, 1990], Coda [Kistler and Satyanarayanan, 1992], Echo [Birrell et al., 1993], and Horus [Renesse et al., 1994], replicate servers and keep the replicas synchronized with one another for each modification to the system’s state. These approaches make different trade-offs between update-time, cost, and recovery-time. Basing availability on logged, redundant storage allows fast updates because it batches modifications, and it is inexpensive because it stores the backup copy of system state on disk rather than maintaining multiple running copies of the system. On the other hand, after a machine fails, this approach must recover that machine’s state from disk, so fail-over may take a significant amount of time. The other approach, server replication, has faster fail-over because it maintains an on-line replica, but updates are slow because each update must be committed at both the primary machine and the backup machine, and the approach is expensive because it maintains two running copies of the system. Research is needed to systematically investigate the impact of logging updates to disk, batching updates, and maintaining multiple running versions of a system. Such research would help to understand when each approach is appropriate, or when a hybrid approach should be used. One example of such a hybrid is Bressoud’s *hypervisor* [Bressoud and Schneider, 1995], which transmits a log of updates directly from a primary server to a secondary, gaining some of the advantages of logging while maintaining fast recovery.

Finally, it would be valuable to design a system that is “always” available — one whose mean time to failure is measured in years or decades rather than weeks or months. Designing such a system would require a top-to-bottom evaluation of design decisions and would also require the designer to address such issues as geographic distribution of data (to protect against natural or man-made disasters) and hot-swap upgrades of hardware or software.

8.3. Summary

The original goals of distributed systems were better performance and availability, but distributed systems, in general, and distributed file systems, in particular, have often been built around a central server that implements most of the system's functionality. The lack of location independence in central server architectures results in crucial performance or availability bottlenecks. The serverless file system architecture exploits fast, scalable networks and aggressive location independence to provide scalable performance and availability and to deliver the full capabilities of a collection of "killer micros" to file systems users. The challenge in the future is to generalize the principles explored in this file system design to other applications to enable a new generation of high performance, highly-available, and scalable distributed systems.

Bibliography

Adler, M., Chakrabarti, S., Mitzenmacher, M., and Rasmussen, L. (1995). Parallel Randomized Load Balancing. In *Proceedings of the Twenty-seventh ACM Symposium on Theory of Computing*.

Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M. (1988). An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the Fifteenth International Symposium on Computer Architecture*, pages 280–289.

Anderson, T., Culler, D., Patterson, D., and the NOW team (1995). A Case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64.

Archibald, J. and Baer, J. (1984). An Economical Solution to the Cache Coherence Problem. In *Proceedings of the Eleventh International Symposium on Computer Architecture*, pages 355–362.

Archibald, J. and Baer, J. (1986). Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298.

Arpaci, R., Dusseau, A., Vahdat, A., Liu, L., Anderson, T., and Patterson, D. (1995). The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 267–278.

ATM Forum (1993). *The ATM Forum User-Network Interface Specification, version 3.0*. Prentice Hall Intl., New Jersey.

Baker, M. (1994). *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley.

Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. (1992). Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 10–22.

Baker, M., Hartman, J., Kupfer, M., Shirriff, K., and Ousterhout, J. (1991). Measurements of a Distributed

File System. In *Proceedings of the ACM Thirteenth Symposium on Operating Systems Principles*, pages 198–212.

Basu, A., Buch, V., Vogels, W., and von Eicken, T. (1995). U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 40–53.

Bellovin, S. (1992). There be Dragons. In *USENIX Unix Security III*, pages 1–16.

Ben-Or, M. (1990). *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, chapter Randomized Agreement Protocols, pages 72–83. Springer-Verlag.

Birman, K. and Cooper, R. (1990). The ISIS Project: Real Experience with a Fault Tolerant Programming System. In *European Workshop on Fault-Tolerance in Operating Systems*, pages 103–107.

Birrell, A., Hisgen, A., Jerian, C., Mann, T., and Swart, G. (1993). The Echo Distributed File System. Technical Report 111, Digital Equipment Corp. Systems Research Center.

Blackwell, T., Harris, J., and Seltzer, M. (1995). Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proceedings of the Winter 1995 USENIX Conference*.

Blaum, M., Brady, J., Bruck, J., and Menon, J. (1994). EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 245–254.

Blaze, M. (1993). *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University.

Blumrich, M., Li, K., Alpert, R., Dubnicki, C., Felten, E., and Sandberg, J. (1994). Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 142–153.

Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J., and Su, W. (1995). Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, pages 29–36.

Bondurant, D. (1992). Enhanced Dynamic RAM. *IEEE Spectrum*, page 49.

Braunstein, A., Riley, M., and Wilkes, J. (1989). Improving the Efficiency of UNIX File Buffer Caches. In *Proceedings of the ACM Twelfth Symposium on Operating Systems Principles*, pages 71–82.

Bressoud, T. and Schneider, F. (1995). Hypervisor-based Fault Tolerance. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 1–11.

Brewer, E., Gauthier, P., Goldberg, I., and Wagner, D. (1995). Basic Flaws in Internet Security and Commerce. http://www.cs.berkeley.edu/ gauthier/ endpoint-security.html.

Burrows, M. (1988). *Efficient Data Sharing*. PhD thesis, Cambridge University, Cambridge, England.

Burrows, M., Jerian, C., Lampson, B., and Mann, T. (1992). On-line data compression in a log-structured file system. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 2–9.

Bursky, D. (1992). Memory-CPU Interface Speeds Up Data Transfers. *Electronic Design*, pages 137–142.

Cabrera, L. and Long, D. (1991). Swift: A Storage Architecture for Large Objects. In *Proceedings of the Eleventh Symposium on Mass Storage Systems*, pages 123–128.

Cao, P., Felten, E., Karlin, A., and Li, K. (1995). Implementation and Performance of Integrated Application-Controlled Caching, Prefetching, and Disk Scheduling. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–197.

Cao, P., Felten, E., and Li, K. (1994). Application Controlled File Caching Policies. In *Proceedings of the Summer 1994 USENIX Conference*, pages 171–82.

Censier, L. and Feautrier, P. (1978). A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118.

Chaiken, D., Kubiatowicz, J., and Agarwal, A. (1991). LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 224–234.

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

Comer, D. and Griffioen, J. (1992). Efficient Order-Dependent Communication in a Distributed Virtual Memory Environment. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 249–262.

Corbett, P., Baylor, S., and Feitelson, D. (1993). Overview of the Vesta Parallel File System. *Computer Architecture News*, 21(5):7–14.

Coyne, R. A. and Hulen, H. (1993). An Introduction to the Mass Storage System Reference Model, Version 5. In *Proceedings of the Thirteenth Symposium on Mass Storage Systems*, pages 47–53.

Cristian, F. (1991). Reaching Agreement on Processor Group Membership in Synchronous Distributed

Systems. *Distributed Computing*, 4:175–187.

Cristian, F., Dolev, D., Strong, R., and Aghili, H. (1990). *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, chapter Atomic Broadcast in a Real-Time Environment, pages 51–71. Springer-Verlag.

Cypher, R., Ho, A., Konstantinidou, S., and Messina, P. (1993). Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 2–13.

Dahlin, M., Mather, C., Wang, R., Anderson, T., and Patterson, D. (1994). A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150–160.

Denning, D. and Denning, P. (1979). Data Security. *Computing Surveys*, 11(3):227–249.

Dibble, P. and Scott, M. (1989). Beyond Striping: The Bridge Multiprocessor File System. *Computer Architechture News*, 17(5):32–39.

Douglis, F. (1993). The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the Winter 1993 USENIX Conference*, pages 519–529.

Douglis, F. and Ousterhout, J. (1991). Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(7):757–785.

Drapeau, A. (1993). *Striped Tertiary Storage Systems: Performance and reliability*. PhD thesis, University of California at Berkeley.

Eager, D., Lazowska, E., and Zahorjan, J. (1986). Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675.

Feeley, M., Morgan, W., Pighin, F., Karlin, A., Levy, H., and Thekkath, C. (1995). Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 201–212.

Felten, E. and Zahorjan, J. (1991). Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Dept. of Computer Science, University of Washington.

Franklin, M., Carey, M., and Livny, M. (1992). Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of the International Conference on Very Large Data Bases*, pages 596–609.

Gibson, G. A. (1992). *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. ACM Distinguished

Dissertations. MIT Press, Cambridge, Massachusettes.

Gray, J. (1995). Personal Communication.

Griffioen, J. and Appleton, R. (1994). Reducing File System Latency Using a Predictive Approach. In *Proceedings of the Summer 1994 USENIX Conference*, pages 197–207.

Gwennap, L. (1995). Processor Performance Climbs Steadily. *Microprocessor Report*.

Hagersten, E., Landin, A., and Haridi, S. (1992). DDM–A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):45–54.

Hagmann, R. (1987). Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the ACM Eleventh Symposium on Operating Systems Principles*, pages 155–162.

Hart, C. (1992). Dynamic RAM as Secondary Cache. *IEEE Spectrum*, page 46.

Hartman, J. and Ousterhout, J. (1995). The Zebra Striped Network File System. *ACM Transactions on Computer Systems*.

Haynes, R. and Kelly, S. (1992). Software Security for a Network Storage Service. In *USENIX Unix Security III*, pages 253–265.

Heidemann, J. and Popek, G. (1994). File-system Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89.

Hennessy, J. and Patterson, D. (1996). *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2nd edition.

Hitz, D. (1993). An NFS Server Appliance. Technical Report TR01, Network Appliance Corporation.

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

Hsiao, H. and DeWitt, D. (1989). Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. Technical Report CS TR 854, University of Wisconsin, Madison.

Iftode, L., Li, K., and Petersen, K. (1993). Memory Servers for Multicomputers. In *Proceedings of COMPCON93*, pages 538–547.

Jones, F. (1992). A New Era of Fast Dynamic RAMs. *IEEE Spectrum*, pages 43–48.

Karedla, R., Love, J., and Wherry, B. (1994). Caching Strategies to Improve Disk System Performance. *IEEE Computer*, pages 38–46.

Kazar, M. (1989). Ubik: Replicated Servers Made Easy. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 60–67.

Kazar, M., Leverett, B., Anderson, O., Apostolides, V., Bottos, B., Chutani, S., Everhart, C., Mason, W., Tu, S., and Zayas, E. (1990). Decorum File System Architectural Overview. In *Proceedings of the Summer 1990 USENIX Conference*, pages 151–163.

Keeton, K., Anderson, T., and Patterson, D. (1995). LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Proceedings of the 1995 Hot Interconnects III Conference*.

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

Kistler, J. and Satyanarayanan, M. (1992). Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25.

Kotz, D. and Ellis, C. (1991). Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 182–189.

Kubiatowicz, J. and Agarwal, A. (1993). Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the Seventh International Conference on Supercomputing*.

Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J. (1994). The Stanford FLASH Multiprocessor. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 302–313.

Le, M., Burghardt, F., Seshan, S., and Rabaey, J. (1995). InfoNet: The Networking Infrastructure of InfoPad. In *Proceedings of COMPCON 95*, pages 163–168.

Lee, E. (1995). Highly-Available, Scalable Network Storage. In *Proceedings of COMPCON 95*, pages 397–402.

Leff, A., Wolf, J., and Yu, P. (1993a). Replication Algorithms in a Remote Caching Architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204.

Leff, A., Yu, P., and Wolf, J. (1991). Policies for Efficient Memory Utilization in a Remote Caching Architecture. In *Proceedings of the First International Conference on Parallel and Distributed*

Information Systems, pages 198–207.

Leff, A., Yu, P., and Wolf, J. (1993b). Performance Issues in Object Replication for a Remote Caching Architecture. *Computer Systems Science and Engineering*, 8(1):40–51.

Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. (1990). The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 148–159.

Lieberman, H. and Hewitt, C. (1983). A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429.

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. (1991). Replication in the Harp File System. In *Proceedings of the ACM Thirteenth Symposium on Operating Systems Principles*, pages 226–238.

LoVerso, S., Isman, M., Nanopoulos, A., Nesheim, W., Milne, E., and Wheeler, R. (1993). sfs: A Parallel File System for the CM-5. In *Proceedings of the Summer 1993 USENIX Conference*, pages 291–305.

Martin, R. (1994). HPAM: An Active Message Layer for a Network of HP Workstations. In *Proceedings of the 1994 Hot Interconnects II Conference*.

Martin, R. (1995). Personal Communication.

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

Mummert, L., Ebling, M., and Satyanarayanan, M. (1995). Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 143–155.

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

Mutka, M. and Livny, M. (1991). The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84.

Myllymaki, J. (1994). Overview of Current RAID Technology. http:// www.cs.wisc.edu/ jussi/ raidtech.html.

Nelson, M., Welch, B., and Ousterhout, J. (1988). Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1).

Nichols, D. (1987). Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the ACM Eleventh Symposium on Operating Systems Principles*, pages 5–12.

Ousterhout, J. (1990). Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the Summer 1990 USENIX Conference*.

Patterson, D., Gibson, G., and Katz, R. (1988). A Case for Redundant Arrays of Inexpensive Disks (RAID). In *International Conference on Management of Data*, pages 109–116.

Patterson, R., Gibson, G., Ginting, E., Stodolsky, D., and Zelenka, J. (1995). Informed Prefetching and Caching. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 79–95.

Pierce, P. (1989). A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of the Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 155–160.

Popek, G., Guy, R., Page, T., and Heidemann, J. (1990). Replication in the Ficus Distributed File System. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 5–10.

Prince, B., Norwood, R., Hartigan, J., and Vogley, W. (1992). Synchronous Dynamic RAM. *IEEE Spectrum*, pages 44–46.

Rashid, R. (1994). Microsoft's Tiger Media Server. In *The First Networks of Workstations Workshop Record*.

Renesse, R. V., Hickey, T., and Birman, K. (1994). Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR94-1442, Cornell University Computer Science Department.

Rivest, R. (1992a). The MD4 Message-Digest Algorithm. Request for Comments 1320, Network Working Group, ISI.

Rivest, R. (1992b). The MD5 Message-Digest Algorithm. Request for Comments 1321, Network Working Group, ISI.

Rosenblum, M., Bugnion, E., Herrod, S., Witchel, E., and Gupta, A. (1995). The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 285–298.

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

Rosti, E., Smirni, E., Wagner, T., Apon, A., and Dowdy, L. (1993). The KSR1: Experimentation and

Modeling of Poststore. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 74–85.

Ruemmler, C. and Wilkes, J. (1993). UNIX Disk Access Patterns. In *Proceedings of the Winter 1993 USENIX Conference*, pages 405–420.

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130.

Sandhu, H. and Zhou, S. (1992). Cluster-Based File Replication in Large-Scale Distributed Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 91–102.

Satyanarayanan, M. (1989). Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, 7(3):247–280.

Schilit, B. and Duchamp, D. (1991). Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, Dept. of Computer Science, Columbia University.

Schmuck, F. and Wyllie, J. (1991). Experience with Transactions in Quicksilver. In *Proceedings of the ACM Thirteenth Symposium on Operating Systems Principles*, pages 239–253.

Schroeder, M., Birrell, A., Burrows, M., Murray, H., Needham, R., Rodeheffer, T., Satterthwaite, E., and Thacker, C. (1991). Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communication*, 9(8):1318–1335.

Schroeder, M. and Burrows, M. (1990). Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17.

Seagate (1994). *ST-11200N SCSI-2 Fast (Barracuda 4) Specification*. Seagate Technology, Inc.

Seltzer, M., Bostic, K., McKusick, M., and Staelin, C. (1993). An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the Winter 1993 USENIX Conference*, pages 307–326.

Seltzer, M., Chen, P., and Ousterhout, J. (1990). Disk Scheduling Revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324.

Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. (1995). File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the Winter 1995 USENIX Conference*, pages 249–264.

Smith, A. (1977). Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE*

Transactions on Software Engineering, SE-3(1):94–101.

Smith, A. (1981). Long Term File Migration: Development and Evaluation of Algorithms. *Computer Architecture and Systems*, 24(8):521–532.

Smith, K. and Seltzer, M. (1994). File Layout and File System Performance. Technical Report TR-35-94, Harvard University.

Srinivasan, V. and Mogul, J. (1989). Spritely NFS: Experiments with Cache Consistency Protocols. In *Proceedings of the ACM Twelfth Symposium on Operating Systems Principles*, pages 45–57.

Steiner, J., Neuman, C., and Schiller, J. (1988). Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of the Winter 1988 USENIX Conference*, pages 191–202.

Stolarchuk, M. (1993). Faster AFS. In *Proceedings of the Winter 1993 USENIX Conference*, pages 67–75.

Tang, C. (1976). Cache Design in the Tightly Coupled Multiprocessor System. In *Proceedings of the AFIPS National Computer Conference*.

Teorey, T. and Pinkerton, T. (1972). A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM*, pages 177–84.

Theimer, M. and Lantz, K. (1989). Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–57.

Thekkath, C. and Levy, H. (1993). Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203.

Thompson, J. (1987). *Efficient Analysis of Caching Systems*. PhD thesis, University of California at Berkeley.

von Eicken, T., Basu, A., and Buch, V. (1995). Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, pages 46–53.

von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. E. (1992). Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 256–266.

Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. (1983). The LOCUS distributed operating system. In *Proceedings of the ACM Ninth Symposium on Operating Systems Principles*, pages 49–69.

Wang, R. and Anderson, T. (1993). xFS: A Wide Area Mass Storage File System. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 71–78.

Wheeler, D. (1993). A Bulk Data Encryption Algorithm. In *Proceedings of the Fast Software Encryption Cambridge Security Workshop*, pages 127–133.

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. (1995). The HP AutoRAID Hierarchical Storage System. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles*, pages 96–108.

Wittle, M. and Keith, B. (1993). LADDIS: The Next Generation in NFS File Server Benchmarking. In *Proceedings of the Summer 1993 USENIX Conference*, pages 111–28.

Worthington, B., Ganger, G., and Patt, Y. (1994). Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 241–251.

Yen, W., Yen, D., and Fu, K. (1985). Data Coherence Problem in a Multicache System. *IEEE Transactions on Conputers*, 34(1):56–65.