A Quantitative Analysis of Cache Policies for Scalable Network File Systems

Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang,
Thomas E. Anderson, and David A. Patterson

Computer Science Division, University of California at Berkeley
{dahlin, cjmather, rywang, tea, pattrsn}@cs.berkeley.edu

Abstract

Current network file system protocols rely heavily on a central server to coordinate file activity among client workstations. This central server can become a bottleneck that limits scalability for environments with large numbers of clients. In central server systems such as NFS and AFS, all client writes, cache misses, and coherence messages are handled by the server. To keep up with this workload, expensive server machines are needed, configured with high-performance CPUs, memory systems, and I/O channels. Since the server stores all data, it must be physically capable of connecting to many disks. This reliance on a central server also makes current systems inappropriate for wide area network use where the network bandwidth to the server may be limited.

In this paper, we investigate the quantitative performance effect of moving as many of the server responsibilities as possible to client workstations to reduce the need for high-performance server machines. We have devised a cache protocol in which all data reside on clients and all data transfers proceed directly from client to client. The server is used only to coordinate these data transfers. This protocol is being incorporated as part of our experimental file system, xFS. We present results from a trace-driven simulation study of the protocol using traces from a 237 client NFS installation. We find that the xFS protocol reduces server load by more than a factor of six compared to AFS without significantly affecting response time or file availability.

1 Introduction

Current network file systems rely on powerful central servers that make it difficult to build economical large-scale file systems. Ideally, a network file system should scale to hundreds or thousands of client machines using nothing more than commodity workstations, even for the server. In reality, the widely used SUN Network File System, NFS [Sand85], has spawned a new industry dedicated to building the high-performance multiprocessor systems needed to scale NFS to more than a few dozen clients. The Andrew File System, AFS [Howa88], was designed to reduce server load relative to NFS in the interest of scalability, but its ultimate scalability is limited because AFS still relies on a central server to receive a copy of all modified data and to supply data for all client cache miss requests. Also, NFS and AFS must generally use specialized servers rather than commodity desktop workstations as server machines because the server must support enough disks to hold a copy of the entire file system, and desktop workstations are generally limited to a single SCSI string. Commodity workstations are a more cost effective way to buy computing power and I/O bandwidth because server machines must be designed with greater I/O expandability and because development costs of the more complicated servers must be amortized over a smaller sales volume. For instance in the SUN product line, a server costs three times as much as a similarly configured workstation.

Trends in file system use promise to place even heavier demands on the central servers of file systems. Baker et al. [Bake91] report that the size of large files grew by an order of magnitude between 1985 and 1992. If this trend continues, the cost of transferring all data through the central server may become prohibitive. File systems are also being asked to manage data over wide area networks (WANs) where bandwidth restrictions limit the amount of data that can be supplied from a central source.

At the same time, technology trends are giving clients tremendous amounts of disk space, main memory, and processing power and are also providing high-speed low-latency networks to tie these resources together. Inexpensive disks make it feasible for clients to store large amounts of data locally. A 1.3 GB SCSI disk currently costs less than $1000, and most workstations sold today are configured with significant amounts of local disk. Similarly, the aggregate memories and processing resources of client machines dwarf the capacity of a single server machine. Additionally, high speed local area networks (LANs) allow clients to access data from peers across a local area network almost as quickly as they can access local data.

This paper investigates the quantitative benefits of utilizing cache techniques oriented towards extreme scalability

This work is supported in part by the Advanced Research Projects Agency (N00600-93-C-2481), the National Science Foundation (CDA 8722788), California MICRO, Digital Equipment Corporation, the AT&T Foundation, Xerox Corporation, and Siemens Corporation. Dahlin was also supported under a National Science Foundation Graduate Research Fellowship. Anderson was also supported by a National Science Foundation Young Investigator Award.

to reduce the load on the central server. These techniques provide better and more cost-effective file service than a specialized server machine by pushing responsibilities onto the clients in the system to exploit the aggregate client disk, processing, and memory capacities. The protocol has four pieces inspired by efforts to achieve scalable cache coherence in massively parallel processors [Arch86, Leno90]. The protocol uses a no write through policy, utilizes client-to-client data transfers, implements write ownership, and takes advantage of cluster servers.

In this paper we compare the effects of this protocol to a baseline AFS system. We base this comparison on event driven simulation parameterized to model the service demands of file system requests on a DECstation 5000. We compared the performance of the systems under a workload taken from a large NFS system at Berkeley in which an Auspex file server provides service for 237 client workstations[1].

Our principal result is that for this workload the experimental protocol reduced server load by more than a factor of six. In addition, we show that each of the four parts of the protocol has a significant impact on performance, that the protocol not only reduces average server load but also significantly reduces peak demand at the server, that the aggregate client memories are more effective at reducing disk I/O than the server memory, and that cluster servers isolate almost all communication to within clusters, as is desirable when clusters are connected by a WAN.

This study also addresses a number of issues that arise when clients become responsible for more file system services. We examine the problems of scalable backup, data availability in the presence of client failures, and security when clients supply data to each other.

We are currently implementing the protocol described here as part of a file system called xFS. To facilitate comparison with AFS, this paper assumes the AFS policy of synchronizing file consistency when a file is closed for writing and assumes whole-file caching. However our implementation of xFS does not have these restrictions. The actual xFS implementation also stripes data in a RAID distributed across client disks to improve I/O bandwidth and availability. We do not consider these other issues further.

Section 2 of this paper outlines the file system caching algorithms used by NFS and AFS and motivates the alternative strategies we consider in this paper. Section 3 describes our workload, and in Section 4 we discuss the key aspects of our simulation. Section 5 details the results of our study, with emphasis on the impact of the protocol on server scalability, network load, and client load. Section 6 examines the potentially thorny issues of backup, availability, and security that arise when clients are given responsibilities that were formerly the server's. We survey related network file system studies in Section 7. Finally in Section 8 we summarize our conclusions.

[1] The Auspex is built with special hardware to allow it to support this large number of clients [Hitz90].

2 File System Cache Protocols

An important factor in a file system's scalability is its caching policy. File systems use caches to improve response time and to reduce server load. Clients can access data found in their memory caches more quickly than they can access remote data on a server. File caches reduce server load by satisfying some requests without server interaction. The use of caches, however, introduces the problem of *cache consistency*: different caches may hold copies of the same file, and if the file is changed by one client, the changes must be seen when the file is read by a different client. How the cached copies are kept consistent can have a large effect on server scalability. This section describes the cache protocols found in the industry-standard NFS, the emerging AFS standard, and our more scalable xFS.

2.1 Existing Protocols

NFS, the current industry-standard distributed file system protocol, was designed to provide good response time for moderate numbers of clients rather than to provide scalability. NFS caches file system data in main memory on each client workstation. NFS does not attempt to use the client's local disk space as a cache, nor does it attempt to keep file data strictly coherent. Instead, periodic invalidations of file attribute information ensure that new data eventually (within several seconds) replace any out-of-date cached copies. Once a file's attributes are invalidated, the next time the file is referenced the client will verify that its cached copy is current. If it is not, the client will fetch the new data from the server. Clients write through all modified file data to the server to ensure that fetches from other clients will receive the new data. NFS maintains separate caches for data, attributes, and names, and the protocol caches data on a per-block basis.

NFS's scalability is limited by its use of relatively small in-memory file caches rather than the larger caches possible if local disks were used. NFS's policy of periodically invalidating attributes guarantees a stream of client requests to the server as attributes expire, even for files that are not being modified. NFS's write through policy sends all changes to the server disk, even if no other clients are using the data.

AFS improves upon NFS's scalability by using a large local on-disk cache at each client and by using callbacks for cache consistency. AFS uses a two-level cache on each client. An in-memory file cache similar to NFS's file cache provides good response time for most accesses, but misses to the in-memory file cache go to a client on-disk file cache and only go to the server if not satisfied there. Rather than requiring periodic verification of a file's consistency as NFS does, AFS reduces server load further by using *callbacks*: the server maintains a list of all cached copies of each data file and notifies clients when another client modifies the file.

The client fetches the new data from the server the next time it opens the file. AFS clients send all modified data to the server when a file is closed, guaranteeing that the server has the most current version of the data and allowing the server to know when to invalidate the other cached copies. Clients also cache directory information in write through directory caches. All modifications to directories are sent immediately to the server, which maintains callbacks to keep cached copies of directory information consistent.

Despite these improvements, AFS's scalability is still limited. All communication and data transfer takes place between the clients and the server; no direct client-to-client communication is allowed. In particular, the server supplies data each time a client has a cache miss and receives data each time a client closes a file it has written. The central server must have enough disk space to store all of the file system data; this is despite the fact that the aggregate size of the client disks is typically much larger than that of the server disks. The server is also responsible for fielding all directory modification operations and for generating callback messages on every cache coherence operation.

2.2 xFS Protocol

In this paper we consider the effect of four separate optimizations to the AFS protocol. Together, these push most server responsibilities onto the client machines. Collectively we refer to these as the xFS protocol. Each of these optimizations has been proposed as a way to improve the scalability of multiprocessor hardware caches, and some have also been suggested for file systems. In Section 5.2 we evaluate their performance impact individually and find that all are important to get good performance.

The first two optimizations, write through and client-to-client data transfers, eliminate file transfers through the server, making the server responsible only for coordinating the data that flow from client to client. These two aspects of the protocol also eliminate the need to buy a specialized server machine configured with a large amount of disk space.

1. **No write through**. Clients no longer write modified data to the server on close. Instead they inform the server of the update, and the server invalidates cached copies at other clients using callbacks. Modified files remain on each client's local disk.

The elimination of write through is motivated by a number of studies showing that when a client writes a file, it is often deleted or quickly rewritten by the same client [Thom87, Bake91, Blaz91, Kist92]. We confirmed this pattern for the Berkeley NFS traces. After discarding the statistics for writes made during the last day of the simulation, we found that of the bytes that would be sent to the server in AFS, 85% were overwritten or deleted without being read by another client, another 5% were never read by another client, and only 10% were read by another client. With no write through, bytes that are overwritten can be discarded at the client without ever being transferred to the server. Note that there is no need to write data to the server to ensure the data's durability. Delayed writes of about 30 seconds have been used in other network file systems to reduce writes to the server without putting too much data at risk of loss in a client crash [Nels88], but xFS's use of client disks allows a complete no write through policy where files are never written to the server.

2. **Client-to-client data transfers**. When a client has a cache miss, it sends a request for the data to the server, and the server forwards the request to a client that is currently caching the needed data. The second client then sends the desired data directly to the client that wants the data.

Client-to-client data transfers reduce server load by replacing a large server data transfer with a small forwarding packet. Direct client-to-client communication also permits the no write through policy to be implemented without the significant delays that would be incurred if all requested dirty data were first written to the central server and then supplied by the server. Client-to-client data transfers are an example of separating the control and data paths as suggested by the Mass Storage Reference Model [Coyn93].

The first two parts of the protocol allow all data to be stored on client disks, implementing what is referred to for multiprocessors as a cache only memory architecture (COMA) [Hage92, Rost93]. An important detail of this approach is that we must guarantee that the clients don't discard the last copy of any file. The clients do this by marking one copy of each file as permanent. A copy becomes marked when it is written, and marked copies may be passed between clients but not discarded. When a client's cache is full it sends any marked copies it would normally discard to a randomly selected client. The client notifies the server of this transfer. These marked data copies are otherwise managed in the same way as unmarked data copies.

The final two optimizations, write ownership and clustering, try to reduce the demands on the server of coordinating cached copies of files.

3. **Write ownership based cache consistency**. The first time a client closes a modified file the server is notified, triggering an invalidation of all copies cached on other clients. At that point the client has exclusive *write ownership* [Arch86] and may modify the file freely without notifying the server; there are no other copies of the data to be invalidated. A client will lose exclusive ownership of a file when another client opens the file for reading. Its copy will be invalidated if another client acquires exclusive ownership.

Write ownership is an optimization of the write invalidate consistency protocol on which AFS's callback mechanism is based. It allows us to eliminate messages to the server in the common case of repeated writes by the same client to the same file.

4. **Clustering**. Clusters are formed by selecting groups of workstations that closely cooperate or are near each other on the network topology, for instance, on the same LAN. Cluster servers keep track of the ownership and callback state for all of the clients in the cluster. The central server only tracks file location information to the cluster level, relying on the cluster server to forward requests to the specific clients caching data. The cluster servers isolate ownership changes and data transfers internal to the cluster from the central server. For instance, if ownership is transferred between two clients in the same cluster, the cluster server is notified of the change, but the central server need not be. On the other hand, if ownership is transferred between clients from different clusters, the central server must be involved so it can know that a new cluster server is responsible for tracking the ownership of the file.

Clustering is inspired by the DASH multiprocessor architecture [Leno90] which clusters processing nodes on busses as we cluster clients on LANs. Clustering improves scalability by off-loading some central server state to cluster servers and isolating the central server from changes in state only affecting clients in the cluster [Blaz91, Munt92, Sand92]. Clustering also allows the system to work in a wide area network context by organizing communication around the cluster LAN networks and using the WAN links only when necessary.

xFS clustering is distinct from name space splitting and read replication, two methods of utilizing multiple servers available to NFS and AFS. Name space splitting improves file system scalability by manually splitting the file system into logical pieces, each managed by a different server. However, it can be difficult to divide files among servers so as to balance load and avoid hot spots [Wolf89]. Name space splitting is, however, useful when the different parts of the file system are managed by different administrative domains. xFS can support this splitting by using multiple "central" servers, with each cluster server providing a combined file system view to its clients. File systems can also be replicated across multiple servers to improve scalability for reading files, at a cost of making file writes more expensive [Lisk91, Kist92]. We will show in Section 5 that xFS-style clustering reduces the cost of both reads and writes.

The combination of these four changes allows xFS to be dramatically more scalable than AFS. The central server is no longer involved in any data transfers and coordinates a much smaller amount of control activity: the central server must forward read miss requests between clusters when they cannot be satisfied within a cluster; the central server must send consistency messages between clusters when a modification in one cluster invalidates cached copies in another; finally, the central server is informed when files are created or deleted so that it always knows what files exist and where to forward requests for all files.

3 Trace Overview

To evaluate the performance impact of these changes we gathered traces of NFS file system activity from a large NFS installation served by an Auspex file server. The system includes 237 clients spread over four Ethernets, each of which connects directly to the central server. The trace spans seven days, and unless noted, the measurements that appear in this paper cover the last six days of the trace after using the first day's activity to warm the caches. During the full seven day trace 141,574 files were referenced.

We gathered this trace by monitoring network activity on each of the four Ethernets. On each subnet we placed a workstation that monitored all network traffic using rpcspy [Blaz93] which is built on the Ultrix Packetfilter interface [Mogu87]. Over the trace period, rpcspy reported that it dropped 4% of all network traffic calls due to buffer overflow.

We postprocessed the NFS trace to reflect the semantics of the AFS and xFS protocols. Since we gathered the traces at the network, we had access only to NFS network traffic, which introduced some biases of NFS into our raw trace. For instance NFS has no network-visible open or close calls, and many `getattr` (get file attribute) calls are really for validating cache consistency.

In the first step of the postprocessing we added opens and closes to the trace. We added file opens before the first access to an unopened file. Read and read/write opens signal AFS and xFS to bring the file being accessed into the local on-disk cache. We inserted file closes immediately after the last file access before a long (2 minute) period of inactivity for the file or before a block write to block zero of a file after a series of writes to other parts of the file. We use AFS's write close semantics: after the close, the newly written file should be supplied to any subsequent read open.

Block reads and writes in the trace are caused by NFS in-memory cache misses. In AFS and in the version of xFS simulated here, these reads and writes cause local disk traffic, but no network activity, since whole file caching is assumed and file consistency is handled when the file is opened or closed.

We included NFS directory reads and writes as AFS and xFS directory reads and writes. Directories were simulated with the semantics that each directory write is immediately visible to the entire system. AFS implements this by writing directory changes through to the server while xFS uses its file ownership and invalidation mechanisms.

Finally, we include most NFS `getattr` calls as simulator requests for file attributes. We excluded `getattr` calls immediately before an access to a block of the same file, assuming those calls to be NFS cache validation packets. The simulator also dynamically eliminates many `getattr` calls by filtering calls through an attribute cache. The attribute cache is kept consistent in the same way as the directory cache for each protocol. An attribute is invalidated when the file it references is written. Note that we are not simulating the "access time" attribute, which is updated for each file read, for either AFS or xFS.

The resulting trace is similar to other measured AFS workloads in macro characteristics. Our simulated AFS server supplied on average 5.0 MB to each of its clients per day for read opens; [Spas94] measured 5.3 MB per client per day for a large AFS installation. We measured a 5.7 MB per client per day write back load; 4.7 MB per client per day loads were measured by [Spas94].

This trace reflects the file system activity of a real system. Although this enhances our confidence that the trace is realistic, the capabilities of the traced system can limit the activity seen in the trace. The prime example of this limitation is on peak load. Our trace will underestimate the peak server load that might be imposed on a more scalable system for two reasons. First, the limited speed of the traced system will spread out requests, resulting in longer periods of activity but lower peaks. Second, users will tend to avoid operations that take a long time on the traced system, lowering both peak and overall load. Sharing is another example where the system's limitations may distort the workload. Since NFS has weak data sharing semantics, few users attempt to share data. If more files were shared, both AFS and xFS would see increased server loads, although AFS's increase would be larger since AFS's sharing requires data transfer through the server while sharing under xFS is accomplished with read forwarding and invalidation packets.

4 Simulator Methodology

We built a simulator to evaluate the performance of AFS and xFS for the traced workload. This simulator starts with a model of system behavior describing what actions are taken to implement the AFS and xFS protocols. We then parameterized the system to reflect the performance of real hardware. The subsections below describe the system model and the hardware parameters used.

4.1 System Model

Our simulator provides both average resource utilization and more detailed performance information. In the simplest case we can determine average processor, disk, and network utilizations by simulating cache behavior on the trace input and counting accesses to the different hardware resources. We get more detailed performance information by adding an event driven model to the cache simulation to measure the response time of different requests and monitor the burstiness of the utilization of different parts of the system. This event-driven hardware model includes both hardware and queuing delays. The rest of this subsection provides details about the simulated caches.

Our simulations of xFS and AFS include both on-disk and in-memory client file caches, and our AFS simulations include an in-memory file cache at the server. These caches are simulated using whole-file caching for simplicity, although in practice both AFS and xFS would cache chunks of files. We do break large transfers into 64 KB chunks for realistic latency measurements. We assume in-memory caches of 8 MB per client and 128 MB at the AFS server, and we give each client a 100 MB on-disk file cache.

Our simulations also include attribute caches used when clients access a file's attributes without fetching the entire file. Each client had a 2048 entry in-memory attribute cache backed by its disk, and the server has a 32,768 entry in-memory cache. The server supplies attributes and the systems maintain attribute consistency using the same protocols used for the files themselves.

Because cache behavior is so crucial to performance of large scale file systems, we warm the caches before gathering statistics. The results presented in this paper are gathered during the last six days of our seven day trace, after warming the caches for the first day, a Saturday. Figure 1 plots the hit rate of read opens not satisfied completely in the in-memory cache over time and indicates that after the first day, the hit rate fluctuates between 30% and 95%. There appears to be no general upward trend as we would expect once the caches are warm. The steady state hit rate is relatively low because opens that are completely satisfied in the NFS local in-memory cache did not appear in the trace.

Even after warming the caches, 8% of the read opens (21% of those that miss on the local disk) access files that have not been referenced earlier in the trace. We must make some assumption about which xFS client owns these files. We arbitrarily assume each file with unknown location is stored on a randomly selected client disk. The impact of this

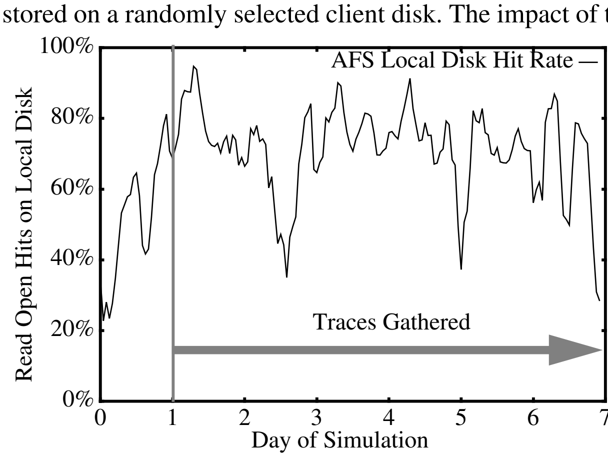

Figure 1. Local hit rate over time. This plot of local on-disk cache hit rate against day of the simulation of AFS suggests that one day is sufficient to warm the caches. Hour-to-hour fluctuations have been smoothed in this plot by averaging over the previous 4 hours for each point.

assumption is limited since 92% of the file opens are located normally, at a client that is currently caching the data.

4.2 Hardware Parameters

To estimate the performance of xFS and compare it to AFS, absent an implementation, we parameterized the model to reflect the performance of a mid-range workstation for tasks similar to those that would be performed in an xFS or AFS implementation. We approximate the performance of a DECstation 5000/200 using measured and reported performance results for its subsystems summarized in Figure 2. Each hardware resource services a *requestSize* request in time *overhead+requestSize/bandwidth*. This approach is clearly an oversimplification: not all requests to a given piece of hardware will have the same overheads and bandwidths and the actual overheads are unlikely to exactly match those for current systems. Nevertheless, these simple assumptions provide a starting point for system evaluation. [Lazo86] used a similar approach in parameterizing performance for network file system simulations.

The processor overhead time represents the CPU and memory subsystem time to send or receive one network request and do a small amount of work in the file system. We estimated this time by measuring the time for a DECstation 5000/200 to handle an NFS `getattr` request. For the CPU bandwidth for large requests, we use the time to supply file system data from the in-memory file cache reported in [Chen93].

We assume that the machines use disks that rotate at 5400 RPM, that the typical seek time is 4ms[2], and that the disk bandwidth is 2MB/s.

We base the network topology on the configuration of the clients in our NFS trace: four subnets each connected to the server. For AFS, each subnet connects directly to the server, and for xFS each subnet connects to a cluster server. The cluster servers connect to the central server using a fifth subnet. The network latency is the time to transmit a minimum-sized Ethernet packet and the network bandwidth is an optimistic estimate of the net bandwidth available on an Ethernet.

While other performance assumptions would result in differences in the absolute latency and burstiness numbers reported later, they are unlikely to affect our central conclusion that the xFS protocol scales significantly better than AFS.

	Overhead	Bandwidth
Processor	1.4 ms	7 MB/s
Disk	9.6 ms	2 MB/s
Network	0.1 ms	4 or 5 x 1 MB/s

Figure 2. Service demand parameters.

2. This estimate of the typical disk overhead differs from the "average" seek time reported by manufacturers because it accounts for locality seen in real workloads [Henn90] while the manufacturer-reported average seek is the mean time over all possible source and destination tracks—seeks that average one third of the distance across the disk surface.

5 Results

This section presents the results of our simulations. We show that the proposed optimizations reduce server load by more than a factor of six compared to AFS, and we also show that the xFS protocol greatly reduces peak bursts of server load. xFS also significantly reduces total network load, and the distribution of traffic that remains is better suited for a mixed LAN/WAN environment. The increased responsibility this protocol places on clients does increase client file system load slightly, but the extra forwarding of read requests does not increase response time.

The next subsection presents our overall results in more detail, and the subsection after that examines the individual impact of each of the four main aspects of xFS: its no write through policy, client-to-client transfers, write ownership, and clustering. We find that all four techniques make significant contributions to the overall performance.

5.1 Overall Results

This section compares the xFS protocol to AFS in terms of server CPU load, the burstiness of server load, response time, network load, and client load.

Figure 3 summarizes our results showing that xFS reduces server load by more than a factor of six compared to AFS. This load estimate is the total server processor demand including both overhead and bandwidth as described in Section 4.2, expressed as a fraction of AFS's server demand. We find that xFS reduces server load by 85% compared to AFS by eliminating data transferred at the server and by reducing the number of messages the server must handle.

Figure 4 details how much server load each type of operation demands. Write close operations at the server include write through (for AFS), notifying the server of a write of a file that is not write owned (for xFS), and the messages sent by the server to invalidate cached copies. Read open operations at the server are caused by client misses and include handling the client request and supplying the data (for AFS) or forwarding the request (for xFS). Delete operations include the message sent to the server to indicate that a file has been deleted and the server messages notifying the clients caching that file. Attribute messages include packets to request, update, and invalidate file attribute information. The category "other" includes all other packets sent to or

	Server Messages	Server Data	Server Load
AFS	1,411,504	15.2 GB	1.000
xFS	457,356	0.0 GB	0.153

Figure 3. Total server load. The normalized server load expresses the server CPU load for the simulated protocols as a fraction of the simulated server load for AFS.

received by the server; an xFS client notifies the server when it purges a file (potentially forwarding marked data to another cache) to make room for new data.

In addition to the total work at the server, performance and scalability will also be dependant on periods of heavy load. Figure 5 summarizes the distribution of time spent at increasing levels of server load for AFS and xFS. It shows that xFS's reduction in average load translates into a reduction in time spent at high load. This figure indicates that the AFS server spends several minutes per day working at loads of over 0.5 while the xFS server is never loaded that heavily. The extremely low peak demands of xFS suggest that we could scale the system to a larger number of clients than AFS. We note that the absolute load level for both machines is relatively low, suggesting that either server could probably handle the Berkeley Auspex workload. As we noted earlier, however, the maximum load that either system experiences in this simulation is limited by the maximum load accepted by the NFS system where the workload trace was gathered.

Having servers forward read requests can potentially increase latency. Our measurements show, however, that the aggregate effect of the client in-memory caches minimizes the impact of the extra step. We focus on the time spent to open a file for reading, from when the request is issued until the first chunk of up to 64 KB arrives on the local disk. We consider both requests that are found on the local disk without additional network communication and requests that are satisfied over the network.

	AFS	xFS
Write Close	0.429	0.018
Read Open	0.453	0.073
Delete	0.014	0.013
Attribute	0.104	0.043
Other	0.000	0.006
Total	**1.000**	**0.153**

Figure 4. Server load breakdown. Portion of AFS server load due to each type of request.

Figure 6 breaks down the response time based on where requested files are found. Opens that are satisfied locally, requiring no disk accesses, account for most of the opens and are satisfied quickly by both systems. Misses that are

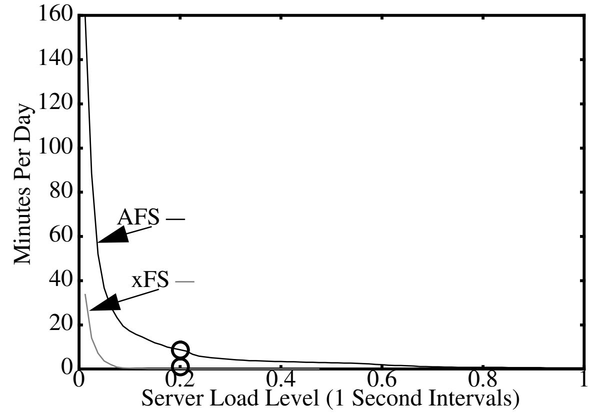

Figure 5. Cumulative distribution of server load. The X axis is the amount of load presented to the server during a one second interval. This load is the sum of the service demands for all requests that arrive at the server during a one second interval. The Y value is the amount of time during the day that the server experienced at least that load level. The AFS server handled at least one request per second for 159 minutes per day, while the xFS server was completely idle for all but 34 minutes per day. The circled points indicate that the AFS server would have a load of more than 0.2 for over eight minutes per day while the xFS server would have a load that high less than ten seconds per day.

	AFS		**xFS**	
	Freq.	**Time**	**Freq.**	**Time**
Local	60%	6.3 ms	60%	6.3 ms
Remote	40%	57 ms	40%	56 ms
Mem	9%	25 ms	15%	30 ms
Disk	31%	66 ms	25%	71 ms
Total	**100%**	**27 ms**	**100%**	**26 ms**

Figure 6. Read open response time. The response time is the time needed to put the first chunk of the opened file onto the local disk and return. Local hits are data that are already on the local disk. For both AFS and xFS data not found in the local on-disk cache are fetched from a remote machine. For AFS that remote machine is the server but for xFS that remote machine is another client. At that remote machine the desired data may be found on disk or in the in-memory cache.

Protocol	Write	Read	Delete	Attr.	Other	Total
AFS	0.429	0.453	0.014	0.104	0.000	1.000
+ no write through	***0.113***	***0.496***	0.014	0.102	***0.003***	***0.728***
+ client-to-client	0.113	***0.175***	0.014	0.102	***0.015***	***0.418***
+ write ownership	***0.032***	0.175	0.014	0.102	0.015	***0.337***
+ clusters (full xFS)	***0.018***	***0.073***	***0.013***	***0.043***	***0.006***	***0.153***

Figure 11. Server load by type of activity and protocol. The AFS line indicates the server load stemming from write closes, read opens, deletes, attribute operations, and other operations. Each subsequent line shows the breakdown and total after one more part of the xFS protocol is added. Server loads that changed significantly from the previous line are highlighted.

satisfied in the remote in-memory cache and misses that require remote disk accesses are slightly slower in the xFS implementation because of the extra forwarding step. This does not, however, increase the cost of a miss because xFS misses are satisfied by the remote client in-memory cache more often than AFS requests are satisfied by the server in-memory cache. The higher remote client in-memory cache hit rate is initially surprising because the AFS server in-memory cache is 128 MB while each xFS client cache is just 8 MB. We note, however, that the aggregate size of the 237 client caches is 1896 MB, making them together an effective file cache even though many of the files stored in this distributed cache are duplicates. Further, note that the server does not attempt to keep track of which clients have a file cached in memory rather than on disk; although this would be an obvious optimization, it could increase server load. Figure 7 plots the remote in-memory hit rate for xFS as a function of client memory cache size and indicates that the 128 MB server cache is about equivalent to 5 MB client caches.

Network load, the number of bytes transferred over the network during the trace, is an important metric of scalability. We are particularly concerned about minimizing network usage for wide area network file systems, where network bandwidth can become a bottleneck. Bandwidth can also be an issue for mobile computing using wireless interconnects [Kist92].

Figure 8 summarizes total network traffic for AFS and xFS using the assumption that each packet sent has a header of 128 bytes. The table indicates that xFS reduces total network traffic by 52%. The major difference in total network bandwidth is xFS's elimination of write through traffic for files that are later modified or deleted by the same client.

Although xFS only reduces total network traffic by a factor of two compared to AFS, it significantly changes the nature of that traffic. Figure 9 shows that client-to-client transfers reduce the bytes transferred to the server by more than 99%. This reduction is crucial for file systems where the server may be located across a WAN. The use of clustering also reduces the number of bytes transferred out of the cluster to less than 20% of AFS's total traffic, a consideration when clusters are separated by gateways or WANs.

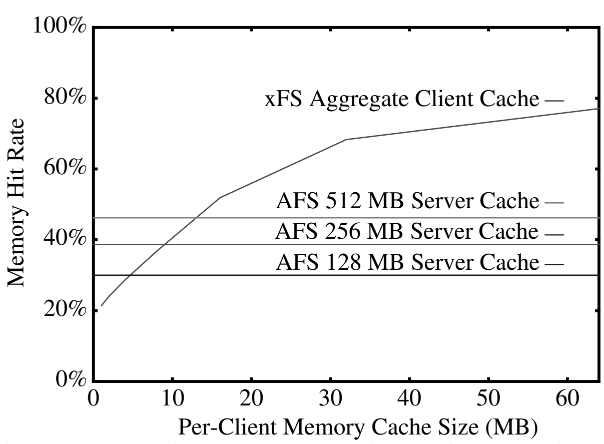

Figure 7. Remote in-memory cache hit rates for local misses.

Finally, we note that overall client load is increased only slightly even though clients shoulder considerably more responsibility in xFS than in AFS, for instance by supplying data to other clients. The xFS protocol increases the total amount of file system work done by clients by 10% for the measured workload. The increase in client load is small because most of a client's load comes from local block reads and writes which are unchanged in xFS. Further, although xFS increases the load on each client to handle data requests from other clients, the reduced write through activity largely offsets this increase. This fraction would be smaller still if the trace included the even larger amount of file system activity that is purely local, such as reads that hit the client's in-memory cache. Figure 10 shows that the demands on clients are not greatly altered by the xFS protocol.

5.2 Protocol Breakdown

In the previous section, we considered the aggregate effect of all four optimizations studied. Here we consider their individual effects. We conclude that each of the optimizations contributes significantly to the performance of the xFS protocol. Figure 11 summarizes the load as each part of the xFS protocol is added to the system. This section explains the benefits of each of the strategies in more detail.

Simply eliminating write through from AFS would reduce the server load by 27% for this workload. The server load associated with closing files that have been written

	AFS	**xFS**
Packets	1,411,504	1,968,242
Overhead	180 MB	252 MB
Data Bytes	15,251 MB	7,222 MB
Write Through	8,096 MB	0 MB
Other Data	7,154 MB	7,222 MB
Total Bytes	15,431 MB	7,474 MB

Figure 8. Network traffic for xFS and AFS from all sources. The total bytes transferred is an estimate formed by adding the total data bytes transferred plus 128 bytes per request to reflect protocol overhead and control information. The packet count for xFS reported here differs from the number of messages reported in Figure 3 because Figure 3 only considered traffic to and from the server.

	AFS	**xFS**
Central Server	15,431 MB	58 MB
Other Out Cluster	N/A	2,902 MB
In Cluster	N/A	4,514 MB

Figure 9. Total network traffic over different parts of the network. The total includes both data and a 128 byte per-packet header. In a WAN or large-scale environment the connection to the central server or to other clusters may be slower than the network within a cluster.

would be reduced by nearly a factor of four because clients only need to send a small notification message to the server rather than transmitting the modified file in one or more larger messages. However, the server load associated with supplying read misses is increased slightly as the server endures write backs of modified files that other clients want to read. 10% of the bytes that were written to the server by AFS are later read and show up as increased read load. Another small load, in the category *Other*, comes from write throughs that must eventually be made to free cache space.

Utilizing client-to-client data transfers reduces the server load by an amount equal to 0.31 times the original AFS load. This reduction comes from the elimination of data transfers through the server on read opens. Note, however, that the work in the category *Other* is increased slightly. This increase is from messages clients send when they free space by discarding files from their caches. The server must be informed when even clean files are discarded so that it doesn't forward read requests to a client no longer caching the desired data.

Write ownership reduces the number of messages processed by the server, and therefore server load, by an additional 25% compared to the client-to-client line. Figure 12 indicates that over 80% of the messages notifying the server that a file has been closed are eliminated using write ownership.

Cluster servers reduce messages of all types by intercepting requests that would have been handled by the central server. Cluster servers reduce the server load for reading files and reading attributes by forwarding requests that can be satisfied within the cluster. This read forwarding is the primary benefit of cluster servers. The number of write close messages is also reduced, and this reduction comes from two sources that combine to reduce server load by 0.014 times AFS's load. First, a few write closes transfer ownership between clients of the same cluster. These writes are handled by the cluster server, reducing central server load by 0.003 times AFS's original load. More significantly, the cluster server acts to fan out invalidation packets from the central server. One invalidation packet from the central server to the cluster server is sufficient to invalidate all data copies in the cluster. These invalidations reduce the central server load by 0.011 times the original load. The load of delete messages and *Other* messages processed by the central server is also reduced slightly. Some delete messages, invalidating multiple copies of the same file in caches in one cluster, are eliminated because the cluster server distributes these messages to the appropriate clients in the cluster. *Other* messages are reduced when a file discarded from a cache is still cached in some client in the cluster; in that case the server may still forward requests for that file to the cluster and so need not be notified of the change. Finally, note that the four cluster servers' loads ranged from 0.05 to 0.20 as a fraction of the AFS server load. In other words, they have about the same load as the xFS central server.

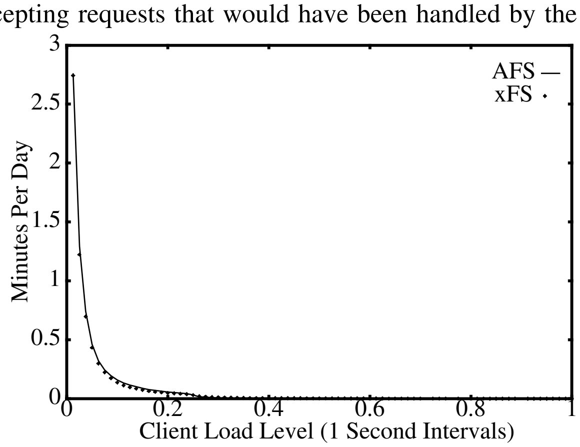

Figure 10. Cumulative distribution of one-second load levels at clients. The average client processor is active doing file system activity for less than three minutes per day. The xFS and AFS client loads are almost indistinguishable.

Protocol	Notify Server	Invalidate Client
no write through	291,198	46,015
+ownership	50,423	46,015

Figure 12. Write close messages without and with write ownership. *Notify Server* messages tell the server to invalidate any other cached copies of a file. It invalidates files with *Invalidate Client* messages.

We also considered extending the strategy of write ownership of files to include ownership of directory subtrees to allow us to avoid notifying the server of all file creations and deletions. If a client owned a directory, it would notify the server of file creations and deletions when ownership of the directory containing the file was lost. This strategy would exploit the common case of files being created, used, and deleted without ever being seen by another client. Our simulations did not measure the benefits of directory ownership, but they allow us to place an upper bound on the benefits by noting that of the 0.018 load for write closes, 0.009 was for newly created files and of the 0.013 load for deletes, 0.012 was for messages notifying the server of the delete. If all of these messages could be omitted, the total server load would be reduced by 0.021 units, a 14% reduction from the xFS protocol in this unrealistically optimistic case. We conclude that this improvement would not justify the considerable added complexity of the approach. If some of the other sources of load were reduced further, this reduction would be a more significant fraction of the remaining load, and this decision would have to be reexamined.

6 Challenges for Decentralized Operation

Although the xFS protocol's reliance on client disk caches improves scalability, it introduces three potential challenges for reliable operation. We must provide backup that scales with the number of clients; we must ensure that the files are highly available despite being distributed over many disks; and we must provide security guarantees so that unauthorized clients cannot read or change data they store. We find that the data replication that is a natural part of the xFS protocol makes backup easier and increases availability. Also, message digests can be used to provide security for data supplied by other clients.

6.1 Backup

xFS's ability to manage multiple data copies in normal operation can be used to manage the backup copies of the data as well. This simplifies the design of the system and also allows us to use multiple backup archives to scale the backup bandwidth as the rest of the system scales.

xFS treats each archive as another client, and the server keeps track of the backup copies of data just as it tracks other cached copies. A client cache backs up a file by sending it to an archive and telling the server about the new copy. The system must have policies for the frequency of backup, for deciding which of the potentially many clients caching a file is responsible for backing it up (this decision can be made without additional communication), and for retrieving data from the backing store.

We plan to use tertiary storage robots to manage the backup media. Tertiary storage robots provide from hundreds of gigabytes to tens of terabytes of storage with file access times measured in tens of seconds [Katz91]. The robots provide deep storage as traditional tape systems do, but they have the added advantage that all files are on-line in the sense that a user may access the data without human intervention. Tertiary robots are not a requirement of xFS; backup could be done using traditional off-line tapes. The advantage of using storage robots for backup is that data may be sent to or retrieved from the tertiary storage system without operator intervention, allowing the system to automatically provide services such as access to old versions of files or deleted files.

Note that the server's disk need not be backed up; the server can reconstruct its list of cached copies and metadata by polling the cluster servers [Nels88].

Backup over the network exerts a small additional load on the system. Although we did not include this load in the simulations, its impact on performance should be small. As noted in Section 2.2, almost all files are overwritten or deleted quickly and so need not be copied from the client to the backup archive. Further, backup may be scheduled for periods of low system load to avoid disturbing regular system activity.

6.2 Availability

xFS's second challenge is availability. As the file system is spread over more machines, the probability that one of the machines containing file system data is unavailable increases. Availability problems are mitigated by large client caches, file replication, and the file access patterns observed in our trace. Even higher availability could be achieved using explicit data replication.

Large client caches and file replication from caching and backup reduce xFS's vulnerability to unavailable clients. Large client caches provide some insulation—the crash of one machine will often not be noticed by others [Kist92]. xFS also automatically stores redundant copies of shared read files in different caches increasing the availability of those files, and on-line backup provides added copies of older files. These properties of xFS mean that only a few files, those recently written but not backed up and not read by a second client, are vulnerable to single point failures. Since files written by one client are seldom read by another, these vulnerable files are seldom accessed when their writer is down.

We estimate from our trace of file activity that the average client will go hundreds of days without noticing file unavailability stemming from the crash of another client. This low rate suggests that the xFS protocol will not significantly change data availability which will still be dominated by the availability of the server. For this calculation we assumed that clients fail randomly with an exponentially distributed mean time to failure of 30 days and an exponentially distributed mean time to repair of one hour. We also assumed that data was backed up to a reliable on-line tape robot every morning at 2 AM. Under these assumptions we found that each day an average of 0.56 of the 237 clients in the trace would try to access data that was cached only on an unavailable client. This figure was based on 500,000 seven day trials and has a 95% confidence interval of ±0.04. Availability could be more of a problem if write-sharing of data were more widespread than seen in our trace. Also, if a user's machine crashes, the user may not be able to switch to an alternate machine to do work since modified data will unavailable until the crashed machine recovers.

If stronger availability guarantees are needed, client-to-client data replication of recently modified data provides a scalable solution. Shortly after a client closes a file for writing, it would send the data to one or more other clients [Lisk91, Birr93]. This solution is scalable since it adds no additional server messages if the server knows ahead of time which clients mirror writes to each other. The copy delay chosen is a trade-off between performance and availability guarantees with longer delays significantly reducing client-to-client bandwidth [Bake91] while increasing the length of time the file is vulnerable to a single point failure. In the future we plan to investigate these trade-offs for client-to-client transfers and also plan to look at using striping to reduce the cost of high availability. Our current simulations do not make additional client copies.

6.3 Security

xFS's use of client disks to store and supply data raises two security concerns, data confidentiality and integrity. We do not want clients to transfer data into a cache that is not authorized to read the data, and we do not want to accept altered data from a malicious client on a client-to-client transfer.

We believe that in many environments most clients will trust at least the other clients in the same cluster to enforce the system's data access rules. Communication between trusting clients does not require the steps described here.

The confidentiality of data cached on client disks is also addressed by AFS [Saty89] and the techniques used there apply to xFS as well, for data read by the client. xFS, however, adds one new way that data can be brought into a client's cache: clients flush data to one another as their caches fill. Since this flushing is rare, the performance impact of the chosen strategy will be limited. In the extreme case, if no other clients are trusted with the data, the data could be encrypted before it is flushed. The same client would have to decrypt it if it were later accessed. More commonly, data will only be flushed to a limited subset of trusted clients, for instance only to clients in the same cluster and administrative domain.

xFS can guarantee data integrity, allowing clients to accept data from even untrusted clients, by guaranteeing two things: that a secure copy of each file always exists and that a client can detect when a file has been altered from the secure image.

To guarantee the existence of pristine data, the system must trust the client that created the data—which it must do in any event since it has given the client permission to modify the data—and must trust the on-line backup archive. The client that created the data pins a copy in its cache until the file is backed up to the robotic storage. After the file is backed up, the writer is free to flush the data, since if another client modifies it without permission, the system may still recover the file from the tape robot.

A client verifies untrusted data using a *message digest*, a special checksum that can be calculated efficiently, but for which it is computationally infeasible to create different data to match [Rive92]. The server stores a 128-bit digest for each 64 KB data chunk with its list of chunks cached at the clients. When it forwards a client's request for data, it includes the digest for the pristine data in the forwarding packet. The digest in the forwarding packet is protected using an encrypted digital signature [NIS92]. The protected part of the forwarding packet would also include a request identifier to protect against playback attacks. The client supplying data forwards the protected digest along with the data. The original client then verifies the data supplied against the original digest. If the file is corrupt, the client asks the server for a copy from another source.

Because the work of computing digests is done at the clients, digests do not severely impact server scalability. Digests can be supplied to clients reading data using no additional network packets and they are updated at the server only when file write ownership is lost. When the server forwards a data read request, it must encrypt a short message including the digest and request identifier and append that message to the forwarding packet. If the work of encrypting this message is small compared to sending the packet, digests will not increase server load for read requests. (If encryption is hard compared to sending a message, the unprotected digest may be sent directly to the client requesting the read in a separate message.) Our simulation assumes that the cluster servers are trusted by the clients in the cluster, so once a cluster server knows the digest for a particular file chunk, the cluster server may forward the digest to the appropriate client. Digests only change when a file is written, so clients only calculate a new digest and send it to the server when they lose write ownership.

We simulated message digests assuming that the signature encryption is cheap compared to sending a message. In that case the only additional server load is receiving 15,510 digest updates when write ownership is lost. This increases server load by less than 1% of AFS's total server load.

Message digests do not severely impact response time. We measured the bandwidth to compute the MD4 digest on a DEC Alpha AXP 3000/400 to be 13.3 MB/s. To be consistent with the other processor speeds used in this paper, we simulated MD4 calculations using our measured DECstation 5000/200 MD4 bandwidth of 2.5 MB/s. Even if clients calculate a message digest on all data received, trusting no other clients, the read open time for files is 28 ms, just 2 ms slower than the 26 ms read open time reported in Section 5 for xFS without message digests. Since the MD4 calculation bandwidth exceeds the bandwidth of the network, the impact to performance is minimal. This approach will become even more attractive if processor speed improvements continue to outpace I/O system improvements.

7 Related Work

This paper evaluates the effectiveness of a file system that combines the strategies of eliminating write through, client-to-client transfers, write ownership, and clustering. The fusion of these strategies has produced a system that we believe will scale in size and across wide area networks. This section surveys some other combinations of these schemes that have been suggested as methods to achieve scalable file systems.

The Andrew file system, AFS, was designed with scalability as a main criteria [Howa88, Saty90]. Andrew based scalability on the use of, first, large on-disk client caches to reduce file reads from the server, and second, callbacks to reduce the number of protocol messages handled by the server. xFS is also based on large on-disk client caches and callbacks but generalizes their use using four additional techniques.

The mass storage system reference model [Coyn93] decouples location and name service from the actual storage of data. The model defines a name server and location server that locate the storage server that actually manages the bitfile. Goldick et al. [Gold93] have implemented an AFS-based storage system which allows data to reside in up to 32 separate locations. In xFS each client acts as a storage server, and server and cluster servers together act as a two-level location server. This study indicated that this division greatly reduced the load on the central resource, the central server.

Sprite [Nels88] uses delayed writes to the server to reduce server load. The diskless Sprite clients, however, must write data through to the server within about 30 seconds to reduce vulnerability to crashes. xFS's extends delayed writes to a no write through policy by using the clients' local disks.

Blaze and Alonso [Blaz91, Blaz92] suggest dynamically building hierarchies for widely shared data. Once a server has supplied a threshold number of copies of a file, the server will refuse to supply the data to any more clients. Instead, the request will be forwarded to a client already caching the file. Clients acting as intermediate servers are also responsible for keeping callback information on the files they have supplied to other caches. The authors also suggest a number of strategies which clients may use to guess which other client has desired data without going to the server. These hinting techniques could be applied to an xFS implementation.

Muntz and Honeyman [Munt92] studied the effect of putting an intermediate data server between the central server and the clients in an Andrew system. They found that the hit rates at the intermediate server were surprisingly low. Client caches of 40 MB, small for an on-disk cache, reduced the intermediate cache hit rate to under 20% for both traces studied. The reason is that it is difficult to give the intermediate server a big enough cache to hold significant amounts of data not found in client caches. Because of this result, xFS is designed with intermediate servers that field only consistency requests; the intermediate servers do not store data. We believe it is feasible to provide enough storage on the intermediate servers to hold all of a cluster's consistency information

The Frolic system [Pang92, Sand92] implements replication of files among cluster servers. When a client accesses data from a remote cluster, Frolic creates a copy of the data in the local cluster. Clients use a different protocol, such as NFS, to access data from the local cluster server. Frolic cluster servers differ from xFS cluster servers in that Frolic cluster servers act as intermediate data caches between the clients and remote servers while xFS's cluster servers merely monitor the location of file copies within the cluster. Frolic's concept of a "locating server" responsible for tracking the current owner of a file is similar to xFS's use of the central server. The authors studied the behavior of shared files using a synthetic workload and found that cluster replication improved performance and server load for shared files unless the "degree of cluster locality" was low; clusters do not perform well if files are read by one cluster and quickly invalidated by another.

8 Conclusions

In this paper, we present and evaluate the xFS caching protocol, designed to improve network file system scalability by taking full advantage of clients' processors, memories, and disks. All files are stored at the clients and all data transfers go directly from client to client. The server is used only to coordinate transfers among the clients. xFS reduces the server load necessary for this coordination by using write ownership and clustering, in most cases allowing clients and cluster servers to avoid interacting with the central server.

We evaluated the performance of xFS using a trace-driven simulation of 237 clients. We found that xFS reduced server load by 85% compared to AFS by eliminating server data transfers and by reducing the number of messages to and from the server by 68%. By moving data storage and data transfer responsibilities to the clients, xFS makes it possible to build a large network file system using only commodity desktop workstations, even for the file server.

Acknowledgments

We would like to thank Matt Blaze for providing us with his rpcspy and nfstrace tools; these formed the basis for our trace processing tools. We would also like to thank John Hartman and the anonymous referees whose comments were very helpful in improving this paper.

References

[Arch86] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4:273–298, November 1986.

[Bake91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[Birr93] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garett Swart. The Echo Distributed File System. Technical Report 111, Digital Equipment Corp. Systems Research Center, 1993.

[Blaz91] Matt Blaze and Rafael Alonso. Long-Term Caching Strategies for Very Large Distributed File

Systems. In *Proc. of the Summer 1991 USENIX*, pages 3–15, June 1991.

[Blaz92] Matt Blaze and Rafael Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proc. of the 12th International Conf. on Distributed Computing Systems*, pages 521–528, June 1992.

[Blaz93] Matt Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[Chen93] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proc. of 1993 ACM SIGMETRICS*, pages 1–12, May 1993.

[Coyn93] Robert A. Coyne and Harry Hulen. An Introduction to the Mass Storage System Reference Model, Version 5. In *Twelfth IEEE Symposium on Mass Storage Systems*, pages 47–53, April 1993.

[Gold93] Jonathan S. Goldick, Kathy Benninger, Woody Brown, Christopher Kirby, Christopher Maher, Daniel S. Nydick, and Bill Zumach. An AFS-Based Supercomputing Environment. In *Twelfth IEEE Symposium on Mass Storage Systems*, pages 127–132, April 1993.

[Hage92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM–A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):45–54, 1992.

[Henn90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[Hitz90] David Hitz, Guy Harris, James K. Lau, and Allan M. Schwartz. Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers. In *Proc. of the Winter 1990 USENIX*, pages 285–296, 1990.

[Howa88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[Katz91] Randy H. Katz, Thomas E. Anderson, John K. Ousterhout, and David A. Patterson. Robo-Line Storage: Low Latency High Capacity Storage Systems Over Geographically Distributed Networks. Sequoia 2000 Technical Report 91/3, University of California, September 1991.

[Kist92] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[Lazo86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[Leno90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Lisk91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *Proc. of the 13th Symposium on Operating Systems Principles*, pages 226–238, October 1991.

[Mogu87] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, 1987.

[Munt92] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In *Proc. of the Winter 1992 USENIX*, pages 305–313, January 1992.

[Nels88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[NIS92] The Digital Signature Standard Proposed by NIST. *Communications of the ACM*, 35(7):36–40, July 1992.

[Pang92] James Y.C. Pang, Deepinder S. Gill, and Songnian Zhou. Implementation and Performance of Cluster-Based File Replication in Large-Scale Distributed Systems. Technical report, Computer Science Research Institute, University of Toronto, August 1992.

[Rive92] R. Rivest. The MD4 Message-Digest Algorithm. Request for Comments 1320, Network Working Group, ISI, April 1992.

[Rost93] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L.W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proc. of 1993 ACM SIGMETRICS*, pages 74–85, 1993.

[Sand85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. of the Summer 1985 USENIX*, pages 119–130, June 1985.

[Sand92] Harjinder S. Sandhu and Songnian Zhou. Cluster-Based File Replication in Large-Scale Distributed Systems. In *Proc. of 1992 ACM SIGMETRICS*, pages 91–102, June 1992.

[Saty89] Mahadev Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, pages 247–280, August 1989.

[Saty90] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, pages 9–21, May 1990.

[Spas94] Marjana Spasojevic and M. Satyanarayanan. A Usage Profile and Evaluation of a Wide-Area Distributed File System. In *Proc. of the Winter 1994 USENIX*, January 1994.

[Thom87] James Gordon Thompson. *Efficient Analysis of Caching Systems*. PhD thesis, University of California at Berkeley, 1987.

[Wolf89] Joel Wolf. The Placement Optimization Problem: A Practical Solution to the Disk File Assignment Problem. In *Proc. of 1989 ACM SIGMETRICS*, pages 1–10, May 1989.