Serverless Network File Systems

THOMAS E. ANDERSON, MICHAEL D. DAHLIN, JEANNA M. NEEFE,
DAVID A. PATTERSON, DREW S. ROSELLI, and RANDOLPH Y. WANG
University of California at Berkeley

We propose a new paradigm for network file system design: *serverless network file systems*. While traditional network file systems rely on a central server machine, a serverless system utilizes workstations cooperating as peers to provide all file system services. Any machine in the system can store, cache, or control any block of data. Our approach uses this location independence, in combination with fast local area networks, to provide better performance and scalability than traditional file systems. Furthermore, because any machine in the system can assume the responsibilities of a failed component, our serverless design also provides high availability via redundant data storage. To demonstrate our approach, we have implemented a prototype serverless network file system called xFS. Preliminary performance measurements suggest that our architecture achieves its goal of scalability. For instance, in a 32-node xFS system with 32 active clients, each client receives nearly as much read or write throughput as it would see if it were the only active client.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies*; *secondary storage*; D.4.3 [**Operating Systems**]: File Systems Management—*access methods*; *directory structures*; *distributed file systems*; *file organization*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault tolerance*; D.4.8 [**Operating Systems**]: Performance—*measurement*; *simulation*; E.5 [**Data**]: Files—*organization/structure*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*

General Terms: Algorithms, Design, Measurement, Performance, Reliability

Additional Key Words and Phrases: Log-based striping, log cleaning, logging, log structured, RAID, redundant data storage, scalable performance

This work is supported in part by the Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett-Packard, IBM, Siemens Corporation, Sun Microsystems, and Xerox Corporation. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship, Neefe by a National Science Foundation Graduate Research Fellowship, and Roselli by a Department of Education GAANN fellowship.

Authors' address: Computer Science Division, University of California at Berkeley, 387 Soda Hall, Berkeley, CA 94720-1776; email: {tea; dahlin; neefe; patterson; drew; rywang}@cs.Berkeley.edu; additional information about xFS can be found at http://now.CS.Berkeley.edu/Xfs/xfs.html.

1. INTRODUCTION

A serverless network file system distributes storage, cache, and control over cooperating workstations. This approach contrasts with traditional file systems such as Netware [Major et al. 1994], NFS [Sandberg et al. 1985], Andrew [Howard et al. 1988], and Sprite [Nelson et al. 1988] where a central server machine stores all data and satisfies all client cache misses. Such a central server is both a performance and reliability bottleneck. A serverless system, on the other hand, distributes control processing and data storage to achieve scalable high performance, migrates the responsibilities of failed components to the remaining machines to provide high availability, and scales gracefully to simplify system management.

Three factors motivate our work on serverless network file systems: the opportunity provided by fast switched LANs, the expanding demands of users, and the fundamental limitations of central server systems.

The recent introduction of switched local area networks such as ATM or Myrinet [Boden et al. 1995] enables serverlessness by providing aggregate bandwidth that scales with the number of machines on the network. In contrast, shared-media networks such as Ethernet or FDDI allow only one client or server to transmit at a time. In addition, the move toward low-latency network interfaces [Basu et al. 1995; von Eicken 1992] enables closer cooperation between machines than has been possible in the past. The result is that a LAN can be used as an I/O backplane, harnessing physically distributed processors, memory, and disks into a single system.

Next-generation networks not only enable serverlessness; they require it by allowing applications to place increasing demands on the file system. The I/O demands of traditional applications have been increasing over time [Baker et al. 1991]; new applications enabled by fast networks—such as multimedia, process migration, and parallel processing—will further pressure file systems to provide increased performance. For instance, continuous-media workloads will increase file system demands; even a few workstations simultaneously running video applications would swamp a traditional central server [Rashid 1994]. Coordinated Networks of Workstations (NOWs) allow users to migrate jobs among many machines and permit networked workstations to run parallel jobs [Anderson et al. 1995; Douglas and Ousterhout 1991; Litzkow and Solomon 1992]. By increasing the peak processing power available to users, NOWs increase peak demands on the file system [Cypher et al. 1993].

Unfortunately, current centralized file system designs fundamentally limit performance and availability since all read misses and all disk writes go through the central server. To address such performance limitations, users resort to costly schemes to try to scale these fundamentally unscalable file systems. Some installations rely on specialized server machines configured with multiple processors, I/O channels, and I/O processors. Alas, such machines cost significantly more than desktop workstations for a given amount of computing or I/O capacity. Many installations also attempt to achieve scalability by distributing a file system among multiple servers by partitioning the directory tree across multiple mount points. This approach only moderately improves scalability because its coarse distribution often results in hot spots when the partitioning allocates heavily used files and directory trees to a single server [Wolf 1989]. It is also expensive, since it requires the (human) system manager to effectively become *part* of the file system—moving users, volumes, and disks among servers to balance load. Finally, Andrew [Howard et al. 1988] attempts to improve scalability by caching data on client disks. Although this made sense on an Ethernet, on today's fast LANs fetching data from local disk can be an order of magnitude slower than from server memory or remote striped disk.

Similarly, a central server represents a single point of failure, requiring server replication [Birrell et al. 1993; Kazar 1989; Kistler and Satyanarayanan 1992; Liskov et al. 1991; Popek et al. 1990; Walker et al. 1983] for high availability. Replication increases the cost and complexity of central servers and can increase latency on writes since the system must replicate data at multiple servers.

In contrast to central server designs, our objective is to build a truly distributed network file system—one with no central bottleneck. We have designed and implemented xFS, a prototype serverless network file system, to investigate this goal. xFS illustrates serverless design principles in three ways. First, xFS dynamically distributes control processing across the system on a per-file granularity by utilizing a new serverless management scheme. Second, xFS distributes its data storage across storage server disks by implementing a software RAID [Chen et al. 1994; Patterson et al. 1988] using log-based network striping similar to Zebra's [Hartman and Ousterhout 1995]. Finally, xFS eliminates central server caching by taking advantage of cooperative caching [Dahlin et al. 1994b; Leff et al. 1991] to harvest portions of client memory as a large, global file cache.

This article makes two sets of contributions. First, xFS synthesizes a number of recent innovations that, taken together, provide a basis for serverless file system design. xFS relies on previous work in areas such as scalable cache consistency (DASH [Lenoski et al. 1990] and Alewife [Chaiken et al. 1991]), cooperative caching, disk striping (RAID and Zebra), and log-structured file systems (Sprite LFS [Rosenblum and Ousterhout 1992] and BSD LFS [Seltzer et al. 1993]). Second, in addition to borrowing techniques developed in other projects, we have refined them to work well in our serverless system. For instance, we have transformed DASH's scalable cache consistency approach into a more general, distributed control system that is also fault tolerant. We have also improved upon Zebra to eliminate bottlenecks in its design by using distributed management, parallel cleaning, and subsets of storage servers called stripe groups. Finally, we have actually implemented cooperative caching, building on prior simulation results.

The primary limitation of our serverless approach is that it is only appropriate in a restricted environment—among machines that communicate over a fast network and that trust one another's kernels to enforce security. However, we expect such environments to be common in the future. For instance, NOW systems already provide high-speed networking and trust to run parallel and distributed jobs. Similarly, xFS could be used within a group or department where fast LANs connect machines and where uniform system administration and physical building security allow machines to trust one another. A file system based on serverless principles would also be appropriate for "scalable server" architectures currently being researched [Kubiatowicz and Agarwal 1993; Kuskin et al. 1994]. Untrusted clients can also benefit from the scalable, reliable, and cost-effective file service provided by a core of xFS machines via a more restrictive protocol such as NFS.

We have built a prototype that demonstrates most of xFS' key features, including distributed management, cooperative caching, and network disk striping with parity and multiple groups. As Section 7 details, however, several pieces of implementation remain to be done; most notably, we must still implement the cleaner and much of the recovery and dynamic reconfiguration code. The results in this article should thus be viewed as evidence that the serverless approach is promising, not as "proof" that it will succeed. We present both simulation results of the xFS design and a few preliminary measurements of the prototype. Because the prototype is largely untuned, a single xFS client's performance is slightly worse than that of a single NFS client; we are currently working to improve single-client performance to allow one xFS client to significantly outperform one NFS client by reading from or writing to the network-striped disks at its full network bandwidth. Nonetheless, the prototype does demonstrate remarkable scalability. For instance, in a 32-node xFS system with 32 clients, each client receives nearly as much read or write bandwidth as it would see if it were the only active client.

The rest of this article discusses these issues in more detail. Section 2 provides an overview of recent research results exploited in the xFS design. Section 3 explains how xFS distributes its data, metadata, and control. Section 4 describes xFS' distributed log cleaner. Section 5 outlines xFS' approach to high availability, and Section 6 addresses the issue of security and describes how xFS could be used in a mixed security environment. We describe our prototype in Section 7, including initial performance measurements. Section 8 describes related work, and Section 9 summarizes our conclusions.

2. BACKGROUND

xFS builds upon several recent and ongoing research efforts to achieve our goal of distributing all aspects of file service across the network. xFS' network disk storage exploits the high performance and availability of Redundant Arrays of Inexpensive Disks (RAIDs). We organize this storage in a log structure as in the Sprite and BSD Log-structured File Systems (LFS), largely because Zebra demonstrated how to exploit the synergy between RAID and LFS to provide high-performance, reliable writes to disks that are distributed across a network. To distribute control across the network, xFS draws inspiration from several multiprocessor cache consistency designs. Finally, since xFS has evolved from our initial proposal [Wang and Anderson

1993], we describe the relationship of the design presented here to previous versions of the xFS design.

2.1 RAID

xFS exploits RAID-style disk striping to provide high performance and highly available disk storage [Chen et al. 1994; Patterson et al. 1988]. 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 stores each data and parity block on a different disk. The parallelism of a RAID's multiple disks provides high bandwidth, while its parity 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 blocks. xFS uses single-parity disk striping to achieve the same benefits; in the future we plan to cope with multiple workstation or disk failures using multiple-parity blocks [Blaum et al. 1994].

RAIDs suffer from two limitations. First, the overhead of parity management can hurt performance for small writes; if the system does not simultaneously overwrite all $N - 1$ blocks of a stripe, it must first read the old parity and some of the old data from the disks to compute the 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. We expect cooperative caching—using workstation memory as a global cache—to further this workload trend. A second drawback of commercially available hardware RAID systems is that they are significantly more expensive than non-RAID commodity disks because the commercial RAIDs add special-purpose hardware to compute parity.

2.2 LFS

xFS implements log-structured storage based on the Sprite and BSD LFS prototypes [Rosenblum and Ousterhout 1992; Seltzer et al. 1993] because this approach provides high-performance writes, simple recovery, and a flexible method to locate file data stored on disk. LFS addresses the RAID small-write problem by buffering writes in memory and then committing them to disk in large, contiguous, fixed-sized groups called *log segments*; it threads these segments on disk to create a logical append-only log of file system modifications. When used with a RAID, each segment of the log spans a RAID stripe and is committed as a unit to avoid the need to recompute parity. LFS also simplifies failure recovery because all recent modifications are located near the end of the log.

Although log-based storage simplifies writes, it potentially complicates reads because any block could be located anywhere in the log, depending on when it was written. LFS' solution to this problem provides a general mechanism to handle location-independent data storage. LFS uses per-file *inodes*, similar to those of the Fast File System (FFS) [McKusick et al. 1984], to store pointers to the system's data blocks. However, where FFS' inodes reside in fixed locations, LFS' inodes move to the end of the log each time they are modified. When LFS writes a file's data block, moving it to the end of the log, it updates the file's inode to point 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*. The imap contains the current log pointers to the system's inodes; LFS stores the imap in memory and periodically checkpoints it to disk.

These checkpoints form a basis for LFS' efficient recovery procedure. After a crash, LFS reads the last checkpoint in the log and then *rolls forward*, reading the later segments in the log to find the new location of inodes that were written since the last checkpoint. When recovery completes, the imap contains pointers to all of the system's inodes, and the inodes contain pointers to all of the data blocks.

Another important aspect of LFS is its *log cleaner* that creates free disk space for new log segments using a form of generational garbage collection. 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 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. Furthermore, some workloads, such as transaction processing, incur larger cleaning overheads [Seltzer et al. 1993; 1995].

2.3 Zebra

Zebra [Hartman and Ousterhout 1995] provides a way to combine LFS and RAID so that both work well in a distributed environment. Zebra uses a software RAID on commodity hardware (workstation, disks, and networks) to address RAID's cost disadvantage, and LFS' batched writes provide efficient access to a network RAID. Furthermore, the reliability of both LFS and RAID makes it feasible to distribute data storage across a network.

LFS' solution to the small-write problem is particularly important for Zebra's network striping since reading old data to recalculate RAID parity would be a network operation for Zebra. As Figure 1 illustrates, each Zebra client coalesces its writes into a private *per-client log*. It 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 efficiently calculate *parity fragments* entirely as a local operation and then store them on an additional storage server to provide high data availability.

Zebra's log-structured architecture significantly simplifies its failure recovery. Like LFS, Zebra provides efficient recovery using checkpoint and roll forward. To roll the log forward, Zebra relies on *deltas* stored in the log. 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, to allow the system to replay the modification during recovery. Deltas greatly simplify

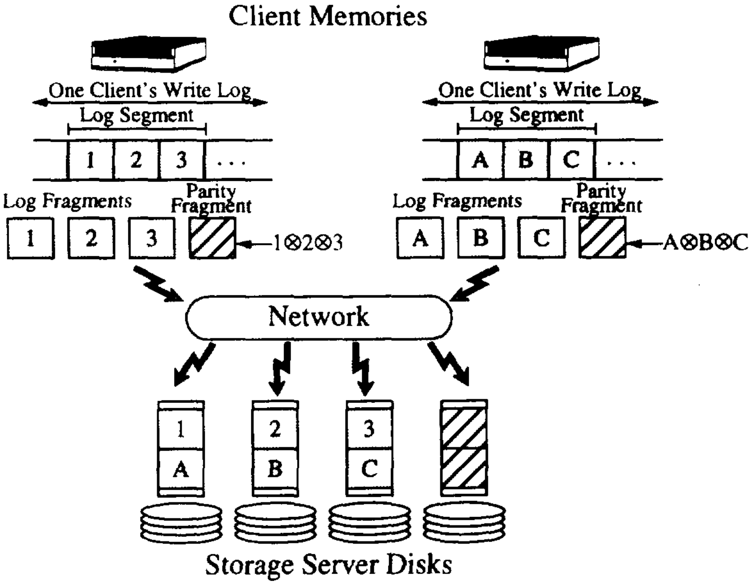


Fig. 1. Log-based striping used by Zebra and xFS. Each client writes its new file data into a private append-only log and stripes this log across the storage servers. Clients compute parity for segments, not for individual files.

recovery by providing an atomic commit for actions that modify state located on multiple machines: each delta encapsulates a set of changes to file system state that must occur as a unit.

Although Zebra points the way toward serverlessness, several factors limit Zebra's scalability. First, a single *file manager* tracks where clients store data blocks in the log; the manager also handles cache consistency operations. Second, Zebra, like LFS, relies on a single cleaner to create empty segments. Finally, Zebra stripes each segment to all of the system's storage servers, limiting the maximum number of storage servers that Zebra can use efficiently.

2.4 Multiprocessor Cache Consistency

Network file systems resemble multiprocessors in that both provide a uniform view of storage across the system, requiring both to track where blocks are cached. This information allows them to maintain cache consistency by invalidating stale cached copies. Multiprocessors such as DASH [Lenoski et al. 1990] and Alewife [Chaiken et al. 1991] scalably distribute this task by dividing the system's physical memory evenly among processors; each processor manages the cache consistency state for its own physical memory locations.[1]

[1] In the context of scalable multiprocessor consistency, this state is referred to as a *directory*. We avoid this terminology to prevent confusion with file system directories that provide a hierarchical organization of file names.

Unfortunately, the fixed mapping from physical memory addresses to consistency managers makes this approach unsuitable for file systems. Our goal is graceful recovery and load rebalancing whenever the number of machines in xFS changes; such reconfiguration occurs when a machine crashes or when a new machine joins xFS. Furthermore, as we show in Section 3.2.4, by directly controlling which machines manage which data, we can improve locality and reduce network communication.

2.5 Previous xFS Work

The design of xFS has evolved considerably since our original proposal [Dahlin et al. 1994a; Wang and Anderson 1993]. The original design stored all system data in client disk caches and managed cache consistency using a hierarchy of metadata servers rooted at a central server. Our new implementation eliminates client disk caching in favor of network striping to take advantage of high-speed, switched LANs. We still believe that the aggressive caching of the earlier design would work well under different technology assumptions; in particular, its efficient use of the network makes it well suited for both wireless and wide-area network use. Moreover, our new design eliminates the central management server in favor of a distributed metadata manager to provide better scalability, locality, and availability.

We have also previously examined cooperative caching—using client memory as a global file cache—via simulation [Dahlin et al. 1994b] and therefore focus only on the issues raised by integrating cooperative caching with the rest of the serverless system.

3. SERVERLESS FILE SERVICE

The RAID, LFS, Zebra, and multiprocessor cache consistency work discussed in the previous section leaves three basic problems unsolved. First, we need scalable, distributed metadata and cache consistency management, along with enough flexibility to reconfigure responsibilities dynamically after failures. Second, the system must provide a scalable way to subset storage servers into groups to provide efficient storage. Finally, a log-based system must provide scalable log cleaning.

This section describes the xFS design as it relates to the first two problems. Section 3.1 provides an overview of how xFS distributes its key data structures. Section 3.2 then provides examples of how the system as a whole functions for several important operations. This entire section disregards several important details necessary to make the design practical; in particular, we defer discussion of log cleaning, recovery from failures, and security until Sections 4 through 6.

3.1 Metadata and Data Distribution

The xFS design philosophy can be summed up with the phrase, "anything, anywhere." All data, metadata, and control can be located anywhere in the system and can be dynamically migrated during operation. We 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. Furthermore, we 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.

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 indicates 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.[2]

The xFS system performs the same tasks, but it builds on the ideas discussed in Section 2 to distribute this work over all of the machines in the system. To provide scalable control of disk metadata and cache consistency state, xFS splits management among *metadata managers* similar to multiprocessor consistency managers. Unlike multiprocessor managers, xFS managers can alter the mapping from files to managers. Similarly, to provide scalable disk storage, xFS uses log-based network striping inspired by Zebra, but it dynamically clusters disks into *stripe groups* to allow the system to scale to large numbers of storage servers. Finally, xFS replaces the server cache with *cooperative caching* that forwards data among client caches under the control of the managers. In xFS, four types of entities—the clients, storage servers, and managers already mentioned, and the *cleaners* discussed in Section 4—cooperate to provide file service as Figure 2 illustrates.

The key challenge for xFS is locating data and metadata in this dynamically changing, completely distributed system. The rest of this subsection examines four key maps used for this purpose: the *manager map*, the *imap*, *file directories*, and the *stripe group map*. The manager map allows clients to determine which manager to contact for a file, and the imap allows each manager to locate where its files are stored in the on-disk log. File directories serve the same purpose in xFS as in a standard UNIX file system, providing a mapping from a human-readable name to a metadata locator called an index number. Finally, the stripe group map provides mappings from segment identifiers embedded in disk log addresses to the set of physical machines storing the segments. The rest of this subsection discusses these four data structures before giving an example of their use in file reads and writes. For reference, Table I provides a summary of these and other key xFS

[2] Note that the NFS server does not keep caches consistent. Instead NFS relies on clients to verify that a block is current before using it. We rejected that approach because it sometimes allows clients to observe stale data when a client tries to read what another client recently wrote.

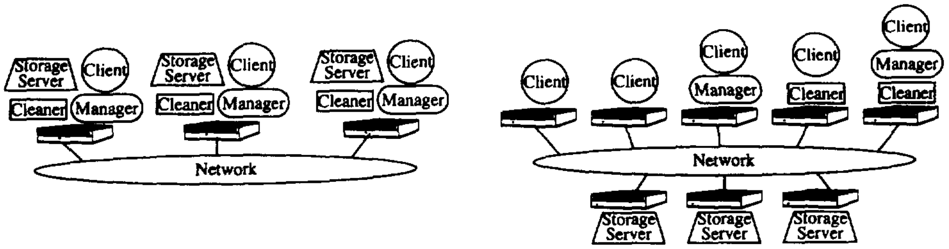


Fig. 2. Two simple xFS installations. In the first, each machine acts as a client, storage server, cleaner, and manager, while in the second each node only performs some of those roles. The freedom to configure the system is not complete; managers and cleaners access storage using the client interface, so all machines acting as managers or cleaners must also be clients.

data structures. Figure 3 in Section 3.2.1. illustrates how these components work together.

3.1.1 *The Manager Map.* xFS distributes management responsibilities according to a globally replicated manager map. A client uses this mapping to locate a file's manager from the file's index number by extracting some of the index number's bits and using them as an index into the manager map. The map itself is simply a table that indicates which physical machines manage which groups of index numbers at any given time.

This indirection allows xFS to adapt when managers enter or leave the system; the map can also act as a coarse-grained load-balancing mechanism to split the work of overloaded managers. Where distributed multiprocessor cache consistency relies on a fixed mapping from physical addresses to managers, xFS can change the mapping from index number to manager by changing the manager map.

To support reconfiguration, the manager map should have at least an order of magnitude more entries than there are managers. This rule of thumb allows the system to balance load by assigning roughly equal portions of the map to each manager. When a new machine joins the system, xFS can modify the manager map to assign some of the index number space to the new manager by having the original managers send the corresponding parts of their manager state to the new manager. Section 5 describes how the system reconfigures manager maps. Note that the prototype has not yet implemented this dynamic reconfiguration of manager maps.

xFS globally replicates the manager map to all of the managers and all of the clients in the system. This replication allows managers to know their responsibilities, and it allows clients to contact the correct manager directly —with the same number of network hops as a system with a centralized manager. We feel it is reasonable to distribute the manager map globally because it is relatively small (even with hundreds of machines, the map would be only tens of kilobytes in size) and because it changes only to correct a load imbalance or when a machine enters or leaves the system.

Table I. Summary of Key xFS Data Structures

Data Structure	Purpose	Location	Section
Manager Map	Maps file's index number → manager.	Globally replicated.	3.1.1
Imap	Maps file's index number → disk log address of file's index node.	Split among managers.	3.1.2
Index Node	Maps file offset → disk log address of data block.	In on-disk log at storage servers.	3.1.2
Index Number	Key used to locate metadata for a file.	File directory.	3.1.3
File Directory	Maps file's name → file's index number.	In on-disk log at storage servers.	3.1.3
Disk Log Address	Key used to locate blocks on storage server disks. Includes a stripe group identifier, segment ID, and offset within segment.	Index nodes and the imap.	3.1.4
Stripe Group Map	Maps disk log address → list of storage servers.	Globally replicated.	3.1.4
Cache Consistency State	Lists clients caching or holding the write token of each block.	Split among managers.	3.2.1 3.2.3
Segment Utilization State	Utilization, modification time of segments.	Split among clients.	4
S-Files	On-disk cleaner state for cleaner communication and recovery.	In on-disk log at storage servers.	4
I-File	On disk copy of imap used for recovery.	In on-disk log at storage servers.	5
Deltas	Log modifications for recovery roll-forward.	In on-disk log at storage servers.	5
Manager Checkpoints	Record manager state for recovery.	In on-disk log at storage servers.	5

This table summarizes the purpose of the key xFS data structures. The location column indicates where these structures are located in xFS, and the Section column indicates where in this article the structure is described.

The manager of a file controls two sets of information about it: cache consistency state and disk location metadata. Together, these structures allow the manager to locate all copies of the file's blocks. The manager can thus forward client read requests to where the block is stored, and it can invalidate stale data when clients write a block. For each block, the cache consistency state lists the clients caching the block or the client that has write ownership of it. The next subsection describes the disk metadata.

3.1.2 *The Imap.* Managers track not only where file blocks are cached but also where in the on-disk log they are stored. xFS uses the LFS imap to encapsulate disk location metadata; each file's index number has an entry in the imap that points to that file's disk metadata in the log. To make LFS' imap scale, xFS distributes the imap among managers according to the manager map so that managers handle the imap entries and cache consistency state of the same files.

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 addresses of indirect blocks that contain more data block addresses, double indirect blocks that contain addresses of indirect blocks, and so on.

3.1.3 *File Directories and Index Numbers.* xFS uses the data structures described above to locate a file's manager given the file's index number. To determine the file's index number, xFS, like FFS and LFS, uses file directories that contain mappings from file names to index numbers. xFS stores directories in regular files, allowing a client to learn an index number by reading a directory.

In xFS, the index number listed in a directory determines a file's manager. When a file is created, we choose its index number so that the file's manager is on the same machine as the client that creates the file. Section 3.2.4 describes simulation results of the effectiveness of this policy in reducing network communication.

3.1.4 *The Stripe Group Map.* Like Zebra, xFS bases its storage subsystem on simple storage servers to which clients write log fragments. To improve performance and availability when using large numbers of storage servers, rather than stripe each segment over all storage servers in the system, xFS implements stripe groups as have been proposed for large RAIDs [Chen et al. 1994]. Each stripe group includes a separate subset of the system's storage servers, and clients write each segment across a stripe group rather than across all of the system's storage servers. xFS uses a globally replicated stripe group map to direct reads and writes to the appropriate storage servers as the system configuration changes. Like the manager map, xFS globally replicates the stripe group map because it is small and seldom changes. The current version of the prototype implements reads and writes from multiple stripe groups, but it does not dynamically modify the group map.

Stripe groups are essential to support large numbers of storage servers for at least four reasons. First, without stripe groups, clients would stripe each of their segments over all of the disks in the system. This organization would require clients to send small, inefficient fragments to each of the many storage servers or to buffer enormous amounts of data per segment so that they could write large fragments to each storage server. Second, stripe groups match the aggregate bandwidth of the groups' disks to the network bandwidth of a client, using both resources efficiently; while one client writes at its full network bandwidth to one stripe group, another client can do the same with a different group. Third, by limiting segment size, stripe groups make cleaning more efficient. This efficiency arises because when cleaners extract segments' live data, they can skip completely empty segments, but they must read partially full segments in their entirety; large segments linger in the partially full state longer than small segments, significantly increasing cleaning costs. Finally, stripe groups greatly improve availability. Because each group stores its own parity, the system can survive multiple server failures if they happen to strike different groups; in a large system with random failures this is the most likely case. The cost for this improved availability is a marginal reduction in disk storage and effective bandwidth because the system dedicates one parity server per group rather than one for the entire system.

The stripe group map provides several pieces of information about each group: the group's ID, the members of the group, and whether the group is *current* or *obsolete*; we describe the distinction between current and obsolete groups below. When a client writes a segment to a group, it includes the stripe group's ID in the segment's identifier and uses the map's list of storage servers to send the data to the correct machines. Later, when it or another client wants to read that segment, it uses the identifier and the stripe group map to locate the storage servers to contact for the data or parity.

xFS distinguishes between current and obsolete groups to support reconfiguration. When a storage server enters or leaves the system, xFS changes the map so that each active storage server belongs to exactly one current stripe group. If this reconfiguration changes the membership of a particular group, xFS does not delete the group's old map entry. Instead, it marks that entry as "obsolete." Clients write only to current stripe groups, but they may read from either current or obsolete stripe groups. By leaving the obsolete entries in the map, xFS allows clients to read data previously written to the groups without first transferring the data from obsolete groups to current groups. Over time, the cleaner will move data from obsolete groups to current groups [Hartman and Ousterhout 1995]. When the cleaner removes the last block of live data from an obsolete group, xFS deletes its entry from the stripe group map.

3.2 System Operation

This section describes how xFS uses the various maps we described in the previous section. We first describe how reads, writes, and cache consistency work and then present simulation results examining the issue of locality in the assignment of files to managers.

3.2.1 *Reads and Caching.* Figure 3 illustrates how xFS 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 with fast LANs. On today's fast LANs, 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, xFS breaks this recursion at the root; the kernel learns the index number of the root when it mounts the file system.

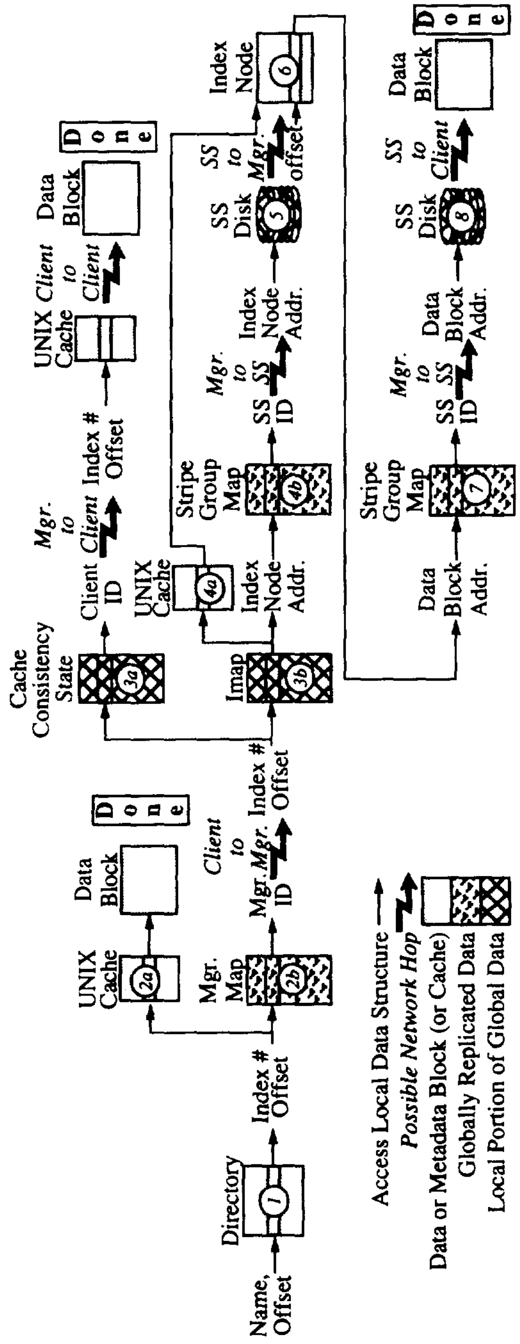
As the top left path in the figure indicates, the client first checks its local UNIX block cache for the block (*2a*); if the block is present, the request is done. Otherwise it follows the lower path to fetch the data over the network. xFS first uses the manager map to locate the correct manager for the index number (*2b*) and then sends the request to the manager. If the manager is not colocated 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. (We 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.

One important design decision was to cache index nodes at managers but not at clients. Although caching index nodes at clients would allow them to read many blocks from storage servers without sending a request through the


Name, Offset
Directory
1
Index # Offset
UNIX Cache
2a
Data Block
Done
Mgr. Map
2b
Mgr. ID
Client to Mgr.
Index # Offset
Cache Consistency State
3a
Client ID
Mgr. to Client
Index # Offset
UNIX Cache
Client to Client
Data Block
Done
Imap
3b
UNIX Cache
4a
Index Node Addr.
Stripe Group Map
4b
SS ID
Mgr. to SS
Index Node Addr.
SS Disk
5
SS to Mgr.
offset
Index Node
6
Data Block Addr.
Stripe Group Map
7
SS ID
Mgr. to SS
Data Block Addr.
SS Disk
8
SS to Client
Data Block
Done
Access Local Data Structure
Possible Network Hop
Data or Metadata Block (or Cache)
Globally Replicated Data
Local Portion of Global Data

manager for each block, doing so has three 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 now cache the block so that the manager knows to invalidate the block if it is modified. Finally, our approach simplifies the design by eliminating client caching and cache consistency for index nodes—only the manager handling an index number directly accesses its index node.

3.2.2 *Writes.* Clients buffer writes in their local memory until committed to a stripe group of storage servers. Because xFS uses a log-based file system, 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 index nodes and imaps and periodically log these changes to stable storage. As with Zebra, xFS does not need to commit both index nodes and their data blocks "simultaneously" because the client's log includes *deltas* that allows reconstruction of the manager's data structures in the event of a client or manager crash. We discuss deltas in more detail in Section 5.1.

As in BSD LFS [Seltzer et al. 1993], each manager caches its portion of the imap in memory and stores it on 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, the system locates the blocks of the ifile using manager checkpoints described in Section 5.1.

3.2.3 *Cache Consistency.* xFS utilizes a token-based cache consistency scheme similar to Sprite [Nelson et al. 1988] and Andrew [Howard et al. 1988] except that xFS manages consistency on a per-block rather than per-file basis. Before a client modifies a block, it must acquire write ownership of that block. 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 any changes to stable storage, and forward the data to the new client.

xFS 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.

3.2.4 *Management Distribution Policies.* xFS tries to assign files used by a client to a manager colocated on that machine. This section presents a simulation study that examines policies for assigning files to managers. We show that colocating a file's management with the client that creates that file can significantly improve locality, reducing the number of network hops needed to satisfy client requests by over 40% compared to a centralized manager.

The xFS prototype uses a policy we call First Writer. When a client creates a file, xFS chooses an index number that assigns the file's management to the manager colocated with that client. For comparison, we also simulated a Centralized policy that uses a single, centralized manager that is not colocated with any of the clients.

We examined management policies by simulating xFS' behavior under a seven-day trace of 236 clients' NFS accesses to an Auspex file server in the Berkeley Computer Science Division [Dahlin et al. 1994a]. We warmed the simulated caches through the first day of the trace and gathered statistics through the rest. Since we would expect other workloads to yield different results, evaluating a wider range of workloads remains important work.

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

Two artifacts of the trace affect the simulation. First, 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. Because we simulate larger caches than those of the traced system, this factor does not alter the total number of network requests for each policy [Smith 1977], which is the relative metric we use for comparing policies.

The second limitation of the trace is that its finite length does not allow us 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, we 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, we 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—we believe this heuristic will yield results close to a true "First Writer" policy for writes, although it will give pessimistic locality results for "cold-start" read misses that it assigns to random managers.

Figure 4 shows the impact of the policies on locality. The First Writer policy reduces the total number of network hops needed to satisfy client requests by 43%. 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.

Figure 5 illustrates the average number of network messages to satisfy a read block request, write block request, or delete file request. The communication for a read block request includes all of the network hops indicated in Figure 3. Despite the large number of network hops that can be incurred by some requests, the average per request is quite low. Seventy-five percent of

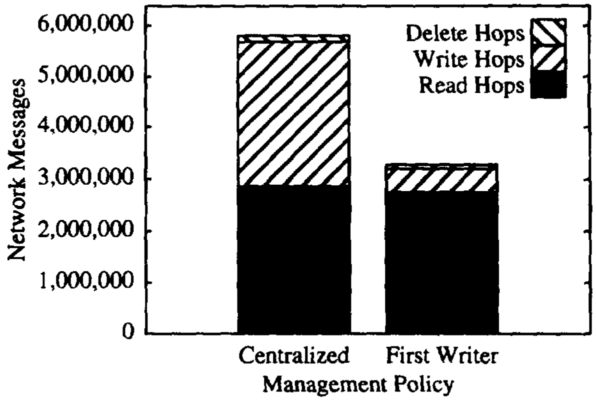


Fig. 4. Comparison of locality as measured by network traffic for the Centralized and First Writer management policies.

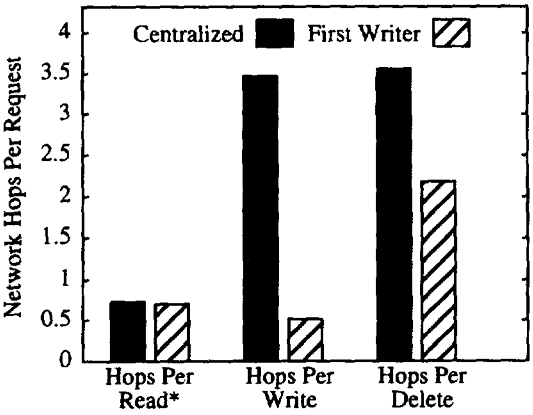


Fig. 5. Average number of network messages needed to satisfy a read block, write block, or delete file request under the Centralized and First Writer policies. The Hops Per Write column does not include a charge for writing the segment containing block writes to disk because the segment write is asynchronous to the block write request and because the large segment amortizes the per-block write cost. *Note that the number of hops per read would be even lower if the trace included all local hits in the traced system.

read requests in the trace were satisfied by the local cache; as noted earlier, the local hit rate would be even higher if the trace included local hits in the traced system. An average local read miss costs 2.9 hops under the First Writer policy; 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 12% of the time it can avoid one hop because the manager is colocated with the client making the request or the client supplying the data. Under both the Centralized and First Writer policies, a read miss will occasionally incur a few additional hops to read an index node or indirect block from a storage server.

Writes benefit more dramatically from locality. Of the 55% of write requests that required the client to contact the manager to establish write ownership, the manager was colocated with the client 90% of the time. When a manager had to invalidate stale cached data, the cache being invalidated was local one-third of the time. Finally, when clients flushed data to disk, they informed the manager of the data's new storage location, a local operation 90% of the time.

Deletes, though rare, also benefit from locality: 68% of file delete requests went to a local manager, and 89% of the clients notified to stop caching deleted files were local to the manager.

In the future, we plan to examine other policies for assigning managers. For instance, we plan to investigate modifying directories to permit xFS to dynamically change a file's index number and thus its manager after it has been created. This capability would allow fine-grained load balancing on a per-file rather than a per-manager map entry basis, and it would permit xFS to improve locality by switching managers when a different machine repeatedly accesses a file.

Another optimization that we plan to investigate is to assign multiple managers to different portions of the same file to balance load and provide locality for parallel workloads.

4. CLEANING

When a log-structured file system such as xFS writes data by appending complete segments to its logs, it often invalidates blocks in old segments, leaving "holes" that contain no data. LFS uses a *log cleaner* to coalesce live data from old segments into a smaller number of new segments, creating completely empty segments that can be used for future full segment writes. Since the cleaner must create empty segments at least as quickly as the system writes new segments, a single, sequential cleaner would be a bottleneck in a distributed system such as xFS. Our design therefore provides a distributed cleaner.

An LFS cleaner, whether centralized or distributed, has three main tasks. First, the system must keep *utilization status* about old segments—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 must examine this bookkeeping information to select segments to clean. Third, the cleaner reads the live blocks from old log segments and writes those blocks to new segments.

The rest of this section describes how xFS distributes cleaning. We first describe how xFS tracks segment utilizations, then how we identify subsets of segments to examine and clean, and finally, how we coordinate the parallel cleaners to keep the file system consistent. Because the prototype does not yet implement the distributed cleaner, this section includes the key simulation results motivating our design.

4.1 Distributing Utilization Status

xFS assigns the burden of maintaining each segment's utilization status to the client that wrote the segment. This approach provides parallelism by distributing the bookkeeping, and it provides good locality; because clients seldom write-share data [Baker et al. 1991; Blaze 1993; Kistler and Satyanarayanan 1992] a client's writes usually affect only local segments' utilization status.

We simulated this policy to examine how well it reduced the overhead of maintaining utilization information. For input to the simulator, we used the Auspex trace described in Section 3.2.4, but since caching is not an issue, we gather statistics for the full seven-day trace (rather than using some of that time to warm caches).

Figure 6 shows the results of the simulation. The bars summarize the network communication necessary to monitor segment 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 clients do not have to send messages when they modify blocks that are still in their local write buffers. The results for this policy are optimistic because the simulator assumes that blocks survive in clients' write buffers for 30 seconds or until overwritten, whichever is sooner; this assumption allows the simulated system to avoid communication more often than a real system since it does not account for segments that are written to disk early due to syncs [Baker et al. 1992]. (Unfortunately, syncs are not visible in our Auspex traces.) Finally, under the Distributed policy, each client tracks the status of blocks that it writes so that it needs no network messages when modifying a block for which it was the last writer.

During the seven days of the trace, of the one million blocks written by clients and then later overwritten or deleted, 33% were modified within 30 seconds by the same client and therefore required no network communication under the Centralized Optimistic policy. However, the Distributed scheme does much better, reducing communication by a factor of 18 for this workload compared to even the Centralized Optimistic policy.

4.2 Distributing Cleaning

Clients store their segment utilization information in *s-files*. We implement s-files as normal xFS files to facilitate recovery and sharing of s-files by different machines in the system.

Each s-file contains segment 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.

A leader in each stripe group initiates cleaning when the number of free segments in that group falls below a low-water mark or when the group is

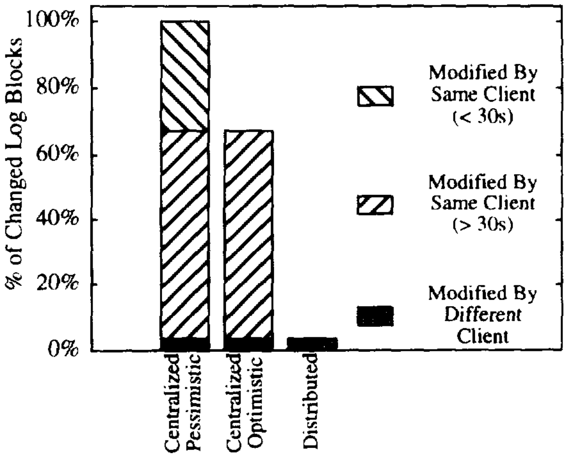


Fig. 6. Simulated network communication between clients and cleaner. Each bar shows the fraction of all blocks modified or deleted in the trace, based on the time and client that modified the block. Blocks can be modified by a different client than originally wrote the data, by the same client within 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.

idle. The group leader decides which cleaners should clean the stripe group's segments. It sends each of those cleaners part of the list of s-files that contain utilization information for the group. By giving each cleaner a different subset of the s-files, xFS specifies subsets of segments that can be cleaned in parallel.

A simple policy would be to assign each client to clean its own segments. An attractive alternative is to assign cleaning responsibilities to idle machines. xFS would do this by assigning s-files from active machines to the cleaners running on idle ones.

4.3 Coordinating Cleaners

Like BSD LFS and Zebra, xFS 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 the blocks' old segments to their new segments, 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 update takes precedence over the cleaner's update. Although our algorithm for distributing 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 resolving conflicts between cleaners.

5. RECOVERY AND RECONFIGURATION

Availability is a key challenge for a distributed system such as xFS. Because xFS distributes the file system across many machines, it must be able to continue operation when some of the machines fail. To meet this challenge, xFS builds on Zebra's recovery mechanisms, the keystone of which is redundant, log-structured storage. The redundancy provided by a software RAID makes the system's logs highly available, and the log-structured storage allows the system to quickly recover a consistent view of its data and metadata through LFS checkpoint recovery and roll-forward.

LFS provides fast recovery by scanning the end of its log to first read a checkpoint and then roll forward, and Zebra demonstrates how to extend LFS recovery to a distributed system in which multiple clients are logging data concurrently. xFS addresses two additional problems. First, xFS regenerates the manager map and stripe group map using a distributed consensus algorithm. Second, xFS recovers manager metadata from multiple managers' logs, a process that xFS makes scalable by distributing checkpoint writes and recovery to the managers and by distributing roll-forward to the clients.

The prototype implements only a limited subset of xFS' recovery functionality—storage servers recover their local state after a crash; they automatically reconstruct data from parity when one storage server in a group fails; and clients write deltas into their logs to support manager recovery. However, we have not implemented manager checkpoint writes, checkpoint recovery reads, or delta reads for roll-forward. The current prototype also fails to recover cleaner state and cache consistency state, and it does not yet implement the consensus algorithm needed to dynamically reconfigure manager maps and stripe group maps. This section outlines our recovery design and explains why we expect it to provide scalable recovery for the system. However, given the complexity of the recovery problem and the early state of our implementation, continued research will be needed to fully understand scalable recovery.

5.1 Data Structure Recovery

Table II lists the data structures that storage servers, managers, and cleaners recover after a crash. For a systemwide reboot or widespread crash, recovery proceeds from storage servers, to managers, and then to cleaners because later levels depend on earlier ones. Because recovery depends on the logs stored on the storage servers, xFS will be unable to continue if multiple storage servers from a single stripe group are unreachable due to machine or network failures. We plan to investigate using multiple parity fragments to allow recovery when there are multiple failures within a stripe group [Blaum et al. 1994]. Less widespread changes to xFS membership—such as when an authorized machine asks to join the system, when a machine notifies the system that it is withdrawing, or when a machine cannot be contacted because of a crash or network failure—trigger similar reconfiguration steps. For instance, if a single manager crashes, the system skips the steps to recover the storage servers, going directly to generate a new manager map

Table II. Data Structures Restored During Recovery

	Data Structure	Recovered From
Storage Server	Log Segments	Local Data Structures
	Stripe Group Map	Consensus
Manager	Manager Map	Consensus
	Disk Location Metadata	Checkpoint and Roll-Forward
	Cache Consistency Metadata	Poll Clients
Cleaner	Segment Utilization	S-Files

Recovery occurs in the order listed from top to bottom because lower data structures depend on higher ones.

that assigns the failed manager's duties to a new manager; the new manager then recovers the failed manager's disk metadata from the storage server logs using checkpoint and roll-forward, and it recovers its cache consistency state by polling clients.

5.1.1 *Storage Server Recovery.* The segments stored on storage server disks contain the logs needed to recover the rest of xFS' data structures, so the storage servers initiate recovery by restoring their internal data structures. When a storage server recovers, it regenerates its mapping of xFS fragment IDs to the fragments' physical disk addresses, rebuilds its map of its local free disk space, and verifies checksums for fragments that it stored near the time of the crash. Each storage server recovers this information independently from a private checkpoint, so this stage can proceed in parallel across all storage servers.

Storage servers next regenerate their stripe group map. First, the storage servers use a distributed consensus algorithm [Cristian 1991; Ricciardi and Birman 1991; Schroeder et al. 1991] to determine group membership and to elect a group leader. Each storage server then sends the leader a list of stripe groups for which it stores segments, and the leader combines these lists to form a list of groups where fragments are already stored (the obsolete stripe groups). The leader then assigns each active storage server to a current stripe group and distributes the resulting stripe group map to the storage servers.

5.1.2 *Manager Recovery.* Once the storage servers have recovered, the managers can recover their manager map, disk location metadata, and cache consistency metadata. Manager map recovery uses a consensus algorithm as described above for stripe group recovery. Cache consistency recovery relies on server-driven polling [Baker 1994; Nelson et al. 1988]: a recovering manager contacts the clients, and each client returns a list of the blocks that it is caching or for which it has write ownership from that manager's portion of the index number space.

The remainder of this subsection describes how managers and clients work together to recover the managers' disk location metadata—the distributed imap and index nodes that provide pointers to the data blocks on disk. Like

LFS and Zebra, xFS recovers this data using a checkpoint and roll-forward mechanism. xFS distributes this disk metadata recovery to managers and clients so that each manager and client log written before the crash is assigned to one manager or client to read during recovery. Each manager reads the log containing the checkpoint for its portion of the index number space, and where possible, clients read the same logs that they wrote before the crash. This delegation occurs as part of the consensus process that generates the manager map.

The goal of manager checkpoints is to help managers recover their imaps from the logs. As Section 3.2.2 described, managers store copies of their imaps in files called ifiles. To help recover the imaps from the ifiles, managers periodically write checkpoints that contain lists of pointers to the disk storage locations of the ifiles' blocks. Because each checkpoint corresponds to the state of the ifile when the checkpoint is written, it also includes the positions of the clients' logs reflected by the checkpoint. Thus, once a manager reads a checkpoint during recovery, it knows the storage locations of the blocks of the ifile as they existed at the time of the checkpoint, and it knows where in the client logs to start reading to learn about more recent modifications. The main difference between xFS and Zebra or BSD LFS is that xFS has multiple managers, so each xFS manager writes its own checkpoint for its part of the index number space.

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 backward 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.

To account for changes that had not reached the managers' checkpoints, the system uses roll-forward, where clients use the deltas stored in their logs to replay actions that occurred later than the checkpoints. To initiate roll-forward, the managers use the log position information from their checkpoints to advise the clients of the earliest segments to scan. Each client locates the tail of its log by querying storage servers, and then it reads the log backward to locate the earliest segment needed by any manager. Each client then reads forward in its log, using the manager map to send the deltas to the appropriate managers. Managers use the deltas to update their imaps and index nodes as they do during normal operation; version numbers in the deltas allow managers to chronologically order different clients' modifications to the same files [Hartman and Ousterhout 1995].

5.1.3 *Cleaner Recovery.* Clients checkpoint the segment utilization information needed by cleaners in standard xFS files, called s-files. Because these checkpoints are stored in standard files, they are automatically recovered by the storage server and manager phases of recovery. 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-file checkpoint. To avoid scanning their logs twice, clients gather this segment utilization summary information during the roll-forward phase for manager metadata.

5.2 Scalability of Recovery

Even with the parallelism provided by xFS' approach to manager recovery, future work will be needed to evaluate its scalability. Our design is based on the observation that, while the procedures described above can require $O(N^2)$ communications steps (where N refers to the number of clients, managers, or storage servers), each phase can proceed in parallel across N machines.

For instance, to locate the tails of the systems logs, each manager and client queries each storage server group to locate the end of its log. While this can require a total of $O(N^2)$ messages, each manager or client only needs to contact N storage server groups, and all of the managers and clients can proceed in parallel, provided that they take steps to avoid having many machines simultaneously contact the same storage server [Baker 1994]; we plan to use randomization to accomplish this goal. Similar considerations apply to the phases where managers read their checkpoints, clients roll forward, and managers query clients for their cache consistency state.

6. SECURITY

xFS, as described, is appropriate for a restricted environment—among machines that communicate over a fast network and that trust one another's kernels to enforce security. xFS managers, storage servers, clients, and cleaners must run on secure machines using the protocols we have described so far. However, xFS can support less trusted clients using different protocols that require no more trust than traditional client protocols, albeit at some cost to performance. Our current implementation allows unmodified UNIX clients to mount a remote xFS partition using the standard NFS protocol.

Like other file systems, xFS trusts the kernel to enforce a firewall between untrusted user processes and kernel subsystems such as xFS. The xFS storage servers, managers, and clients can then enforce standard file system security semantics. For instance, xFS storage servers only store fragments supplied by authorized clients; xFS managers only grant read and write tokens to authorized clients; xFS clients only allow user processes with appropriate credentials and permissions to access file system data.

We expect this level of trust to exist within many settings. For instance, xFS could be used within a group or department's administrative domain, where all machines are administered the same way and therefore trust one another. Similarly, xFS would be appropriate within a NOW where users already trust remote nodes to run migrated processes on their behalf. Even in

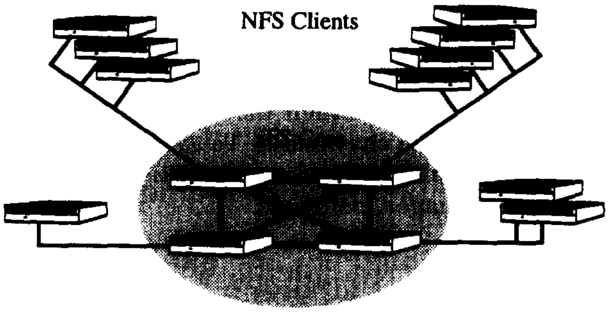


Fig. 7. An xFS core acting as a scalable file server for unmodified NFS clients.

environments that do not trust all desktop machines, xFS could still be used within a trusted *core* of desktop machines and servers, among physically secure compute servers and file servers in a machine room, or within one of the parallel server architectures now being researched [Kubiatowicz and Agarwal 1993; Kuskin et al. 1994]. In these cases, the xFS core could still provide scalable, reliable, and cost-effective file service to less trusted *fringe* clients running more restrictive protocols. The downside is that the core system cannot exploit the untrusted CPUs, memories, and disks located in the fringe.

Client trust is a concern for xFS because xFS ties its clients more intimately to the rest of the system than do traditional protocols. This close association improves performance, but it may increase the opportunity for mischievous clients to interfere with the system. In either xFS or a traditional system, a compromised client can endanger data accessed by a user on that machine. However, a damaged xFS client can do wider harm by writing bad logs or by supplying incorrect data via cooperative caching. In the future we plan to examine techniques to guard against unauthorized log entries and to use encryption-based techniques to safeguard cooperative caching.

Our current prototype allows unmodified UNIX fringe clients to access xFS core machines using the NFS protocol as Figure 7 illustrates. To do this, any xFS client in the core exports the xFS file system via NFS, and an NFS client employs the same procedures it would use to mount a standard NFS partition from the xFS client. The xFS core client then acts as an NFS server for the NFS client, providing high performance by employing the remaining xFS core machines to satisfy any requests not satisfied by its local cache. Multiple NFS clients can utilize the xFS core as a scalable file server by having different NFS clients mount the xFS file system using different xFS clients to avoid bottlenecks. Because xFS 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 xFS' high availability, since they can mount the file system using any available xFS client. Of course, a key to good NFS server performance is to efficiently implement synchronous writes; our prototype does not yet exploit the nonvolatile RAM optimization found in most commercial NFS servers [Baker et al. 1992], so for best performance, NFS clients should mount these partitions using the "unsafe" option to allow xFS to buffer writes in memory.

7. xFS PROTOTYPE

This section describes the state of the xFS prototype as of August, 1995, and presents preliminary performance results measured on a 32-node cluster of SPARCStation 10's and 20's. Although these results are preliminary and although we expect future tuning to significantly improve absolute performance, they suggest that xFS has achieved its goal of scalability. For instance, in one of our microbenchmarks 32 clients achieved an aggregate large-file write bandwidth of 13.9MB/second, close to a linear speedup compared to a single client's 0.6MB/second bandwidth. Our other tests indicated similar speedups for reads and small-file writes.

The prototype implementation consists of four main pieces. First, we implemented a small amount of code as a loadable module for the Solaris kernel. This code provides xFS' interface to the Solaris v-node layer and accesses the kernel buffer cache. We implemented the remaining three pieces of xFS as daemons outside of the kernel address space to facilitate debugging [Howard et al. 1988]. If the xFS kernel 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' functionality by accessing the manager daemons and the storage server daemons over the network.

The rest of this section summarizes the state of the prototype, describes our test environment, and presents our results.

7.1 Prototype Status

The prototype implements most of xFS' key features, including distributed management, cooperative caching, and network disk striping with single parity and multiple groups. We have not yet completed implementation of a number of other features. The most glaring deficiencies are in crash recovery and cleaning. Although we have implemented storage server recovery, including automatic reconstruction of data from parity, we have not completed implementation of manager state checkpoint and roll-forward; also, we have not implemented 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, we have yet to implement the distributed cleaner. As a result, xFS is still best characterized as a research prototype, and the results in this article should thus be viewed as evidence that the serverless approach is promising, not as "proof" that it will succeed.

7.2 Test Environment

For our testbed, we use a total of 32 machines: eight dual-processor SPARCStation 20's, and 24 single-processor SPARCStation 10's. Each of our machines has 64MB of physical memory. Uniprocessor 50MHz 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 27MB/second and 20MB/second, respectively.

We 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]). We use NFS as our 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' limitations by buying shared-memory multiprocessor servers, we would like to compare xFS running on workstations to NFS running on a large multiprocessor server, but such a machine was not available to us; our NFS server therefore runs on essentially the same platform as the clients. We 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 our NFS and AFS tests, we use one of the SS-20's as the server and the remaining 31 machines as clients. For the xFS tests, all machines act as storage servers, managers, and clients unless otherwise noted. For experiments using fewer than 32 machines, we always include all of the SS-20's before starting to use the less powerful SS-10's.

The xFS storage servers store data on a 256MB partition of a 1.1GB Seagate-ST11200N disk. These disks have an advertised average seek time of 5.4ms and rotate at 5411RPM. We measured a 2.7MB/second peak bandwidth to read from the raw disk device into memory. For all xFS tests, we use a log fragment size of 64KB, and unless otherwise noted we use storage server groups of eight machines—seven for data and one for parity; all xFS tests include the overhead of parity computation. The AFS clients use a 100MB partition of the same disks for local disk caches.

The NFS and AFS central servers use a larger and somewhat faster disk than the xFS storage servers, a 2.1GB DEC RZ 28-VA with a peak bandwidth of 5MB/second 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]. We did not use an NVRAM buffer for the xFS machines, but xFS' log buffer provides similar performance benefits.

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 80MB/second, 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 our xFS measurements, we used a kernel that we compiled from the Solaris 2.4 source release. We measured the TCP throughput to be 3.2MB/second for 8KB packets when using this source release. For the NFS and AFS measurements, we used the binary release of Solaris 2.4, augmented with the binary patches recommended by Sun as of June 1, 1995. This release provides better network performance; our TCP test achieved a throughput of 8.4MB/second for this setup. Alas, we could not get sources for the patches, so our xFS measurements are penalized with a slower effective network than NFS and AFS. RPC overheads further reduce network performance for all three systems.

7.3 Performance Results

This section presents a set of preliminary performance results for xFS under a set of microbenchmarks designed to stress file system scalability and under an application-level benchmark.

These performance results are preliminary. As noted above, several significant pieces of the xFS system—manager checkpoints and cleaning—remain to be implemented. We do not expect these additions to significantly impact the results for the benchmarks presented here. We do not expect checkpoints to ever limit performance. However, thorough future investigation will be needed to evaluate the impact of distributed cleaning under a wide range workloads; other researchers have measured sequential cleaning overheads from a few percent [Blackwell et al. 1995; Rosenblum and Ousterhout 1992] 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 we will attack in the future.

(1) xFS is currently implemented as a set of user-level processes by redirecting vnode layer calls. This 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' address space. To fix this limitation, we are working to move xFS into the kernel. (Note that AFS shares this handicap.)

(2) RPC and TCP/IP overheads severely limit xFS' network performance. We are porting xFS' communications layer to Active Messages [von Eicken et al. 1992] to address this issue.

(3) We have done little profiling and tuning. As we do so, we expect to find and fix inefficiencies.

As a result, the absolute performance is much less than we expect for the well-tuned xFS. As the implementation matures, we 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. Our eventual performance goal is for a single xFS client to achieve read and write bandwidths near that of its maximum network throughput and for multiple clients to realize an aggregate bandwidth approaching the system's aggregate local disk bandwidth.

The microbenchmark results presented here stress the scalability of xFS' storage servers and managers. We examine read and write throughput for large files and write performance for small files, but we do not examine small-file read performance explicitly because the network is too slow to provide an interesting evaluation of cooperative caching; we leave this evaluation as future work. We also use Satyanarayanan's Andrew benchmark

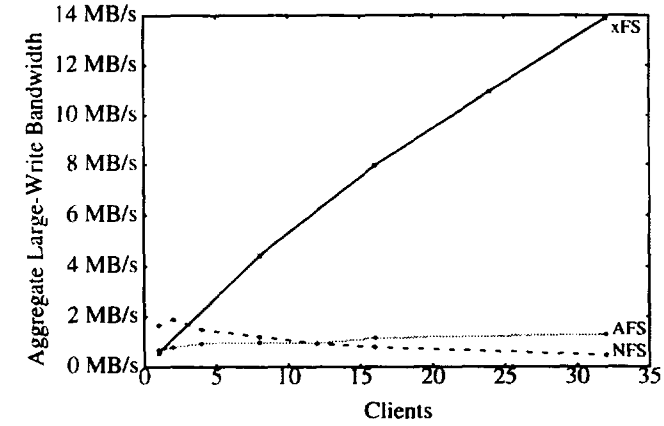


Fig. 8. Aggregate disk write bandwidth. The x-axis indicates the number of clients simultaneously writing private 10MB files, and the y-axis indicates the total throughput across all of the active clients. xFS uses four groups of eight storage servers and 32 managers. NFS' peak throughput is 1.9MB/second with two clients; AFS' is 1.3MB/second with 32 clients; and xFS' is 13.9MB/second with 32 clients.

[Howard et al. 1988] as a simple evaluation of application-level performance. In the future, we plan to compare the systems' performance under more demanding applications.

7.3.1 *Scalability.* Figures 8 through 10 illustrate the scalability of xFS' performance for large writes, large reads, and small writes. For each of these tests, as the number of clients increases, so does xFS' aggregate performance. In contrast, just a few clients saturate NFS' or AFS' single server, limiting peak throughput.

Figure 8 illustrates the performance of our disk write throughput test, in which each client writes a large (10MB), private file and then invokes sync( ) to force the data to disk (some of the data stay in NVRAM in the case of NFS and AFS.) A single xFS client is limited to 0.6MB/second, about one-third of the 1.7MB/second 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 0.7MB/second, limited by AFS' kernel crossings and the overhead of writing data to both the local disk cache and the server disk. As we increase the number of clients, NFS' and AFS' 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 13.9MB/second for 32 clients, and it appears that if we had more clients available for our experiments, they could achieve even more bandwidth from the 32 xFS storage servers and managers.

Figure 9 illustrates the performance of NFS, AFS, and xFS for large reads from disk. For this test, each machine flushes its cache and then sequentially reads a per-client 10MB file. Again, a single NFS or AFS client outperforms a

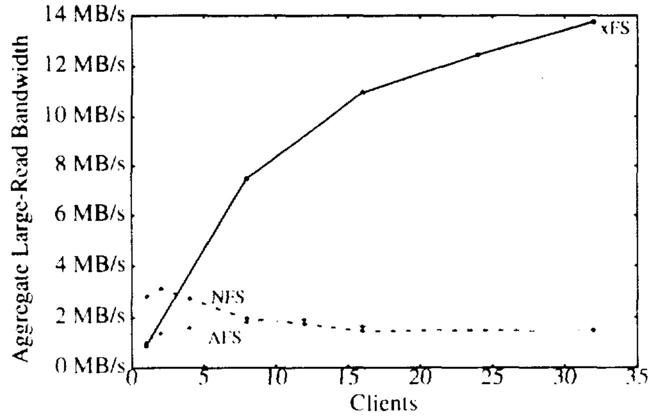


Fig. 9. Aggregate disk read bandwidth. The x-axis indicates the number of clients simultaneously reading private 10MB files and the y-axis the total throughput across all active clients. xFS uses four groups of eight storage servers and 32 managers. NFS' peak throughput is 3.1MB/second with two clients; AFS' is 1.9MB/second with 12 clients; and xFS' is 13.8MB/second with 32 clients.

single xFS client. One NFS client can read at 2.8MB/second, and an AFS client can read at 1.0MB/second, while the current xFS implementation limits one xFS client to 0.9MB/second. As is the case for writes, xFS exhibits good scalability; 32 clients achieve a read throughput of 13.8MB/second. In contrast, two clients saturate NFS at a peak throughput of 3.1MB/second, and 12 clients saturate AFS' central server disk at 1.9MB/second.

While Figure 9 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 with the server. All three systems allow clients to cache data in local memory, providing scalable bandwidths of 20MB/second to 30MB/second 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 40MB/second for such workloads. This aggregate disk bandwidth is significantly larger than xFS' maximum disk bandwidth for two reasons. First, as noted above, xFS is largely untuned, and we expect the gap to shrink in the future. Second, xFS transfers most of the data over the network, while AFS' cache accesses are local. Thus, there will be some workloads for which AFS' disk caches achieve a higher aggregate disk read bandwidth than xFS' network storage. xFS' network striping, however, provides better write performance and will, in the future, provide better read performance for individual clients via striping. Additionally, once we have ported cooperative caching to 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.

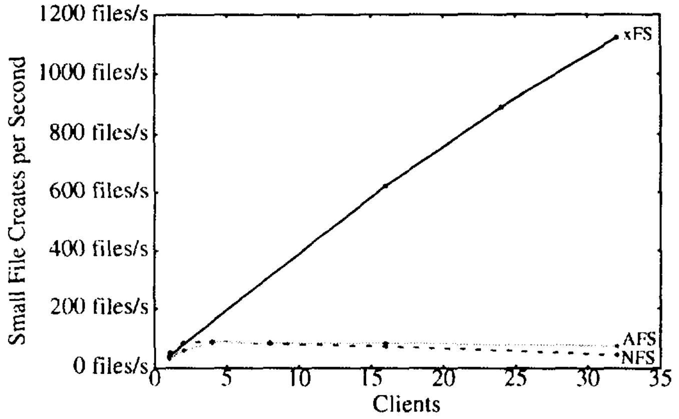


Fig. 10. Aggregate small-write performance. The x-axis indicates the number of clients, each simultaneously creating 2048 1KB files. The y-axis is the average aggregate number of file creates per second during the benchmark run. xFS uses four groups of eight storage servers and 32 managers. NFS achieves 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 1122 files per second with 32 clients.

Figure 10 illustrates the performance when each client creates 2048 files containing 1KB of data per file. For this benchmark, xFS' 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. Thirty-two xFS clients generate a total of 1122 files per second, while NFS' peak rate is 91 files per second with four clients, and AFS' peak is 87 files per second with four clients.

Figure 11 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' scalability, however, allows xFS to outperform the other systems for larger numbers of clients. For instance, with 32 clients xFS takes 117 seconds to complete the benchmark, while increased I/O time, particularly in the copy phase of the benchmark, increases NFS' time to 172 seconds and AFS' 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' on-disk caches.

7.3.2 *Storage Server Scalability.* In the above measurements, we used a 32-node xFS system where all machines acted as clients, managers, and

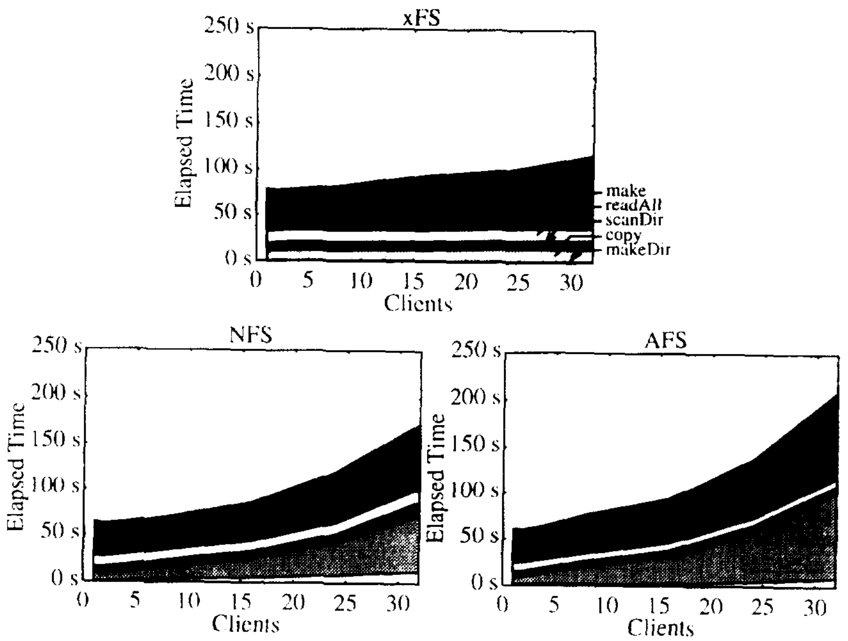


Fig. 11. Average time to complete the Andrew benchmark for NFS, AFS, and xFS as the number of clients simultaneously executing the benchmark varies. The total height of the shaded areas represents the total time to complete the benchmark; each shaded area represents the time for one of the five phases of the benchmark: makeDir, copy, scanDir, readAll, and make. For all of the systems, the caches were flushed before running the benchmark.

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 12 shows the large-write performance as we vary the number of storage servers and as we change the stripe group size.

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

7.3.3 *Manager Scalability.* Figure 13 shows the importance of distributing management among multiple managers to achieve both parallelism and

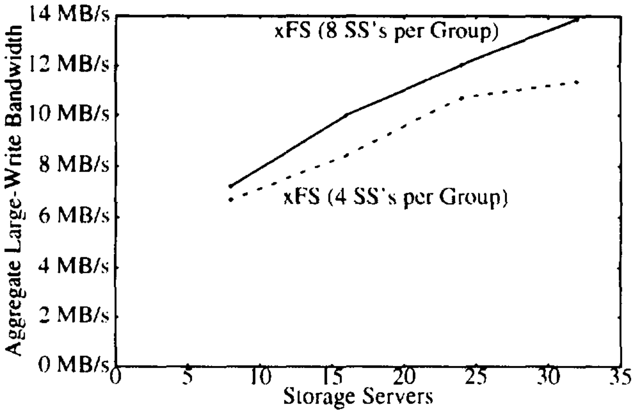


Fig. 12. Large-write throughput as a function of the number of storage servers in the system. The x-axis indicates the total number of storage servers in the system and the y-axis the aggregate bandwidth when 32 clients each write a 10MB file to disk. The solid line indicates performance for stripe groups of eight storage servers (the default), and the dashed line shows performance for groups of four storage servers.

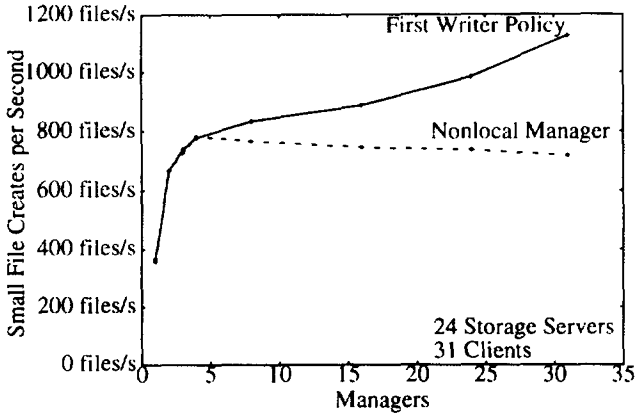


Fig. 13. Small-write performance as a function of the number of managers in the system and manager locality policy. The x-axis indicates the number of managers. The y-axis is the average aggregate number of file creates per second by 31 clients, each simultaneously creating 2048 small (1KB) files. The two lines show the performance using the First Writer policy that colocates 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, we ran this experiment with three groups of eight storage servers and 31 clients. The maximum point on the x-axis is 31 managers.

locality. It varies the number of managers handling metadata for 31 clients running the small-write benchmark.[3] This graph indicates that a single manager is a significant bottleneck for this benchmark. Increasing the sys-

[3] Due to a hardware failure, we ran this experiment with three groups of eight storage servers and 31 clients.

tem 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' First Writer policy. This policy assigns files to managers running on the same machine as the clients that create the files; Section 3.2.4 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 larger fraction of the clients also host managers, the algorithm is able to successfully colocate managers with the clients accessing a file more often.

The Nonlocal Manager line illustrates what would happen without locality. For this line, we 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.

8. RELATED WORK

Section 2 discussed a number of projects that provide an important basis for xFS. This section describes several other efforts to build decentralized file systems and then discusses the dynamic management hierarchies used in some MPPs.

Several file systems, such as CFS [Pierce 1989], Bridge [Dibble and Scott 1989], and Vesta [Corbett et al. 1993], distribute data over multiple storage servers to support parallel workloads; however, they lack mechanisms to provide availability across component failures.

Other parallel systems have implemented redundant data storage intended for restricted workloads consisting entirely of large files, 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 [Lo Verso et al. 1993] provide redundant distributed data storage for parallel environments, and Tiger [Rashid 1994] services multimedia workloads.

TickerTAIP [Cao et al. 1993], SNS [Lee 1995], and AutoRAID [Wilkes et al. 1995] implement RAID-derived storage systems. These systems could provide services similar to xFS' storage servers, but they would require serverless management to provide a scalable and highly available file system interface to augment their simpler disk block interfaces. In contrast with the log-based striping approach taken by Zebra and xFS, TickerTAIP's RAID level 5 [Patterson et al. 1988] architecture makes calculating parity for small writes expensive when disks are distributed over the network. SNS combats this problem by using a RAID level 1 (mirrored) architecture, but this approach approximately doubles the space overhead for storing redundant data. AutoRAID addresses this dilemma by storing data that are actively being written to a RAID level 1 and migrating inactive data to a RAID level 5.

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. While an early xFS study simulated the effect of a hierarchical approach to metadata for file systems [Dahlin et al. 1994a], we now instead support location independence using a manager-map-based approach for three reasons. First, our approach 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.

9. CONCLUSIONS

Serverless file systems distribute file system server responsibilities across large numbers of cooperating machines. This approach eliminates the central server bottleneck inherent in today's file system designs to provide improved performance, scalability, and availability. Furthermore, serverless systems are cost effective because their scalable architecture eliminates the specialized server hardware and convoluted system administration necessary to achieve scalability under current file systems. The xFS prototype demonstrates the viability of building such scalable systems, and its initial performance results illustrate the potential of this approach.

ACKNOWLEDGMENTS

We owe several members of the Berkeley Communications Abstraction Layer group—David Culler, Lok Liu, and Rich Martin—a large debt for helping us to get the 32-node Myrinet network up. We also made use of a modified version of Mendel Rosenblum's LFS cleaner simulator. Eric Anderson, John Hartman, Frans Kaashoek, John Ousterhout, the SOSP program committee, and the anonymous SOSP and TOCS referees provided helpful comments on earlier drafts of this article; their comments greatly improved both the technical content and presentation of this work.

REFERENCES

ANDERSON, T., CULLER, D., PATTERSON, D., AND THE NOW TEAM. 1995. A case for NOW (Networks of Workstations). *IEEE Micro 15*, 1 (Feb.), 54–64.

BAKER, M. 1994. Fast crash recovery in distributed file systems. Ph.D. thesis, Univ. of California at Berkeley, Berkeley, Calif.

BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. 1992. Non-volatile memory for fast, reliable file systems. In *ASPLOS-V* (Sept.). ACM, New York, 10–22.

BAKER, M., HARTMAN, J., KUPFER, M., SHIRRIFF, K., AND OUSTERHOUT, J. 1991. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles* (Oct.). ACM, New York, 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 15th Symposium on Operating Systems Principles* (Dec.). ACM, New York, 40–53.

Birrell, A., Hisgen, A., Jerian, C., Mann, T., and Swart, G. 1993. The Echo distributed file system. Tech. Rep. 111, Digital Equipment Corp., Systems Research Center, Palo Alto., Calif.

Blackwell, T., Harris, J., and Seltzer, M. 1995. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 Winter USENIX*. USENIX Assoc., Berkeley, Calif., 277–288.

Blaum, M., Brady, J., Bruck, J., and Menon, J. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st International Symposium on Computer Architecture* (Apr.). IEEE Computer Society Press, Los Alamitos, Calif., 245–254.

Blaze, M. 1993. Caching in large-scale distributed file systems. Ph.D. thesis, Princeton Univ., Princeton, N.J. Jan.

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 15*, 1 (Feb.), 29–36.

Cabrera, L. and Long, D. 1991. Swift: A storage architecture for large objects. In *Proceedings of the 11th Symposium on Mass Storage Systems* (Oct.). IEEE Computer Society Press, Los Alamitos, Calif., 123–128.

Cao, P., Lim, S., Venkataraman, S., and Wilkes, J. 1993. The TickerTAIP parallel RAID architecture. In *Proceedings of the 20th International Symposium on Computer Architecture* (May). IEEE Computer Society Press, Los Alamitos, Calif., 52–63.

Chaiken, D., Kubiatowicz, J., and Agarwal, A. 1991. LimitLESS directories: A scalable cache coherence scheme. In *ASPLOS-IV Proceedings* (Apr.). ACM, New York, 224–234.

Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June), 145–188.

Corbett, P., Baylor, S., and Feitelson, D. 1993. Overview of the Vesta parallel file system. *Comput. Arch. News 21*, 5 (Dec.), 7–14.

Cristian, F. 1991. Reaching agreement on processor group membership in synchronous distributed systems. *Distrib. Comput. 4*, 175–187.

Cypher, R., Ho, A., Konstantinidou, S., and Messina P. 1993. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th International Symposium on Computer Architecture* (May). IEEE Computer Society Press, Los Alamitos, Calif., 2–13.

Dahlin, M., Mather, C., Wang, R., Anderson, T., and Patterson, D. 1994a. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference* (May). ACM, New York, 150–160.

Dahlin, M., Wang, R., Anderson, T., and Patterson, D. 1994b. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation* (Nov.), 276–280.

Dibble, P. and Scott, M. 1989. The Bridge multiprocessor file system. *Comput. Arch. News 17*, 5 (Sept.), 32–39.

Douglis, F. and Ousterhout, J. 1991. Transparent process migration: Design alternatives and the Sprite implementation. *Softw. Pract. Exp. 21*, 8 (July), 757–785.

Hagersten, E., Landin, A., and Haridi, S. 1992. DDM--A cache-only memory architecture. *IEEE Comput. 25*, 9 (Sept.), 45–54.

Hartman, J. and Ousterhout, J. 1995. The Zebra striped network file system. *ACM Trans. Comput. Syst. 13*, 3 (Aug.), 274–310.

Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, W. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 51–81.

Kazar, M. 1989. Ubik: Replicated servers made easy. In *Proceedings of the 2nd Workshop on Workstation Operating Systems* (Sept.). IEEE Computer Society Press, Los Alamitos, Calif., 60–67.

KEETON, K., ANDERSON, T., AND PATTERSON, D. 1995. LogP quantified: The case for low-overhead local area networks. In *Proceedings of Hot Interconnects* (Aug.). IEEE Computer Society Press, Los Alamitos, Calif.

KISTLER, J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.), 3–25.

KUBIATOWICZ, J. AND AGARWAL, A. 1993. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of the 7th International Conference on Supercomputing* (July). ACM, New York.

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 21st International Symposium on Computer Architecture* (Apr.). IEEE Computer Society Press, Los Alamitos, Calif., 302–313.

LEE, E. 1995. Highly-available, scalable network storage. In *Proceedings of COMPCON* 95. IEEE, New York.

LEFF, A., YU, P., AND WOLF, J. 1991. Policies for efficient memory utilization in a remote caching architecture. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems* (Dec.). IEEE Computer Society Press, Los Alamitos, Calif., 198–207.

LENOSKI, K., LAUDON, J., GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture* (May). IEEE Computer Society Press, Los Alamitos, Calif., 148–159.

LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. 1991. Replication in the Harp file system. In *Proceedings of the 13th Symposium on Operating Systems Principles* (Oct.). ACM, New York, 226–238.

LITZKOW, M. AND SOLOMON, M. 1992. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of the Winter 1992 USENIX* (Jan.). USENIX Assoc., Berkeley, Calif., 283–290.

LO VERSO, 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*. USENIX Assoc., Berkeley, Calif., 291–305.

MAJOR, D., MINSHALL, G., AND POWELL, K. 1994. An overview of the NetWare operating system. In *Proceedings of the 1994 Winter USENIX* (Jan.). USENIX Assoc., Berkeley, Calif., 355–372.

MCKUSICK, M., JOY, W., LEFFLER, S., AND FABRY, R. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3 (Aug.), 181–197.

NELSON, M., WELCH, B., AND OUSTERHOUT, J. 1988. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 134–154.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks (RAID). In the *International Conference on Management of Data* (June). ACM, New York, 109–116.

PIERCE, P. 1989. A concurrent file system for a highly parallel mass storage subsystem. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*. Golden Gate Enterprises, Los Altos, Calif., 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* (Nov.). IEEE Computer Society Press, Los Alamitos, Calif., 5–10.

RASHID, R. 1994. Microsoft's Tiger media server. In *The 1st Networks of Workstations Workshop Record* (Oct.). Presented at ASPLOS 1994 Conference (San Jose, Calif.).

RICCIARDI, A. AND BIRMAN, K. 1991. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th Symposium on Principles of Distributed Computing* (Aug.). ACM, New York, 341–353.

ROSENBLUM, M. AND OUSTERHOUT, J. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.), 26–52.

ROSTI, E., SMIRNI, E., WAGNER, T., APON, A., AND DOWDY, L. 1993. The KSR1: Experimentation and modeling of Poststore. In *Proceedings of 1993 SIGMETRICS* (June). ACM, New York, 74–85.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun network file system. In *Proceedings of the Summer 1985 USENIX* (June). USENIX Assoc., Berkeley, Calif., 119–130.

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 J. Sel. Areas Commun. 9*, 8 (Oct.), 1318–1335.

SELTZER, M., BOSTIC, K., MCKUSICK, M., AND STAELIN, C. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the 1993 Winter USENIX* (Jan.). USENIX Assoc., Berkeley, Calif., 307–326.

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 1995 Winter USENIX* (Jan.). USENIX Assoc., Berkeley, Calif.

SMITH, A. 1977. Two methods for the efficient analysis of memory address trace data. *IEEE Trans. Softw. Eng. SE-3*, 1 (Jan.), 94–101.

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 19th International Symposium on Computer Architecture* (May). IEEE Computer Society Press, Los Alamitos, Calif., 256–266.

WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. 1983. The LOCUS distributed operating system. In *Proceedings of the 5th Symposium on Operating Systems Principles* (Oct.). ACM, New York, 49–69.

WANG, R. AND ANDERSON, T. 1993. xFS: A wide area mass storage file system. In the *4th Workshop on Workstation Operating Systems* (Oct.). IEEE Computer Society Press, Los Alamitos, Calif., 71–78.

WILKES, J., AND GOLDING, R., STAELIN, C., AND SULLIVAN, T. 1995. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Dec.). ACM, New York, 96–108.

WOLF, J. 1989. The placement optimization problem: A practical solution to the disk file assignment problem. In *Proceedings of the 1989 SIGMETRICS* (May). ACM, New York, 1–10.

Received September 1995; revised October 1995; accepted October 1995