Serverless Network File Systems

Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe,
David A. Patterson, Drew S. Roselli, and Randolph Y. Wang

Computer Science Division
University of California at Berkeley

Abstract

In this paper, 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. Further, 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.

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 [Majo94], NFS [Sand85], Andrew [Howa88], and Sprite [Nels88] where a central server machine provides all file system services. 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.

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. The authors can be contacted at {tea, dahlin, neefe, patterson, drew, rywang}@CS.Berkeley.EDU.

Copyright (c) 1995 by the Association for Computing Machinery, Inc. (ACM)

The recent introduction of switched local area networks such as ATM or Myrinet [Bode95] 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 towards low latency network interfaces [vE92, Basu95] 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 [Bake91]; 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 [Rash94]. Coordinated Networks of Workstations (NOWs) allow users to migrate jobs among many machines and also permit networked workstations to run parallel jobs [Doug91, Litz92, Ande95]. By increasing the peak processing power available to users, NOWs increase peak demands on the file system [Cyph93].

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. 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 [Wolf89]. 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, AFS [Howa88] 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 [Walk83, Kaza89, Pope90, Lisk91, Kist92, Birr93] for high availability. Replication increases the cost and complexity of central servers, and can also 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 [Patt88, Chen94] using log-based network striping similar to Zebra's [Hart95]. Finally, xFS eliminates central server caching by taking advantage of cooperative caching [Leff91, Dahl94b] to harvest portions of client memory as a large, global file cache.

This paper 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 [Leno90] and Alewife [Chai91]), disk striping (RAID and Zebra), log structured file systems (Sprite LFS [Rose92] and BSD LFS [Selt93]), and cooperative caching. 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 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 [Kubi93, Kusk94].

xFS could also be used in a mixed environment, containing both "core" trusted machines connected by fast networks and "fringe" clients that are either connected to the core by a slower network or that are less trusted [Howa88]. In such an environment the core machines would act as a traditional — though scalable, reliable, and cost effective — file server for the fringe clients. xFS permits clients to use NFS [Sand85] as one such fringe protocol, allowing the core xFS system to act as a scalable and reliable NFS server for unmodified UNIX clients.

We have built a prototype that demonstrates most of xFS's key features, including distributed management, network disk striping with parity and multiple groups, and cooperative caching. 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. We present both simulation results of the xFS design and a few preliminary measurements of the prototype. Although the prototype is largely untuned, it demonstrates 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 paper 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's distributed log cleaner, Section 5 outlines xFS's 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's network disk storage exploits the high performance and availability of Redundant Arrays of Inexpensive Disks (RAIDs). We use a Log-structured File System (LFS) to organize this storage, largely because Zebra demonstrated how to exploit the synergy between RAID and LFS to provide high performance, reliable writes to disks 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 [Wang93], 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. 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 storage provides fault tolerance — it can reconstruct the contents of a failed disk by taking the exclusive-OR of the remaining data blocks and the parity block. 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 [Blau94].

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 [Bake91], 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 incorporates LFS because it 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's 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) [McKu84], to store pointers to the system's data blocks. However, where FFS's inodes reside in fixed locations, LFS's 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's 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. Further, some workloads, such as transaction processing, incur larger cleaning overheads [Selt93, Selt95].

2.3. Zebra

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

LFS's 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 ver-

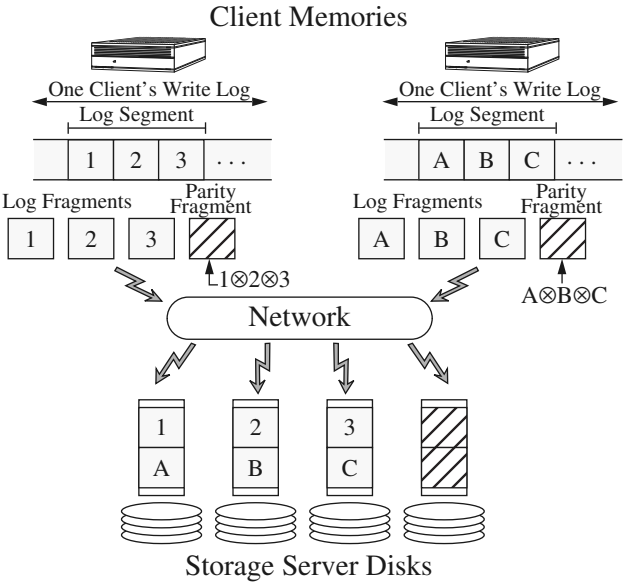

Figure 1. Log-based striping used by Zebra and xFS. Each client writes its new file data into a single append-only log and stripes this log across the storage servers. Clients compute parity for segments, not for individual files.

sions of the block, to allow the system to replay the modification during recovery. Deltas greatly simplify 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 towards 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. To increase the numbers of storage servers in a system, Zebra must either reduce the fragment size (reducing the efficiency of the writes) or increase the size of the segment (increasing memory demands on the clients); even if the system were to increase the segment size, syncs would often force clients to write partial segments to disk, again reducing write efficiency [Bake92].

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 [Leno90] and Alewife [Chai91] 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]

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. 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 [Wang93, Dahl94a]. 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 [Dahl94b] 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 dynamically reconfigure responsibilities 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. Further, 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 system. To provide scalable control of

[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.

[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.

disk metadata and cache consistency state, xFS splits management among *metadata managers* similar to multiprocessor consistency managers. Unlike multiprocessor managers, xFS managers can dynamically alter the mapping from a file to its manager. 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.

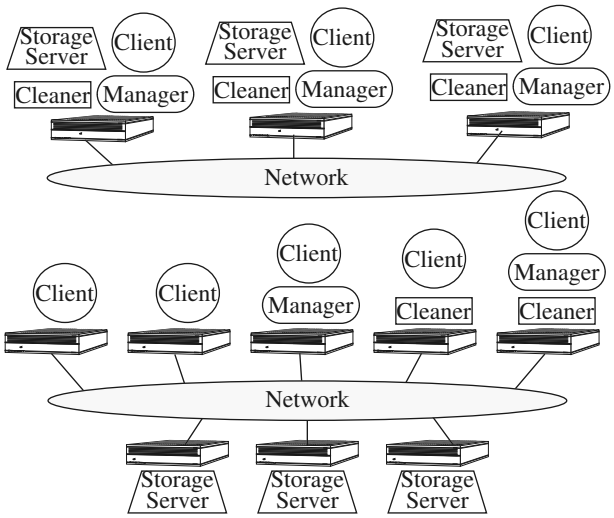

Figure 2. Two simple xFS installations. In the first, each machine acts as a client, storage server, cleaner, and manager, while in the second each node only performs some of those roles. The freedom to configure the system is not complete. Managers and cleaners access storage using the client interface,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

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 1 provides a summary of these and other key xFS 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. Where multiprocessor cache consistency distribution 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. The map can also act as a coarse-grained load balancing mechanism to split the work of overloaded managers.

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

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

Table 1. Summary of key xFS data structures. 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 paper the structure is described.

managers send the corresponding part 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.

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's 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 currently choose its index number so that the file's manager is on the same machine as the client that created the file. Section 3.2.4 describes simulation results of the effectiveness of this policy in reducing network communication.

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 assigning multiple managers to different portions of the same file to balance load and provide locality for parallel workloads.

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 [Chen94]. 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 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 [Hart95]; 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 co-located with the client, this message requires a network hop.

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

If no other client can supply the data from DRAM, 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.)

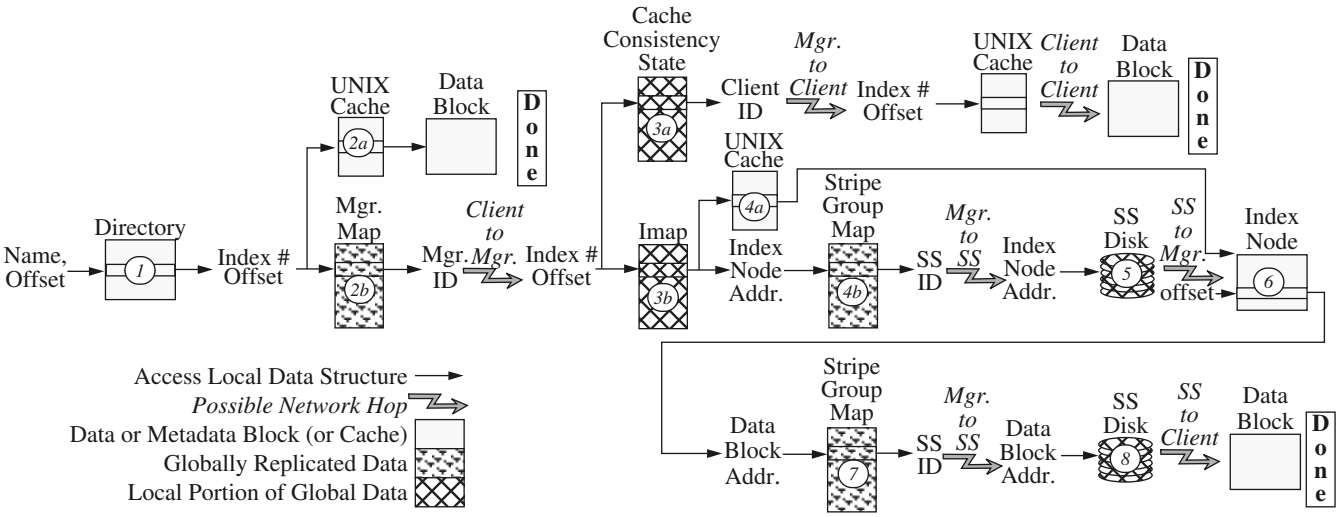

Figure 3. Procedure to read a block. The circled numbers refer to steps described in Section 3.2.1. The network hops are labelled as "possible" because clients, managers, and storage servers can run on the same machines. For example, xFS tries to co-locate the manager of a file on the same machine as the client most likely to use the file to avoid some of the network hops. "SS" is an abbreviation for "Storage Server."

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 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 "simultaneously" commit both index nodes and their data blocks because the client's log includes a *delta* that allows reconstruction of the manager's data structures in the event of a client or manager crash. We discuss deltas in more detail in Section 5.2.

As in BSD LFS [Selt93], each manager caches its portion of the imap in memory, storing 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 5.2.

3.2.3. Cache Consistency

xFS utilizes a token-based cache consistency scheme similar to Sprite [Nels88] and AFS [Howa88] 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 co-located on that machine. This section presents a simulation study that examines policies for assigning files to managers. We show that co-locating 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 co-located with that client. For comparison, we also simulated a Centralized policy that uses a single, centralized manager that is not co-located with any of the clients.

We examined management policies by simulating xFS's behavior under a seven day trace of 236 clients' NFS accesses to an Auspex file server in the Berkeley Computer Science Division [Dahl94a]. 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 16 MB of local cache and that there is a manager co-located 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 [Smit77], 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. We assign management of these files to random managers at the start of the trace; if they are later written in the trace, we reassign their management to the first writer in the trace. Since write sharing is rare — 96% of all block overwrites or deletes are by the block's previous writer — this heuristic will yield results close to a true "First Writer" policy.

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 re-

quest 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. 75% of 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 co-located 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 co-located 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.

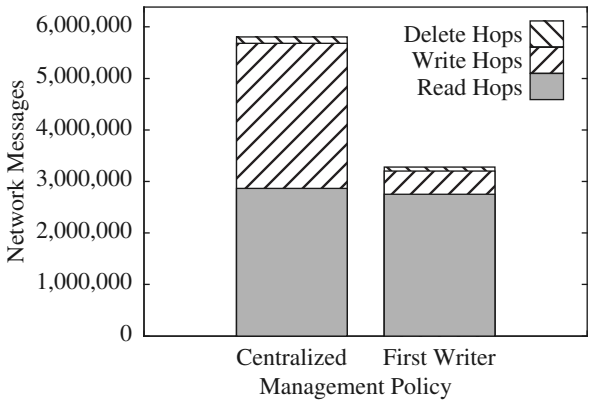

Figure 4. Comparison of locality as measured by network traffic for the Centralized and First Writer management policies.

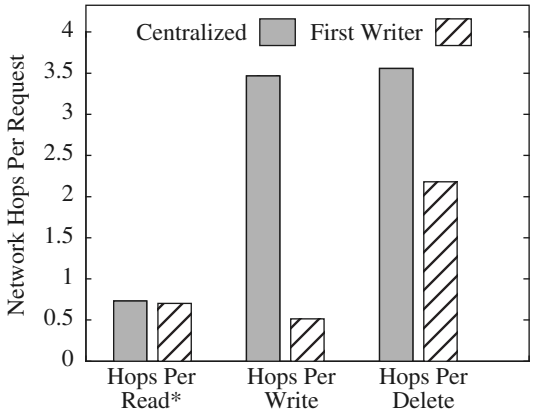

Figure 5. Average number of network messages needed to satisfy a read block, write block, or delete file request under the Centralized and First Writer policies. The Hops Per Write column does not include a charge for writing the segment containing block writes to disk because the segment write is asynchronous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

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.

4. Cleaning

When an LFS system such as xFS writes data by appending complete segments to its log, it deletes or overwrites blocks in old segments, leaving "holes" that contain no data. LFS systems use 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. The xFS architecture therefore provides for a distributed cleaner, although we have not completed implementation of the cleaner in the prototype.

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 [Rose92]. 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.

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 [Bake91, Kist92, Blaz93] 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. As 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 be-

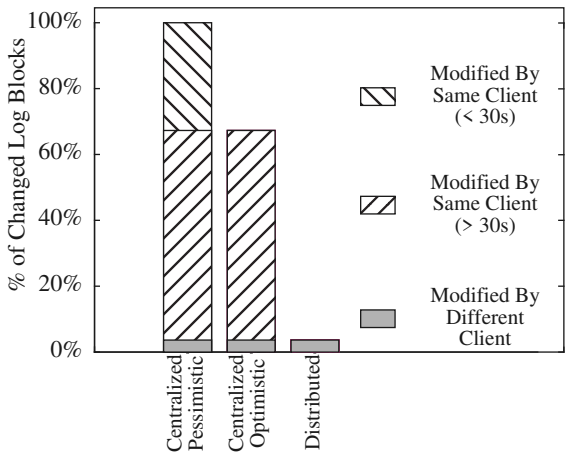

Figure 6. Simulated network communication between clients and cleaner. Each bar shows the fraction of all blocks modified or deleted in the trace, based on the time and client that modified the block. Blocks can be modified by a different client than originally wrote the data, by the same client within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

cause 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 [Bake92]. (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 eighteen 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 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 to 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. Fortunately, techniques to provide highly available file service with potentially unreliable components are known. RAID striping allows data stored on disk to be accessed despite disk failures, and Zebra demonstrated how to extend LFS recovery to a distributed system. Zebra's approach organizes recovery into a hierarchy where lower levels of recovery are performed first, followed by higher levels that depend on lower levels, as Figure 7 illustrates. Under this scheme, recovery proceeds in four steps:

1. Recover log segments stored on disk.
2. Recover managers' disk imap metadata by reading a manager checkpoint and the subsequent deltas from the log.
3. Recover managers' cache consistency state by querying clients.
4. Recover cleaners' state by reading cleaner checkpoints and rolling forward to update their s-files.

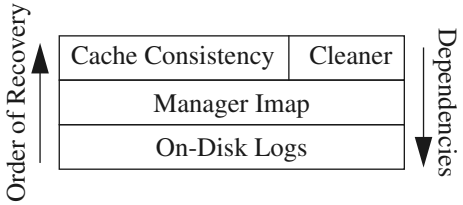

Figure 7. Bottom up recovery in xFS and Zebra rests on the persistent state stored reliably in the logs.

xFS leaves these basic techniques in place, modifying them only to avoid centralized bottlenecks.

These techniques allow xFS to be resilient to uncorrelated failures — for instance, users kicking power or network cords out of their sockets. When one xFS machine fails, access to unaffected clients, managers, and storage servers can continue. However, xFS can not continue operation when multiple machines from a single storage group fail or when a network partition prevents storage servers from regenerating segments.

The prototype currently implements only a limited subset of xFS's 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. Given the complexity of the recovery problem and the early state of our implementation, continued research will be needed to fully understand scalable recovery.

The rest of this section explores the issues involved in scaling the basic Zebra recovery model and discusses one additional aspect of recovery: reaching consensus on manager maps and stripe group maps.

5.1. Persistent State

The storage servers provide the keystone of the system's recovery and availability strategy by storing the system's persistent state in a redundant log structured file system. We base the storage servers' recovery on the Zebra design: after a crash, a storage server reads a local checkpoint block. This checkpoint preserves three sets of state: the storage server's internal mapping from xFS fragment IDs to the fragments' physical disk addresses, the storage server's map of free disk space, and a list of locations where the storage server was planning to store the next few fragments to arrive after the checkpoint.

After reading the checkpoint, the storage server examines the locations where it might have stored data just before the crash. It computes a simple checksum to determine if any of them contain live data, updating its local data structures if any do. Incomplete fragments that were being written at the time of the crash will fail this checksum and be discarded.

To help recover the stripe group map after a crash, xFS includes a field in each fragment that lists its stripe group and the other storage servers in that group.

Storage server recovery should scale well in xFS because each storage server can independently recover its local state and because storage servers' local checkpoints allow them to examine only small fractions of their disks to locate incomplete fragments.

5.2. Manager Metadata

To recover the managers' disk location metadata, xFS managers use the checkpoint and roll forward method developed in LFS and Zebra, but they split responsibility for rolling forward different components of the logs for scalability.

During normal operation, managers store modified index nodes and modified blocks of their ifile in their logs using the standard client interface. The ifile holds the imap containing pointers to the index nodes in the log, but to locate the ifile in the log after a crash, managers use *checkpoints* that they periodically store in their logs. Like BSD LFS [Selt93], xFS's checkpoints consist primarily of lists of pointers to the ifile's disk storage locations at the time of the checkpoints. The checkpoint also lists the segment ID of the last segment in each client's log at the time of the checkpoint.

To recover the manager's state in Zebra, the manager begins by reading its log backwards from the end of the log until it finds the last checkpoint. The manager reads the checkpoint to get pointers to the ifile blocks as they looked at the time of the checkpoint. Using those pointers, the manager recovers the imap from the ifiles. To account for more recent modifications, the manager then reads all of the clients' logs, starting at the time of the checkpoint and rolling forward its checkpoint state using the information in the logs' deltas to play back each modification.

To generalize this approach to handle multiple managers, xFS allows each new manager to recover a separate portion of the imap state. Three scalability issues arise. First, only one recovering manager should read each manager's log. Second, when replaying deltas, the system should read each client's log only once. Third, each machine involved in recovery must locate the tail of the logs it is to read.

To assign one manager to read each manager log, xFS uses the consensus algorithm described in Section 5.5 during recovery to create an initial manager map that assigns each manager's log to one of the new managers. That manager recovers the checkpoint from that log, restoring the portion of the imap formerly handled by the manager that wrote that log. By assigning each log to a manager, we parallelize recovery so that each manager recovers only a subset of the system's metadata, and we make this parallel recovery efficient by reading each log once.

xFS takes a similar approach for reading the deltas from clients' logs. It assigns a client or a manger to read the log and replay its deltas. Note that where a manager's log only contained information of interest to that manager, each client's log contains deltas that potentially affect all managers. Thus, the machine reading deltas from a client's log sends each delta to the manager that the delta affects. Like Zebra, managers use version numbers included in the deltas to order conflicting updates to the same data by different clients.

To enable machines to locate the tails of the logs they are to recover, each storage server keeps track of the newest segment that it stores for each client or manager. A machine can locate the end of the log it is to recover by asking all of the storage groups and choosing the newest segment.

Even with the parallelism and efficiency provided by xFS's 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, and the work done in each phase can be further limited by decreasing the interval between checkpoints.

For instance, to locate the tails of the systems logs, all machines involved in recovery must query all storage servers to locate the newest segment of the log being recovered. While this requires a total of $O(N^2)$ messages (each machine must ask each storage server group for the newest log segment stored at that group), each client or manager only needs to contact N storage server groups, and all of the clients and managers can proceed in parallel, provided that they take steps to avoid recovery storms where many machines simultaneously contact a single storage server [Bake94]. We plan use randomization to accomplish this goal.

Recovering the log checkpoint or rolling forward logs raises similar scaling issues. Although each manager or client must potentially contact all of the storage servers to read the logs, each log can be recovered in parallel. In fact, the actual number of storage servers contacted for each log will be controlled by the interval between checkpoints; shortening this interval reduces how far back in the log the system must scan and thereby reduces how many storage servers each manager or client must contact.

5.3. Cache Consistency State

After the managers have recovered and rolled forward the imap, they must recover the cache consistency state associated with the blocks they manage. xFS will use server-driven recovery [Bake94]. The manager contacts all of the system's clients, and they send the manager a list of the blocks that they are caching or for which they have write ownership from the indicated portion of the index number space. As with the other manager state, the N to N communication in this phase is tempered by its N-way parallelism.

5.4. Cleaner State

The xFS cleaners' state consists of segment utilization information that resides in the s-files. Since the s-files are normal xFS files, earlier levels of recovery recover them. However, because clients buffer their writes to the s-files, the s-files may not be completely up to date, even after the lower levels of recovery have rolled forward all of the deltas in the logs — the s-files may not account for modifications at about the time of the failure.

Cleaners combat this problem with a checkpoint and roll forward protocol. Each cleaner periodically flushes its s-files to disk and writes a *cleaner checkpoint* to a regular file in its s-directory. The checkpoint indicates the most recent segment that each client had written to its log at the time of the checkpoint. After xFS recovers the s-files and the checkpoints, each cleaner rolls forward the utilization state stored in its s-files by asking each client for a summary of the modifications since the cleaner checkpoint. Each client responds with a list of segments controlled by that cleaner that the client modified since the time of the cleaner checkpoint. This list includes a count of how many "holes" that client created in each modified segment. The cleaner updates its s-files by decrementing the utilization of each segment by the total number of "holes" created by clients since the cleaner checkpoint.

Clients create these summaries when they scan their logs during the main xFS roll-forward phase. As a client reads the deltas from each segment, it tallies the modifications that writes to that segment made to other segments.

A drawback of this approach is that it can decrement a segment's utilization twice for the same modification. For instance, a cleaner can store an s-file to disk between the time of a cleaner checkpoint and a crash. In that case, the cleaner will use client summaries that include modifications already reflected in the s-files. This mistake will result in the segment being cleaned too early, but no permanent damage is done. When the cleaner cleans the segment, it reads the deltas from that segment, correctly identifies all of the live blocks, and moves them to a new segment.

5.5. Reconfiguration and Consensus

xFS reconfigures its manager map and stripe group map when the system recovers from a crash or when machines are added or removed. Although we have not yet implemented dynamic reconfiguration of either of these data structures in the prototype, we plan to do so as follows. When the system detects a configuration change, it initiates a global consensus algorithm that elects a leader from among the active machines and supplies that leader with a list of currently active nodes. We will adapt the spanning tree algorithm used by Autonet for reconfiguration for this purpose [Schr91]. The leader then computes a new manager or stripe group map and then distributes it to the rest of the nodes.

In the case of incremental configuration changes — when a machine is added or removed or one or a small number of machines crash — the system can continue operation throughout this process. For stripe group map reconfiguration, clients can continue to read from soon to be obsolete stripe groups using the old map, and if they try to write to a storage server that has left the system, they will find out about the missing machine and either rewrite the segment to a new, undamaged stripe group or simply write the segment without parity protection. In the case of a manager map change, access to unaffected managers can continue, but accesses to portions of the map being reconfigured have to wait until the management assignments have been transferred.

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 in 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 environments that do not trust all desktop machines, the 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 [Kubi93, Kusk94]. 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 can not 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 8 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's 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 non-volatile RAM optimization found in most commercial NFS servers [Bake92], so for best performance, NFS clients should mount these partitions using the "unsafe" option to allow xFS to buffer writes in memory.

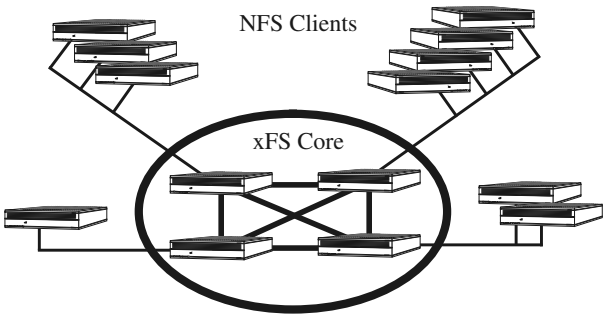

Figure 8. An xFS core acting as a scalable file server for unmodified NFS clients.

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 13.9 MB/s, close to a linear speedup compared to a single client's 0.6 MB/s bandwidth. Our other tests indicated similar speedups for reads and small file writes.

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's key features, including distributed management, network disk striping with single parity and multiple groups, and cooperative caching. We have not yet completed implementation of a number of other features. The most glaring deficiency is in xFS's crash recovery procedures. Although the system can automatically reconstruct data when a storage server crashes, 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 storage group maps; the system currently reads these mappings from a non-xFS file and can not change them. We have not implemented code to change a file's index number to dynamically assign it a new manager after it has been created, and we have yet to implement the cleaner. Finally, xFS is still best characterized as a research prototype; although the system is becoming more stable over time, considerable stress testing is needed before real users will want to entrust their data to it.[3]

[3] The current version of the xFS source tree is available at http://now.cs.berkeley.edu/Xfs/release/sosp95-snapshot.tar.Z. We make this code available to provide detailed documentation of our design as of August 1995, not with the illusion that anyone will be able to download the code and start running xFS. In the future, we plan to provide more stable releases of xFS in that directory.

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's interface to the Solaris v-node layer and also accesses the in-memory file cache. We implemented the remaining three pieces of xFS as daemons outside of the kernel address space to facilitate debugging. If the xFS kernel code cannot satisfy a request using the buffer cache, the it sends the request to the client daemon. The client daemons provide the rest of xFS's functionality by accessing the manager daemons and the storage server daemons over the network.

7.2. Test Environment

For our testbed, we used a total of 32 machines: eight dual-processor SPARCStation 20's, and 24 single-processor SPARCStation 10's. Each of our machines had 64 MB of physical memory. Uniprocessor 50 MHz SS-20's and SS-10's have SPECInt92 ratings of 74 and 65, and can copy large blocks of data from memory to memory at 27 MB/s and 20 MB/s, respectively.

For our NFS tests, we use one of the SS-20's as the NFS server and the remaining 31 machines as NFS 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 256 MB partition of a 1.1 GB Seagate-ST11200N disk. These disks have an advertised average seek time of 5.4 ms and rotate at 5,411 RPM. We measured a peak bandwidth to read from the raw disk device into memory of 2.7 MB/s for these disks. For all xFS tests, we use a log fragment size of 64 KB, 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 NFS server uses a faster disk than the xFS storage servers, a 2.1 GB DEC RZ 28-VA with a peak bandwidth of 5 MB/s from the raw partition into memory. The NFS server also uses a Prestoserve NVRAM card that acts as a buffer for disk writes [Bake92]. We did not use an NVRAM buffer for the xFS machines, although xFS's log buffer provides similar performance benefits.

A high-speed, switched Myrinet network [Bode95] connects the machines. Although each link of the physical network has a peak 80 MB/s bandwidth, RPC and TCP/IP protocol overheads place a much lower limit on the throughput actually achieved [Keet95]. 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 3.2 MB/s for 8 KB packets when using this source release. For the NFS 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 8.4 MB/s 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. RPC overheads further reduce network performance for both 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. We examine read and write throughput for large files and write performance for small files.

These performance results are preliminary. As noted above, several significant pieces of the xFS system remain to be implemented. 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 as in AFS [Howa88]. We took this approach to simplify debugging, but it hurts performance since each user/kernel space crossing requires the kernel to schedule the user level process and copy data to or from the user process's address space. To fix this limitation, we are working to move xFS into the kernel.
2. RPC and TCP/IP overheads severely limit xFS's network performance. We have begun to port xFS's communications layer to Active Messages [vE92] to address this issue.
3. We have done little profiling and tuning. As we do so, we expect to find and fix inefficiencies.

Despite these limitations, the prototype is sufficient to demonstrate the scalability of the xFS architecture. However, the absolute performance is much less than we expect for the well-tuned xFS. As the implementation matures, we expect one xFS client to significantly outperform one NFS 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.

7.3.1. Scalability

Figures 9 through 11 illustrate the scalability of xFS's performance for large writes, large reads, and small writes. For each of these tests, as the number of clients increases, so does xFS's aggregate performance. In contrast, NFS's single server is saturated by just a few clients for each of these tests, limiting peak throughput.

Figure 9 illustrates the performance of our disk write throughput test, in which each client writes a large (10 MB), private file and then invokes sync() to force the data to disk (some of the data stay in NVRAM in the case of NFS.) A single xFS client is limited to 0.6 MB/s, about half of the 1.2 MB/s throughput of a single NFS client; this difference is largely due to the extra kernel crossings and associated data copies in the user-level xFS implementation as well as high network protocol overheads. As we increase the number of clients, NFS's throughput does not increase while the xFS configuration scales up to a peak bandwidth of 13.9 MB/s 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 10 illustrates the performance of xFS and NFS for large reads from disk. For this test, each machine flushed its in-memory file cache and then sequentially read a per-client 10 MB file. Again, a single NFS client outperforms a single xFS client. One NFS client can read at 1.2 MB/s, while the user-level xFS implementation and network overheads limit one xFS client to 0.9 MB/s. As is the case for writes, xFS exhibits good scalability; 32 clients achieve a read throughput of 13.8 MB/s. In contrast, five clients saturate NFS at a peak throughput of 2.7 MB/s.

Figure 11 illustrates the performance when each client creates 2,048 files containing 1 KB of data per file. For this benchmark, not only does xFS scale well, its absolute performance is greater than that of NFS, even with one client. Where one xFS client can create 40 files per second, an NFS client can create only 22 files per second. In the single client case for this benchmark, the benefits of xFS's log-based striping for both network and disk efficiency outweigh the limitations of our current implementation. xFS also demonstrates good scalability for this benchmark. 32 xFS clients were able to generate a total of 1,122 files per second, while NFS's peak rate was 86 files per second with five clients.

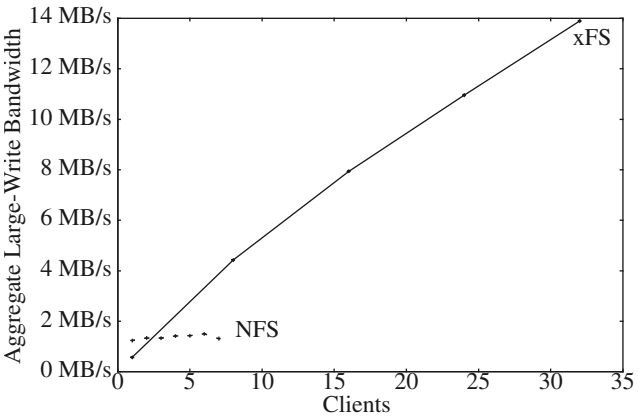

Figure 9. Aggregate disk write bandwidth. The x axis indicates the number of clients simultaneously writing private 10 MB files, and the y axis indicates the total throughput across all of the active clients. xFS used four groups of eight storage servers and 32 managers. NFS's peak throughput is 1.5 MB/s with 6 clients; xFS's is 13.9 MB/s with 32 clients.

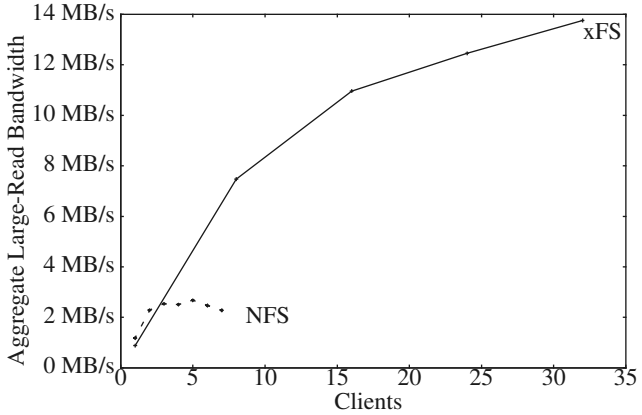

Figure 10. Aggregate disk read bandwidth. The x axis indicates the number of clients simultaneously reading private 10 MB files and the y axis indicates the total throughput across all active clients. xFS used four groups of eight storage servers and 32 managers. NFS's peak throughput is 2.7 MB/s with 5 clients; xFS's is 13.8 MB/s with 32 clients.

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 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 also as we change the stripe group size.

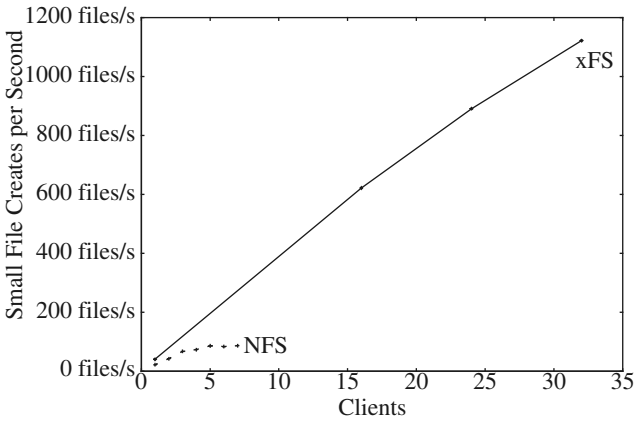

Figure 11. Aggregate small write performance. The x axis indicates the number of clients, each simultaneously creating 2,048 1 KB files. The y axis is the average aggregate number of file creates per second during the benchmark run. xFS used four groups of eight storage servers and 32 managers. NFS achieves its peak throughput of 86 files per second with five clients, while xFS scales up to 1,122 files per second with 32 clients.

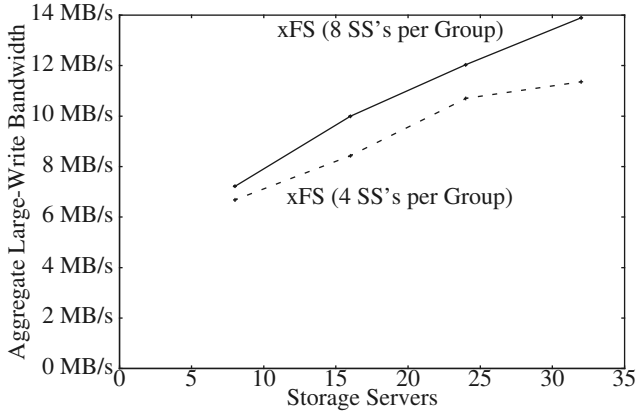

Figure 12. Large write throughput as a function of the number of storage servers in the system. The x axis indicates the total number of storage servers in the system and the y axis indicates the aggregate bandwidth when 32 clients each write a 10 MB file to disk. The 8 SS's line indicates performance for stripe groups of eight storage servers (the default), and the 4 SS's shows performance for groups of four storage servers.

Increasing the number of storage servers improves performance by spreading the system's requests across more CPUs and disks. The increase in bandwidth falls short of linear with the 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 locality. It varies the number of managers handling metadata for 31 clients running the small write benchmark.[4] This graph indicates that a single manager is a significant bottleneck for this benchmark. Increasing the system from one manager to two increases throughput by over 80%, and a system with four managers more than doubles throughput compared to a single manager system.

Continuing to increase the number of managers in the system continues to improve performance under xFS's First Writer policy. This policy assigns files to managers running on the same machine as the clients that create the files; Section 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 co-locate managers with the clients accessing a file more often.

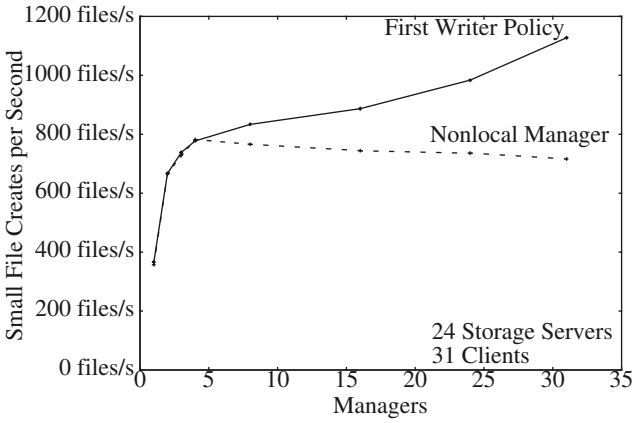

Figure 13. Small write performance as a function of the number of managers in the system and manager locality policy. The x axis indicates the number of managers. The y axis is the average aggregate number of file creates per second by 31 clients, each simultaneously creating 2,048 small (1 KB) files. The two lines show the performance using the First Writer policy that co-locates a file's manager with the client that creates the file, and a Nonlocal policy that assigns management to some other machine. Because of a hardware failure, we ran this experiment with three groups of eight storage servers and 31 clients. The maximum point on the x-axis is 31 managers.

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

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.

7.4. Limitations of these Measurements

Although these measurements suggest that the xFS architecture has significant potential, a great deal of future work remains to fully evaluate our design. First, the workloads examined here are microbenchmarks that provide significant parallelism and spread the load relatively evenly among xFS's components. Real workloads will include hot spots that may limit the scalability of xFS or may require xFS to rely more heavily on its capacity to reconfigure responsibilities to avoid loaded machines.

A second limitation of these measurements is that we compare against NFS. Our reasons for doing so were practical — NFS is a well known system, so it is easy for us to compare to and provides a good frame of reference — but its limitations with respect to scalability are well known [Howa88]. Further, since many NFS installations have attacked NFS's limitations by buying multiprocessor servers, it will be interesting to compare xFS running on workstations to NFS running on more powerful server machines than were available to us.

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.

Several file systems, such as CFS [Pier89], Bridge [Dibb89], and Vesta [Corb93], 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 [Cabrera91b] and SFS [LoVe93] provide redundant distributed data storage for parallel environments, and Tiger [Rash94] services multimedia workloads.

TickerTAIP [Cao93], SNS [Lee95], and AutoRAID [Wilk95] implement RAID-derived storage systems. These systems could provide services similar to xFS's 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 [Patt88] architecture makes cal-

culating 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 is actively being written to a RAID level 1 and migrating inactive data to a RAID level 5.

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. Further, 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 extensive use of a modified version of Mendel Rosenblum's LFS cleaner simulator. Eric Anderson and John Hartman also provided helpful comments on an earlier draft of this paper. Finally, we would like to thank the program committee — particularly John Ousterhout, our shepherd — as well as the other anonymous referees for their comments on initial drafts of this paper. These comments greatly improved both the technical content and presentation of this work.

References

[Ande95] T. Anderson, D. Culler, D. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64, February 1995.

[Bake91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th Symp. on Operating Systems Principles*, pages 198–212, October 1991.

[Bake92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *ASPLOS-V*, pages 10–22, September 1992.

[Bake94] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, 1994.

[Basu95] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th Symp. on Operating Systems Principles*, December 1995.

[Birr93] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo Distributed File System. Technical Report 111, Digital Equipment Corp. Systems Research Center, September 1993.

[Blau94] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In *Proc. of the 21st Symp. on Computer Architecture*, pages 245–254, April 1994.

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

[Bode95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, pages 29–36, February 1995.

[Cao93] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP Parallel RAID Architecture. In *Proc. of the 20th Symp. on Computer Architecture*, pages 52–63, May 1993.

[Chai91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *ASPLOS-IV Proceedings*, pages 224–234, April 1991.

[Chen94] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–188, June 1994.

[Corb93] P. Corbett, S. Baylor, and D. Feitelson. Overview of the Vesta Parallel File System. *Computer Architecture News*, 21(5):7–14, December 1993.

[Cyph93] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proc. of the 20th International Symposium on Computer Architecture*, pages 2–13, May 1993.

[Dahl94a] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proc. of 1994 SIGMETRICS*, pages 150–160, May 1994.

[Dahl94b] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[Dibb89] P. Dibble and M. Scott. Beyond Striping: The Bridge Multiprocessor File System. *Computer Architechture News*, 17(5):32–39, September 1989.

[Doug91] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(7), July 1991.

[Hart95] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *ACM Trans. on Computer Systems*, August 1995.

[Howa88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[Kaza89] M. Kazar. Ubik: Replicated Servers Made Easy. In *Proc. of the Second Workshop on Workstation Operating Systems*, pages 60–67, September 1989.

[Keet95] K. Keeton, T. Anderson, and D. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Proc. 1995 Hot Interconnects*, August 1995.

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

[Kubi93] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proc. of the 7th Internat. Conf. on Supercomputing*, July 1993.

[Kusk94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Internat. Symp. on Computer Architecture*, pages 302–313, April 1994.

[Lee95] E. Lee. Highly-Available, Scalable Network Storage. In *Proc. of COMPCON 95*, 1995.

[Leff91] A. Leff, P. Yu, and J. Wolf. Policies for Efficient Memory Utilization in a Remote Caching Architecture. In *Proc. of the First*

Internat. Conf. on Parallel and Distributed Information Systems, pages 198–207, December 1991.

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

[Lisk91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. of the 13th Symp. on Operating Systems Principles*, pages 226–238, October 1991.

[Litz92] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proc. of the Winter 1992 USENIX*, pages 283–290, January 1992.

[LoVe93] S. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. Milne, and R. Wheeler. sfs: A Parallel File System for the CM-5. In *Proc. of the Summer 1993 Usenix*, pages 291–305, 1993.

[Majo94] D. Major, G. Minshall, and K. Powell. An Overview of the NetWare Operating System. In *Proc. of the 1994 Winter USENIX*, pages 355–72, January 1994.

[McKu84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Trans. on Computer Systems*, 2(3):181–197, August 1984.

[Nels88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[Patt88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Internat. Conf. on Management of Data*, pages 109–116, June 1988.

[Pier89] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proc. of the Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 155–160, 1989.

[Pope90] G. Popek, R. Guy, T. Page, and J. Heidemann. Replication in the Ficus Distributed File System. In *Proc. of the Workshop on the Management of Replicated Data*, pages 5–10, November 1990.

[Rash94] R. Rashid. Microsoft's Tiger Media Server. In *The First Networks of Workstations Workshop Record*, October 1994.

[Rose92] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems*, 10(1):26–52, February 1992.

[Sand85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. of the Summer 1985 USENIX*, pages 119–130, June 1985.

[Schr91] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communication*, 9(8):1318–1335, October 1991.

[Selt93] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX*, pages 307–326, January 1993.

[Selt95] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX*, January 1995.

[Smit77] A. Smith. Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE Trans. on Software Engineering*, SE-3(1):94–101, January 1977.

[vE92] T. von Eicken, D. Culler, S. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of 1992 ASPLOS*, pages 256–266, May 1992.

[Walk83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proc. of the 5th Symp. on Operating Systems Principles*, pages 49–69, October 1983.

[Wang93] R. Wang and T. Anderson. xFS: A Wide Area Mass Storage File System. In *Fourth Workshop on Workstation Operating Systems*, pages 71–78, October 1993.

[Wilk95] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proc. of the 15th Symp. on Operating Systems Principles*, December 1995.

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