The Zebra Striped Network File System

John H. Hartman
John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Zebra is a network file system that stripes file data across multiple servers for increased file throughput. Rather than striping each file separately, Zebra forms all the new data from each client into a single stream, which it then stripes. This provides high performance for reads and writes of large files and also for writes of small files. Zebra also writes parity information in each stripe in the style of RAID disk arrays; this increases storage costs slightly but allows the system to continue operation even while a single storage server is unavailable. A prototype implementation of Zebra, built in the Sprite operating system, provides 5-8 times the throughput of the standard Sprite file system or NFS.

This work was supported in part by the National Science Foundation under grant CCR-8900029, the National Aeronautics and Space Administration and the Advanced Research Projects Agency under contract NAG 2-591, and the California MICRO Program.

1 Introduction

Zebra is a network file system that uses multiple file servers in tandem. The goal of the system is to provide greater throughput and availability than can be achieved with a single server. Clients *stripe* file data across servers so that different pieces of data are stored on different servers. Striping makes it possible for a single client to keep several servers busy and it distributes the load among the servers to reduce the likelihood of hot spots. Zebra also stores redundant information in each stripe (parity), which allows it to continue operation while any one server is unavailable.

In current network file systems the read and write bandwidth for a single file is limited by the performance characteristics of a single server, such as its network interface, memory bandwidth, processor speed, I/O busses (e.g. SCSI), and disks. It is possible to split the file system among multiple servers but it must be done at a coarse grain by assigning major subtrees of the file hierarchy to different servers. Each file must reside on a single server and it is difficult to balance the loads of the different servers. For example, the system directories often reside on a single server, making that server a hot spot.

The limitations of a single file server are likely to be even more severe in the future. For example, a single video playback can consume a substantial fraction of a file server's bandwidth even when the video is compressed. A cluster of workstations can greatly exceed the bandwidth of a file server if they all run video applications simultaneously; this problem will become much worse when video resolution increases with the arrival of HDTV. It would also be convenient for supercomputers and massively parallel machines to use the same file servers as workstations. This would free supercomputer designers from all I/O architecture issues except for the design of a good network interface, but it would present I/O loads to the network that far exceed the capabilities of today's file servers. Lastly, many research groups are exploring the possibility of using collections of workstations connected by high-speed low-latency networks to run massively parallel applications. These "distributed supercomputers" are likely to present I/O loads equivalent to those of other supercomputers.

A striping file system offers the potential to achieve very high performance using collections of inexpensive computers and disks. Several striping file systems have already been built, such as Swift [Cabrera91], and Bridge [Dibble88]. These systems are similar in that they stripe data within individual files, so only large files benefit from the striping. Zebra uses a different approach to striping borrowed from log-structured file systems [Rosenblum91]. Each client forms all of its new data for all files into a sequential log that it stripes across storage servers. This allows even small files to benefit from striping. It also reduces network overheads, simplifies the storage servers, and spreads write traffic uniformly across the servers.

Zebra's approach to striping also made it easy to use redundancy techniques from RAID disk arrays to improve availability and data integrity [Patterson88]. One of the fragments of each stripe stores parity for the rest of the stripe, which allows the stripe's data to be reconstructed in the event of a disk or server failure. This allows Zebra to continue operation while a server is unavailable, and even if a disk is totally destroyed Zebra can reconstruct its lost data.

We have constructed a prototype implementation of Zebra as part of the Sprite operating system [Ousterhout88]. Although it does not yet incorporate all of the reliability and recovery aspects of the Zebra architecture, it does demonstrate the performance benefits. For reads and writes of large files the prototype provides 5-8 times the throughput of either NFS or the standard Sprite file system (up to 4 Mbytes/second from a single client with four servers). For small files the Zebra prototype improves performance only 10-20% over Sprite, due to lack of file name caching. With proper name caching we would expect a greater speedup.

The rest of the paper is organized as follows. Sections 2 and 3 describe the RAID and log-structured-file-system technologies used in Zebra. Section 4 compares the common method of striping files with Zebra's logging approach. Section 5 describes the structure of Zebra, which consists of clients, storage servers, a file manager, and a stripe cleaner. Section 6 shows how the components of the system work together in operation, and Section 7 discusses how Zebra restores consistency after crashes. Section 8 describes the status of the Zebra prototype and presents some preliminary performance measurements. Section 9 discusses related work, and Section 10 concludes.

2 RAID: Striping With Parity

Zebra's design is based on two recent innovations in the management of disk storage systems. The first is RAID technology (Redundant Arrays of Inexpensive Disks) [Patterson88]. RAID is a storage system architecture in which many small disks work together to provide increased performance and data availability. A RAID appears to higher-level software as a single very large disk. Transfers to or from the disk array are divided into blocks called *striping units*. Consecutive striping units are assigned to different disks in the array as shown in Figure 1 and can be transferred in parallel. A group of consecutive striping units that spans the array is called a *stripe*. Large transfers can potentially proceed at the aggregate bandwidth of all the disks in the array, or multiple small transfers can be serviced concurrently by different disks.

Since a RAID has more disks than a traditional disk storage system, disk failures will occur more often. Furthermore, a disk failure anywhere in a RAID can potentially make the entire disk array unusable. To improve data integrity, a RAID reserves one of the striping units within each stripe for parity instead of data (see Figure 1): each bit of the

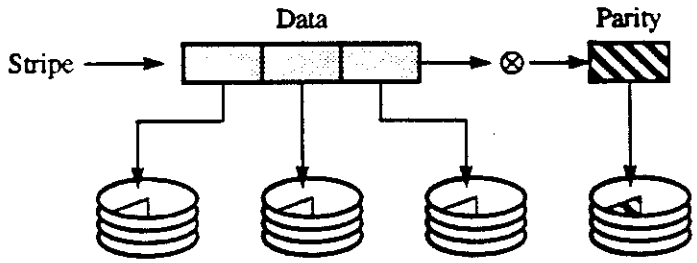

Figure 1. Striping with parity. The storage space of a RAID disk array is divided into stripes, where each stripe contains a striping unit on each disk of the array. All but one of the striping units hold data; the other striping unit holds parity information that can be used to recover after a disk failure.

parity striping unit contains the exclusive OR of the corresponding bits of the other striping units in the stripe. If a disk fails, each of its striping units can be recovered using the data and parity from the other striping units of the stripe. The file system can continue operation during recovery by reconstructing data on the fly.

A RAID offers large improvements in throughput, data integrity, and availability, but it presents two potential problems. The first problem is that the parity mechanism makes small writes expensive. If all write operations are in units of whole stripes, then it is easy to compute the new parity for each stripe and write it along with the data. This increases the cost of writes by only 1/(N-1) relative to a system without parity, where N is the number of disks in the array. However, small writes are much more expensive. In order to keep the stripe's parity consistent with its data, it is necessary to read the current value of the data block that is being updated, read the current value of the corresponding parity block, use this information to compute a new parity block, then rewrite both parity and data. This makes small writes in a RAID about four times as expensive as they would be in a disk array without parity. Unfortunately the best size for a striping unit appears to be tens of kilobytes or more [Chen90], which is comparable to the average file size in many environments [Baker91], so writes will often be smaller than a full stripe.

The second problem with disk arrays is that all the disks are attached to a single machine, so its memory and I/O system are likely to be a performance bottleneck. For example, a SCSI I/O bus can accommodate at least eight disks, each with a bandwidth of 1-2 Mbytes/second, but the SCSI bus has a total bandwidth of only 2-4 Mbytes/second. Additional SCSI busses can be added, but data must be copied from the SCSI channel into memory and from there to a network interface. On the DECstation 5000/200 machines used for Zebra these copies can only proceed at about 6-8 Mbytes/second. The Berkeley RAID project has built a special-purpose memory system with a dedicated high-bandwidth path between the network and the disks [Lee92] but even this system can support only a few dozen disks at full speed.

The way to eliminate performance bottlenecks is to separate the paths between the source or sink of data and the disks so that different paths are used to reach different disks. For example, this might be done by spreading the disks among different machines on a single very high speed network, or even by using different networks to reach different disks. Unfortunately, this turns the disk array into a distributed system and introduces issues such as who should allocate disk space or compute parity. One of our goals for Zebra was to solve these distributed system problems in a simple and efficient way.

3 Log-Structured File Systems

The second technology that drove the Zebra design is that of log-structured file systems (LFS) [Rosenblum91]. LFS is a disk management technique that treats the disk like an append-only log. This increases write performance without reducing read performance. When new files are created or existing files are modified, the new data are batched together and written to the end of the log in large sequential transfers. A single transfer can include data from many files plus metadata such as file attributes, block pointers, and directories. LFS is particularly effective for writing small files, since it allows them to be written at the full bandwidth of the disk without any seeks. Rosenblum

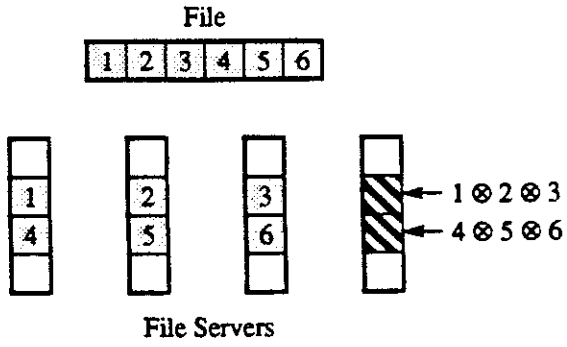

Figure 2. Per-file striping for a large file. The file is divided up into stripe units that are distributed among the servers. Each stripe contains one parity fragment.

reported a tenfold speedup over traditional file systems for writing small files. Log-structured file systems are also well-suited for use with RAIDs because they generate large writes and thereby avoid the expensive parity updates described above.

In order for LFS to operate efficiently there must always be large contiguous regions of free space on disk. This is accomplished by a special process called the *cleaner*, which is a form of copying garbage collector. The cleaner finds regions of disk with large amounts of free space, reads the portions that are still in use, and rewrites them at the end of the log, leaving the region completely free. Although the cost of cleaning is nontrivial (Rosenblum measured about 0.5 byte of I/O for cleaning for each byte of new data written), the cleaning can potentially be carried out during times of low load. In any case, the time LFS saves by eliminating seeks is much greater than the time lost to cleaning.

4 Striped Network File Systems

A striped network file system is one that distributes file data over more than one file server in the same way that a RAID distributes data over multiple disks. This allows multiple servers to participate in the transfer of a single file. The terminology we use to describe a striped network file system is similar to RAID's: a collection of file data that spans the servers is called a *stripe*, and the portion of a stripe stored on a single server is called a *stripe fragment*.

Network file systems that stripe have at least three potential advantages over those that don't. First, a striped network file system can provide a much higher file transfer bandwidth than any single server. Second, it allows the system's performance to be easily scaled by adding more servers. Third, striping makes it possible to use RAID parity techniques to provide high availability and data integrity at low cost. Specifically, one of the fragments for each stripe can store the parity of the other fragments in the stripe. When a server crashes, its data can be reconstructed from the data and parity on the other servers.

4.1 Per-File Striping

The most straightforward way to organize a striped network file system is to stripe each file separately, as shown in Figure 2. We refer to this method as *per-file striping*. Each file is stored in its own set of stripes. As a result, parity is computed on a per-file basis

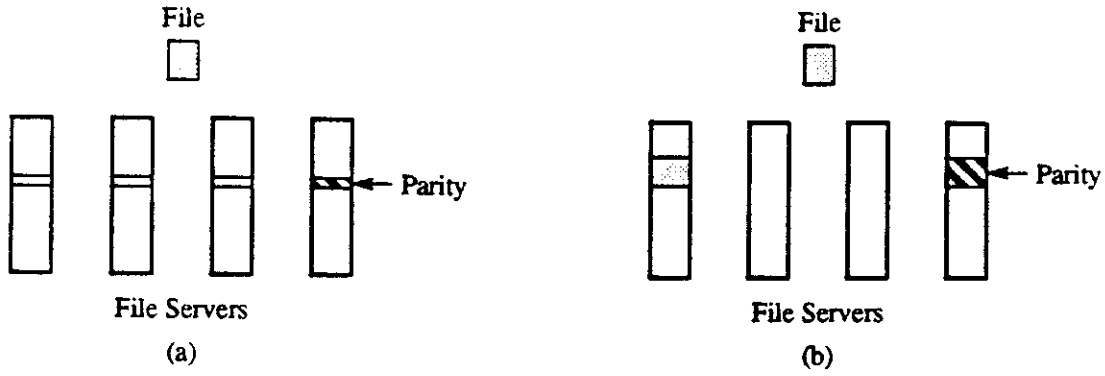

Figure 3. Per-file striping for a small file. In (a) the file is striped evenly across the servers, resulting in small fragments on each server. In (b) the entire file is placed on one server but the parity takes as much space as the file.

because each stripe contains data from only one file. While conceptually simple, per-file striping has several drawbacks. First, small files are difficult to handle efficiently. If a small file is striped across all of the servers as in Figure 3(a) then each server will only store a very small piece of the file. This provides little performance benefit, since most of the access cost is due to network and disk latency, yet it incurs overhead on every server for every file access. Thus it seems better to handle small file differently from large files and to store each small file on a single server, as in Figure 3(b). This leads to problems in parity management, however. If a small file is stored on a single server then its parity will consume as much space as the file itself, resulting in high storage overheads. Furthermore, the approach in Figure 3(b) can result in unbalanced disk utilization and server loading.

Per-file striping also leads to problems with parity management during updates. If an existing file is modified then its parity must be updated to reflect the modification. As with RAIDs, small updates like this require two reads (the old data and the old parity) followed by two writes (the new data and the new parity). Furthermore the two writes must be carried out atomically. If one write should complete but not the other (e.g. because a client or server crashed) then the parity will be inconsistent with the data; if this parity is used later for reconstructing lost data, incorrect results will be produced. There exist protocols for ensuring that two writes to two different file servers are carried out atomically [Bernstein81] but they are complex and expensive.

4.2 Per-Client Striping

Zebra does not use per-file striping; instead it uses a form of striping we call *per-client* striping. Figure 4 illustrates this approach. Per-client striping is inspired by log-structured file systems, which treat the storage system like an append-only log. Zebra can be thought of as a log-structured network file system. Each Zebra client organizes its new file data into an append-only log, which it then stripes across the file servers. The client computes parity for the log, not for individual files. Each client creates its own log, so a single stripe in the file system contains data written by a single client.

Per-client striping has a number of advantages over per-file striping. The first is that the servers are used efficiently regardless of file sizes. Large writes are striped, allowing them to be completed in parallel. Small writes, on the other hand, are batched together by the log mechanism and written to the servers in large transfers. No special handling is

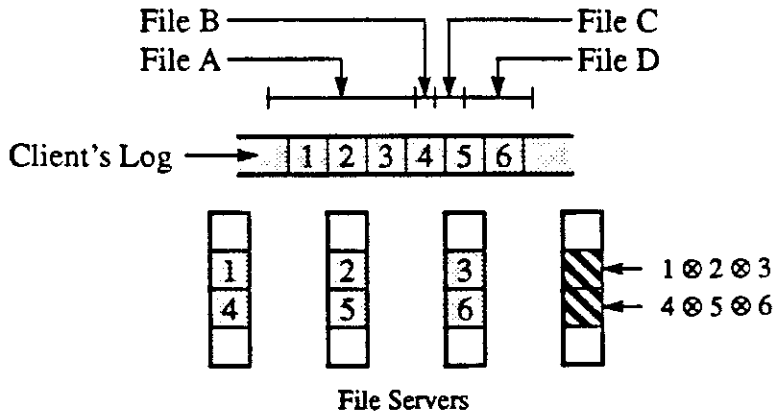

Figure 4. Per-client striping in Zebra. Each client forms its new file data into a single append-only log and stripes this log across the servers. In this example file A spans several servers while file B is stored entirely on a single server. Parity is computed for the log, not for individual files.

needed for either large or small files. Second, the parity mechanism is simplified. Each client computes parity for its own log without fear of interactions with other clients. Small files do not have excessive parity overheads because parity is not computed on a per-file basis. Furthermore, parity never needs to be updated because file data is never overwritten in place.

The above introduction to per-client striping leaves some unanswered questions. First, how can files be shared between client workstations if each client is writing its own log independently of the others? Zebra solves this problem by introducing a central *file manager*, separate from the storage servers, that manages metadata such as directories and file attributes. The file manager supervises interactions between clients. Second, how is free space reclaimed from the logs? Zebra solves this problem with a *stripe cleaner*, which is analogous to the cleaner in a log-structured file system. The next three sections provide a more detailed discussion of these issues and several others.

5 Zebra System Components

The Zebra file system contains four main components, as shown in Figure 5: clients, which are the machines that run application programs; storage servers, which store file data; a file manager, which manages the file and directory structure of the file system; and a stripe cleaner, which reclaims unused space on the storage servers. There may be any number of clients and storage servers but only a single file manager and stripe cleaner (the performance implications of having a single file manager and a single stripe cleaner are discussed in Sections 5.3 and 5.4, respectively). More than one of these components may share a single physical machine; for example, it is possible for one machine to be both a storage server and a client. The remainder of this section describes each of the components in isolation; Section 6 then shows how the components work together to implement operations such as reading and writing files, and Section 7 describes how Zebra restores consistency to its data structures after a crash.

We will describe Zebra as if each storage servers contains a single disk. However, this need not be the case. For example, storage servers could each contain several disks

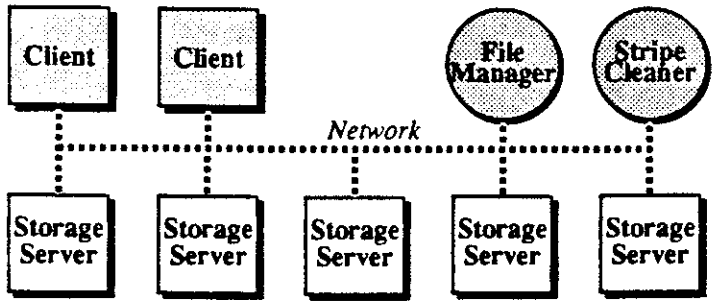

Figure 5: Zebra schematic. Squares represent individual machines; circles represent logical entities. The file manager and the stripe cleaner can run on any machine in the system, although it is likely that one machine will be designated to run both of them.

managed as a RAID, thereby giving the appearance to clients of a single disk with higher capacity and throughput. It is even possible to put all of the disks on a single server; clients would treat it as several logical servers, all implemented by the same physical machine. This approach would still provide many of Zebra's benefits: clients would still batch small files for transfer over the network, and it would still be possible to reconstruct data after a disk failure. However, a single-server Zebra system would limit system throughput to that of the one server and the system would not be able to operate when the server is unavailable.

5.1 Clients, Logs, And Deltas

Clients are machines where application programs execute. When an application reads a file the client must determine which stripe fragments store the desired data, retrieve the data from storage servers, and return it to the application. As will be seen below, the file manager keeps track of where file data is stored and provides this information to clients when needed. When applications write new data to files the client appends the data to its log by creating new stripes to hold the data and writing the stripes to the storage servers.

Each stripe contains two kinds of information: blocks and *deltas*. A block is just a piece of raw data from a file, i.e. the information that is read and written by applications. Deltas identify all changes in the contents of stripes, including both block creations and deletions. When a new block is added to a file a *create delta* is included in the stripe with the new block. The delta contains the location of the block in the stripe and a logical block identifier consisting of a file identifier, a position with the file, and a block version number. When a block is removed from a file then a *delete delta* is output in the client's log. It indicates the location the block used to occupy plus the block's logical identifier; the location usually refers to a different stripe than the one containing the delta. If a file block is overwritten then a delete delta is output for the old block, followed by a create delta for the new version.

Deltas play a key role in consistency and recovery. The file manager reads deltas in order to keep its block pointers consistent with what's in the stripes, and the stripe cleaner reads deltas to keep track of disk usage and reclaim unneeded space. Deltas are written to disk at the same time as file blocks, and they act as a "commit" mechanism for changes to

the file system. During crash recovery, deltas are replayed from the logs to restore consistency to the system.

Clients' logs do not contain file attributes, directories, or other metadata. This information is managed separately by the file manager as described below.

5.2 Storage Servers

The storage servers are the simplest part of Zebra. They are just repositories for stripe fragments. As far as a storage server is concerned, a stripe fragment is a large block of bytes with a unique identifier. All fragments in Zebra are the same size, which should be chosen large enough to make network and disk transfers as efficient as possible. In the Zebra prototype we use 512-Kbyte fragments. Identifiers are chosen by clients; in the Zebra prototype a fragment identifier consists of an identifier for the client that wrote the stripe, a serial number for the stripe, and the position of the fragment within the stripe.

Storage servers provide five operations:

Store a new fragment. The storage server allocates space for the fragment, writes the fragment to disk, and records the fragment's identifier and disk location for use in subsequent accesses. The operation is synchronous: it does not complete until the fragment is safely on disk.

Append to an existing fragment. This operation is similar to storing a new fragment except that it allows a client to write out a fragment in pieces if it doesn't have enough data to fill the entire fragment at once (e.g. due to an `fsync` by an application). Appends are implemented atomically so that a crash during an append cannot cause previously-written data to be lost.

Retrieve a fragment. This operation returns part or all of the data from a fragment. It is not necessary to read the entire fragment; an offset and length specify the desired range of bytes.

Delete a fragment. This operation is invoked by the stripe cleaner when the fragment no longer contains any useful data. It makes the fragment's disk space available for new fragments.

Identify fragments. This operation provides information about the fragments stored by the server, such as the most recent fragment written by a client. It is used to find the end of the clients' logs after a crash.

Stripes are immutable. A stripe may be created with a sequence of append operations, but existing data is never overwritten and once the stripe is complete it is never modified except to delete the entire stripe. The only exception to the rule of never overwriting existing data is that the parity fragment for a stripe may be overwritten during appends in order to keep it up to date with the current contents of the stripe (see Section 6.1).

5.3 The File Manager

The file manager stores all of the information in the file system except for file data. We refer to this information as *metadata*: it includes file attributes such as protection information, block pointers that tell where file data is stored, plus directories, symbolic

links, and special files for I/O devices. The file manager performs all of the usual functions of a file server in a network file system, such as name lookup and maintaining the consistency of client file caches. However, the Zebra file manager doesn't store any file data; where a traditional file server would manipulate data the Zebra file manager manipulates block pointers. For example, consider a read operation. In a traditional file system the client requests the data from the file server; in Zebra the client requests block pointers from the file manager, then it reads the data from the storage servers. For a write, a client in a traditional file system sends the new data to the file server; in Zebra, it sends the data to a storage server and the deltas, which contain block pointers, are eventually read by the file manager.

In the Zebra prototype we implemented the file manager using a Sprite file server with a log-structured file system to manage disk space. For each Zebra file there is one file in the file manager's file system, and the "data" in this file is an array of block pointers. This allows Zebra to use almost all of the existing Sprite network file protocols without modification. There is nothing in the Zebra architecture that requires Sprite to be used as the network file system, however: any existing network file server could be used in the same way by storing block pointers in files instead of data.

Since there is only a single file manager in Zebra it may seem that the file manager will be a performance and reliability bottleneck. However, we think that both of these potential problems can be avoided. Performance problems can be avoided by caching metadata information. For example, a client might cache entire directories, including the file names and their attributes and block pointers. The client could then read and write files in the directories without contacting the file manager. The file manager will only be involved when working sets change, when there are interactions between clients (e.g. one client reads a file that was recently written by another client), or when clients want to be sure that information is safely on disk. Of course, even with name caching there will be some system size at which the file manager will become a performance bottleneck, but the experience of systems like AFS [Howard88] and analyses like those of Floyd [Floyd89] and Shirriff [Shirriff92] suggest that a single file manager should be able to support a large number of clients. We decided not to implement name caching in the Zebra prototype because it would have required major modifications to the Sprite file system, but we would expect any production version of Zebra to incorporate name caching. Even without name caching the file manager is not a performance issue for large files (where most of the time is spent reading or writing data as opposed to opening and closing files) or for systems of modest size.

The second issue is availability and integrity. If all of the metadata is stored non-redundantly on the file manager then the file system will be unusable whenever the file manager is down and the loss of the file manager's disk will effectively destroy the file system. Fortunately these problems can be eliminated by using the Zebra storage servers to store the file manager's metadata. Instead of using a disk on the file manager to read and write the metadata, the metadata can be written to the storage servers as a special *meta-file* whose logical blocks correspond to the physical blocks of the disk being replaced. Updates to the meta-file are added to the file manager's client log, just like any other file, and striped across the storage servers with parity. This provides higher performance for the metadata than storing it on a local disk, and also improve its availability and integrity.

The file manager can run on any machine in the network, since it doesn't depend on having local access to a disk. If the file manager's machine should break then the file manager is restarted on another machine. Of course, if the file manager crashes then Zebra will be unavailable until the file manager restarts, but it should be possible to restart the file manager quickly (see Section 7).

5.4 The Stripe Cleaner

When a client writes a new stripe it is initially full of live data. Over time, though, blocks in the stripe become free, either because their files were deleted or because they were overwritten. If an application overwrites an existing block of a file, Zebra doesn't modify the stripe containing the block; instead it writes a new copy of the block to a new stripe. Zebra is similar to LFS in this respect. Since stripes are immutable, free blocks in a stripe cannot be reused directly. The only way to reuse free space in a stripe is to *clean* the stripe so that it contains no live data whatsoever, then delete the entire stripe. At this point the storage managers will reuse the stripe's disk space for new stripes.

The Zebra stripe cleaner runs as a user-level process and is very similar to the segment cleaner in a log-structured file system. It identifies stripes with a lot of free space, then it reads the remaining live data out of the stripe and writes them to a new stripe (by appending them to its client's log). Once this has been done, the stripe cleaner deletes the stripe's fragments from their storage servers. The deltas stored in stripes play a key role in stripe cleaning; they are used to choose which stripes to clean, and they identify the live data that's left in a stripe. Section 6.4 describes the cleaning algorithm in more detail.

One concern about the stripe cleaner is how much of the system's resources will be consumed by copying live data from one stripe to another. Rosenblum [Rosenblum91] found that only between 2% and 7% of the data in stripes that were cleaned was live and needed to be copied. Sprite LFS reads the entire segment even if only part of it is live, resulting in 0.5 byte of cleaning I/O for each byte written. Zebra reads only the live data out of the stripe. The net result is that the stripe cleaner should have little impact on system performance.

6 System Operation

This section describes several of the key algorithms in Zebra to show how the pieces of the system work together in operation. Most of these algorithms are similar to the approaches used in log-structured file systems, RAIDs, or other network file systems. All of the features described here are implemented in the Zebra prototype except for parity generation and data reconstruction.

6.1 Writing Files

In order for Zebra to run efficiently clients must collect large amounts of new file data and write them to the storage servers in large batches (ideally, whole stripes). The existing structure of the Sprite file caches made batching relatively easy. When an application writes new data it is placed in the client's file cache without writing it through to a server. The dirty data isn't written to a server until either (a) it reaches a threshold age (30

seconds in Sprite), (b) the cache fills with dirty data, (c) an application issues an `fsync` system call to request that the data be written through, or (d) the file manager requests that the data be written in order to maintain consistency among client caches. In many cases files are created and deleted before the threshold age is reached so their data never needs to be written at all.

When information does need to be written to disk, the client forms the new data into one or more stripe fragments and writes them to storage servers. We added support for asynchronous remote procedure calls to Sprite so that the client can transfer fragments to all of the storage servers concurrently. In addition the client can transfer the next stripe fragment to a storage server while it is writing the previous stripe fragment to disk, so that both the network and the disk are kept busy. As the fragments are written their parity is computed, and at the end of each stripe the parity is written to complete the stripe. In the Zebra prototype the client also sends the stripe's deltas to the file manager and stripe cleaner. This provides a performance optimization (avoiding potential disk accesses due to the file manager and stripe cleaner reading the deltas from the log directly) but it isn't necessary for correct operation. If the client should crash before sending the deltas then the file manager and stripe cleaner will read the deltas from the log on their own.

If a client is forced to write data in small pieces (e.g. because an application is invoking `fsync` frequently) then it fills the stripe a piece at a time, appending to the first stripe fragment until it is full, then filling the second fragment, and so on until the entire stripe is full. When writing partial stripes the client has two choices for dealing with parity. First, it can delay writing the parity until the stripe is complete. This is the most efficient alternative and it is relatively safe (the client has a copy of the unwritten parity, so information will only be lost if a disk is destroyed and the client crashes). For even greater protection, though, the client can update the stripe's parity fragment each time it appends to the stripe. Parity fragments written in this way include a count of the number of bytes of data in the stripe at the time the fragment was written, which is used to determine the relationship between the parity and the data after crashes. Parity updates are implemented by storage servers in a non-overwrite fashion, so either the old parity or the new parity is available after a crash.

6.2 Reading Files

File reads in Zebra are carried out in almost the same fashion as in a non-striped network file system. The client opens and closes the file in the same way as for a non-Zebra file; in Sprite this means a remote procedure call to the file manager for each open or close. Reading data is a two-step operation. First the client must fetch the block pointers from the file manager, then it reads the file data from the storage server. This results in an extra RPC relative to a non-striped file system; a better approach would be to return the block pointers as the result of the open RPC. Name caching would eliminate all of the RPCs except those to the storage servers (and file data caching might eliminate many of these too for small files).

For large files being accessed sequentially, Zebra prefetches data far enough ahead to keep all of the storage servers busy. As with writing, asynchronous RPCs are used to transfer data from all of the storage servers concurrently and to read the next stripe

fragment on a given server from disk while transferring the previous one over the network to the client.

The Zebra prototype does not attempt to optimize reads of small files, so each file is read from its storage server in a separate operation, just as for a non-striped file system. However, it is possible to prefetch for small files just as for large files (e.g. read entire stripes at a time, even if they cross file boundaries). If there is locality of file access so that groups of files are written together and then later read together, then this approach might improve read performance. We speculate that such locality exists for files in a directory but we have not attempted to verify its existence or capitalize on it in Zebra.

6.3 Client Cache Consistency

If a network file system allows clients to cache file data and also allows files to be shared between clients, then cache consistency is a potential problem. For example, one client might write a file that is cached on another client; if the other client subsequently reads the file, it must discard its stale cached data and fetch the new data. Zebra introduces no new issues vis-a-vis cache consistency. We chose to use the Sprite approach, which involves cache flushing or disabling when files are opened [Nelson88], because it was readily available, but any other approach could have been used as well. The only changes for Zebra occur when a client flushes a file from its cache. Instead of just returning dirty data to a file server, the Zebra client must write the dirty blocks to a storage server and the file manager must process all of the deltas for the blocks so that it can provide up-to-date block pointers to other clients.

6.4 Stripe Cleaning

The first step in cleaning is to select one or more stripes to clean. To do this intelligently the stripe cleaner needs to know how much live data is left in each stripe. Deltas are used to compute this information. The stripe cleaner processes the deltas from the client logs and uses them to keep a running count of space utilization in each existing stripe. For each create delta the cleaner increments the utilization of the stripe containing the new block, and for each delete delta the cleaner decrements the utilization of the stripe that contained the deleted block. In addition, the cleaner appends all of the deltas that refer to a given stripe to a special file for that stripe, called the *stripe status file*, whose use will be described below. The stripe status files are stored as ordinary files in the Zebra file system.

Given the utilizations computed above the stripe cleaner first looks for stripes with no live data. If any are found then the cleaner deletes the stripes' fragments from the storage servers and also deletes the corresponding stripe status files. If there are no empty stripes and more free space is needed then the cleaner chooses one or more stripes to clean. The policy it uses for this is identical to the one described by Rosenblum [Rosenblum91], i.e. a cost-benefit analysis is done for each stripe, which considers both the amount of live data in the stripe and the age of the data.

There are two issues in cleaning a stripe: identifying the live blocks, and copying them to a new stripe. The stripe status files make the first step easy: the cleaner reads the deltas in the stripe's status file and finds the create deltas for which there are no corresponding

delete deltas. Without the stripe status file this step would be much more difficult, since the delete deltas pertaining to the stripe could be spread throughout all of the stripes in the file system.

Once the live blocks have been identified the stripe cleaner copies them to a new stripe using a special kernel call. The kernel call reads one or more blocks from their storage servers, appends them to its client log, and writes the new log contents to the storage servers. For each block it includes both a delete and a create delta in its log. The kernel call for cleaning blocks is identical to reading and rewriting the blocks except that (a) it doesn't open the file or invoke cache consistency actions, (b) it needn't copy data out to the user-level stripe cleaner process and back into the kernel again, (c) it doesn't update the file's last-modified time, and (d) the new deltas have a special flag to indicate that they were created during cleaning (this information is used to deal with races as described below).

The deltas for the cleaned block are read by both the file manager and the stripe cleaner, using the same mechanism as for any other file changes. The file manager then updates its block pointers to refer to the new copy. The stripe cleaner uses the deltas to decrement the stripe's utilization; once all the deltas have been processed the stripe's utilization will become zero and the stripe will be deleted.

It is possible for an application to modify or delete a file block at the same time that the stripe cleaner is copying it. Without any synchronization a client might modify a block after the cleaner reads its old copy but before the cleaner rewrites the block, in which case the new data might be lost in favor of the rewritten copy of the old data. In the original LFS this race condition was avoided by having the cleaner lock all files that were being cleaned to prevent them from being modified until after cleaning was over. Unfortunately, this tended to produce lock convoys that effectively halted all normal file accesses and resulted in significant pauses during cleaning.

Zebra's stripe cleaner uses an optimistic approach similar to that of Seltzer et al. [Seltzer93]. It doesn't lock any files during cleaning or invoke any cache consistency actions. When a block is cleaned, the delta for the new block has the same version number as the old block, but it has a "cleaner" flag set (the cleaner can't increment the version number since a client might be doing this also; the cleaner flag is like a ".5" increase in version number). When the file manager processes the delta to update its metadata, it compares the version number in the delta with the block's current version. In the normal case where the file hasn't been modified during cleaning, the version numbers will match and the file manager replaces the old block address with the new one. It also sets the cleaner flag in the new block pointer. However, if the block was modified during cleaning then the version number from the cleaner will be lower than the one in the file manager's block pointer. In this case the cleaned block is irrelevant so the file manager doesn't update the block pointer. If the block is in the process of being modified but its new pointer hasn't yet reached the file manager, then the file manager will update its block pointer with the address of the cleaned block; when it later gets the new block pointer from the modification, this value will replace the pointer to the cleaned block.

Cleaning can also invalidate block pointers cached on clients. For example, suppose that a client has retrieved a block's address from the file manager but the block is copied

by the cleaner before the client retrieves it. If the client then tries to use the out-of-date block pointer, one of two things will happen. If the block's stripe still exists then the client can use it safely, since the cleaner didn't modify the old copy of the block. If the stripe has been deleted then the client will get an error from the storage server when it tries to read the old copy. This error indicates that the block pointer is out of date: the client simply discards it and fetches an up-to-date version from the file manager. The use of out-of-date block pointers by the clients can be avoided altogether by having the cleaner adhere to the client cache consistency protocol, but this amounts to having the cleaner lock the file, which is the reason for using an optimistic cleaning policy in the first place.

6.5 Data Reconstruction

Zebra's parity mechanism allows it to tolerate the failure of a single storage server using algorithms similar to those described for RAIDs [Patterson88]. To read a file while a storage server is down, the client must reconstruct any stripe fragment that was stored on the down server. This is done by computing the parity of all the other fragments in the same stripe; the result is the missing fragment. Writes intended for the down server are simply discarded. When the storage server restarts it reconstructs the unwritten data as described in Section 7.1.

For large sequential reads reconstruction is relatively inexpensive: all the fragments of the stripe are needed anyway, so the only additional cost is the parity calculation. For small reads reconstruction is expensive since it requires reading all the other fragments in the stripe. If small reads are distributed uniformly across the storage servers then reconstruction doubles the average cost of a read.

If data from a down server is being read frequently, it may be more efficient to save the reconstructed data for use by subsequent reads of the same data. There is no need to allocate additional space on the storage servers to store reconstructed fragments; they can be stored in place of the parity fragments for their stripes, which are not needed once the stripe's data has been reconstructed. Similarly, when writing a stripe the fragment intended for the down server can be written in place of the parity for the stripe. When the storage server reboots its restores the parity fragment as described in Section 7.1.

6.6 Adding A New Storage Server

Zebra's architecture makes it easy to add a new storage server to an existing system. All that needs to be done is to initialize the new server's disk(s) to an empty state and notify the clients, file manager, and stripe cleaner that each stripe now has one more fragment. From this point on new file data will be striped across the new server. The existing stripes can be used as-is even though they don't cover all of the servers; in the few places where the system needs to know how many fragments there are in a stripe (such as reconstruction after a server failure), it can detect the absence of a fragment for a stripe on the new server and adjust itself accordingly. Over time the old stripes will gradually get cleaned, at which point their disk space will be used for longer stripes that span all of the servers. Old stripes are likely to be cleaned before new ones since they will probably contain less live data. If it should become desirable for a particular file to be reallocated

immediately to use the additional bandwidth of the new server, this can be done by copying the file and replacing the original with the copy.

7 Consistency And Crash Recovery

If one or more of Zebra's components should crash due to a hardware or software failure then its state may be left inconsistent. For example, the file manager might crash before processing all of the deltas written by clients; when it reboots its metadata will not refer to the must up-to-date information in the clients' logs. When this happens Zebra must restore consistency to the metadata. In many respects crash recovery is no different in Zebra than in other network file systems. For example, the file manager will have to restore consistency to all of its on-disk structures. Since the file manager uses the same disk structures as a non-striped file system, it can also use the same recovery mechanism. In the Zebra prototype the metadata is stored in a log-structured file system, so we use the LFS recovery mechanism described by Rosenblum [Rosenblum91]. The file manager will also have to recover the information it uses to ensure client cache consistency; for this Zebra uses the same approach as in Sprite, which is to let clients reopen their files to rebuild the client cache consistency state [Nelson88]. If a client crashes then the file manager cleans up its data structures by closing all of the client's open files, also in the same manner as Sprite.

However, Zebra introduces three consistency problems that are not present in other file systems. These problems arise from the distribution of system state among the storage servers, file manager, and stripe manager; each of the problems is a potential inconsistency between system components. The first problem is that stripes may become internally inconsistent (e.g. some of the data or parity may be written but not all of it); the second problem is that information written to stripes may become inconsistent with metadata stored on the file manager; and the third problem is that the stripe cleaner's state may become inconsistent with the stripes on the storage servers. These three problems are discussed separately in the subsections below.

The solutions to all of the consistency issues are based on three interrelated techniques: logging, checkpoints, and idempotency. Logging means that operations are ordered so it is possible to tell what happened after a particular time and to revisit those operations in order. Logging also implies that information is never modified in place, so if a new copy of information is incompletely written the old copy will still be available. A checkpoint defines a system state that is internally consistent. To recover from a crash, the system initializes its state to that of the most recent checkpoint, then reprocesses the portion of the log that is newer than the checkpoint. Idempotency means that operations are expressed in a way such that the operation can be repeated several times without changing its effect. "Set x to 2" is an idempotent operation, whereas "add 1 to x" is not. Idempotency allows the system to be sloppy during recovery: if it's not sure how much of the log was processed before a crash, it can simply reprocess all of the entries that are in doubt.

The combination of these techniques allows Zebra to recover quickly after crashes. It need not consider any information on disk that is older than the most recent checkpoint. Zebra is similar to other logging file systems such as LFS, Episode [Chutani92], and the

Cedar File System [Hagmann87] in this respect. In contrast, file systems without logs, such as BSD Fast File System [McKusick84], cannot tell which portions of the disk were being modified at the time of a crash, so they must re-scan all of the metadata in the entire file system during recovery.

7.1 Internal Stripe Consistency

When a client crashes it is possible for the data of its newest stripe to be written without the parity or vice versa. The file manager is responsible for correcting this inconsistency (the file manager restores consistency immediately rather than letting the client do it on rebooting, so that correct parity information is available immediately if needed). The file manager must query the storage servers to identify the client's most recent stripe, then it reads the fragments of the stripe, verifies the parity, and rewrites it if it is incorrect. If some of the stripe's fragments are missing then the file manager computes the parity for those that exist.

When a storage server crashes two inconsistencies are possible. First, if a stripe fragment was being written at the time of the crash, it might not have been completely written. To detect incomplete stripe fragments, Zebra stores a simple checksum for each fragment. After they reboot, storage servers verify the checksums for fragments written around the time of the crash and discard any incomplete fragments.

The second inconsistency after a storage server crash is that it won't contain fragments for new stripes written while the it was down. As described in Section 6.5, these fragments were either discarded or stored in place of the stripe's parity fragment. After the storage server reboots it queries other storage servers to find out what new stripes were written. Then it reconstructs the discarded data and writes out the missing stripe fragments. If reconstructed data was stored in place of a stripe's parity, the storage server can fetch the data instead of recomputing it, but it must then recompute the parity and write that back to the appropriate server.

7.2 Stripes vs. Metadata

The file manager is responsible for maintaining consistency between the client logs and its metadata. To to do this it must ensure that it has processed all of the deltas written by clients and updated its block pointers accordingly. In the Zebra prototype the clients send new deltas to the file manager after writing them to storage servers, but if a client should crash the file manager must read the client's log to make sure it has processed all the deltas. If the file manager crashes, when it reboots it must check all the client logs to make sure it has processed all of their deltas. The file manager keeps track of its current position in each client's log and occasionally checkpoints this information to disk along with the metadata, so after a crash it only needs to re-read the deltas that are newer than its most recent checkpoint.

This approach still leaves three problems. First, some metadata updates could be processed by the file manager twice. For example, the file manager might process some deltas, write the corresponding metadata changes to disk, and then crash before writing out its current position in the client's log. When it reboots it will go back in the client's log

to the previous checkpoint and reprocess the deltas. Fortunately the block pointer updates are idempotent so there is no danger in reprocessing the deltas.

The second problem is that deltas might not be processed during crash recovery in the same order that they were originally created. For example, the easiest way for the file server to recover is to process the client logs one at a time without worrying about the ordering of events between logs. This could mean, for example, that the delete delta for a block is processed before its create delta. Fortunately, any potential problems can be solved by means of the version numbers stored in block pointers and deltas. If the version number in a delta being replayed is greater than the version number in the file manager's block pointer, then the file manager replaces its block pointer with the one in the delta (or deletes its block pointer if the delta is a delete delta). If the delta's version number is less than the version number in the block pointer it means that the block was modified after the delta was created, so the file manager ignores the delta. If the version numbers are equal but one of them has the cleaner flag set, then it is considered to be more recent.

The algorithm described in the previous paragraph is general enough to handle not just crash recovery but all the cases where the file manager must update its block pointers. In fact this algorithm is the only one the file manager uses. It is used for normal client writes, deletes, and cleaning, in addition to crash recovery.

The third problem is that a client could have created a new file and written blocks for that file to a stripe without yet informing the file manager of the new file's existence. If this happens then the server might discover that no file exists for a delta it is replaying. This same inconsistency can happen during recovery in LFS, and Zebra uses the same solution as LFS: it creates the file in a special "lost and found" directory and adds the block to that file. This allows the user to reclaim the information from the lost and found directory if it is important.

7.3 Stripes vs. Cleaner State

In order for the stripe cleaner to recover from a crash without completely reprocessing all of the stripes in the file system, it must occasionally checkpoint its state to disk. The state includes the current utilizations for all of the stripes plus a position in each client log, which identifies the last delta processed by the stripe cleaner. Any buffered data for the stripe files must be flushed before writing the checkpoint.

When the stripe cleaner restarts after a crash, it reads in the utilizations and stripe identifiers from the most recent checkpoint, then starts processing deltas again at the saved log positions. If a crash occurs after appending deltas to a stripe status file but before writing the next checkpoint, then the status file could end up with duplicate copies of some deltas. Fortunately, the processing of the deltas is idempotent so the duplication is harmless.

8 Prototype Status And Performance

The implementation of the Zebra prototype began in April 1992. As of March 1993 all of the major components are operational: Zebra supports all of the usual UNIX file operations and the cleaner works. However, the prototype does not yet write out parity

with stripes so it can't reconstruct data during server outages. In addition it doesn't implement most of the crash recovery features described above and it only supports sequential write-sharing (consistency is not guaranteed if a file is accessed concurrently by readers and writers).

The rest of this section contains some preliminary performance measurements made with the prototype implementation. The measurements show that Zebra provides a factor of 5-8 improvement in throughput for large reads and writes relative to either NFS or the Sprite file system, but its lack of name caching prevents it from providing much of a performance advantage for small files. We estimate that a Zebra system with name caching would provide substantial performance improvements for small writes also.

For our measurements we used a cluster of DECstation-5000 Model 200 workstations connected by an FDDI ring (maximum bandwidth 100 Mbits/sec.). The workstations are rated at about 20 integer SPECmarks. For our measurements the memory bandwidth is at least as important as CPU speed; these workstations can copy large blocks of data from memory to memory at about 12 Mbytes/sec. but copies to or from disk controllers and FDDI interfaces run at only about 8 Mbytes/sec. Each storage server is equipped with a single RZ57 disk with a capacity of about 1 Gbyte and an average seek time of 15 ms. The disks transfer large blocks of data at about 2 Mbytes/sec. but the SCSI bus and controller can only sustain about 1.7 Mbytes/sec.

For comparison we also measured a standard Sprite configuration and an Ultrix/NFS configuration. The Sprite system used the normal Sprite network protocols with a log-structured file system as the disk storage manager. Its hardware was the same as that used for Zebra. The NFS configuration had a slightly faster server CPU and slightly faster disks. The NFS server included a 1-Mbyte PrestoServe card for buffering disk writes.

The first benchmark consisted of an application that writes a single very large file (12 Mbytes) and then invokes `fsync` to force the file to disk. We ran one or more instances of this application on different clients (each writing a different file) with varying numbers of servers, and computed the total throughput of the system (total number of bytes written divided by elapsed time). Figure 6 graphs the results.

Even with a single client and server, Zebra runs at about twice the speed of either NFS or Sprite. This is because Zebra uses very large blocks and its asynchronous RPC allows it to overlap disk operations with network transfers. The limiting factor in this case is the server's disk system, which can only write data at about 1.1 Mbyte/sec. As servers are added in the single-client case Zebra's performance increases by more than a factor of two to 2.5 Mbytes/sec. with four servers. The non-linear performance in Figure 6 occurs because programmed I/O must be used to copy output packets to the FDDI interface. As a result, the transfer of the data from the application into the kernel's file cache is not overlapped with the transfer of stripes to the servers; in the 4-server case about 25% of the elapsed time is spent in this sequential portion. The raw speed at which Zebra can write stripes from its file cache to the storage servers (ignoring start-up effects) scales nearly linearly from 1.1 Mbytes/sec. with one server to 3.9 Mbytes/sec. with four servers, at which point the client's FDDI interface saturates. Performance with two or more clients is limited entirely by the servers, so it scales linearly with the number of servers.

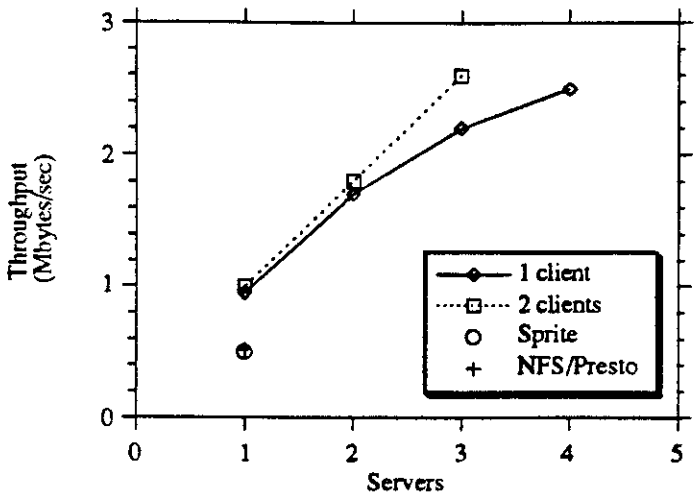

Figure 6. Throughput for large file writes. Each client ran a single application that wrote a 12-Mbyte file and then flushed the file to disk. In multi-server configurations data was striped across all the servers with a fragment size of 512 Kbytes.

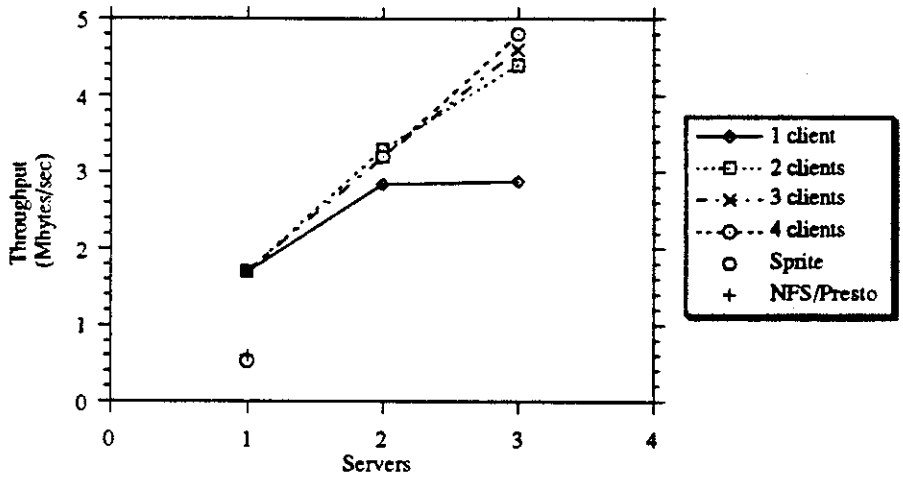

Figure 7. Throughput for large file reads. Each client ran a single application that read a 12-Mbyte file. In multi-server configurations data was striped across all the servers with a fragment size of 512 Kbytes.

Figure 7 shows Zebra's throughput for reading large files. Zebra's performance for reading is better than for writing because the servers can read data from their disks at the full SCSI bandwidth of 1.7 Mbytes/sec. Thus a single client can read files at 1.7 Mbytes/sec. from a single server, and three clients can achieve a total bandwidth of almost 5 Mbytes/sec. with three servers. Two servers can saturate a single client, however, causing the single client curve in Figure 7 to level off at 2.9 Mbytes/sec. At that speed the client is spending all of its time copying data between the application, the file cache, and the network, and in managing the Zebra file's metadata. This overhead could be reduced significantly by modifying the Sprite kernel use the FDDI interface's DMA capability to transfer incoming network packets directly into the file cache, rather than into an intermediate network buffer, and by optimizing the metadata manipulation routines.

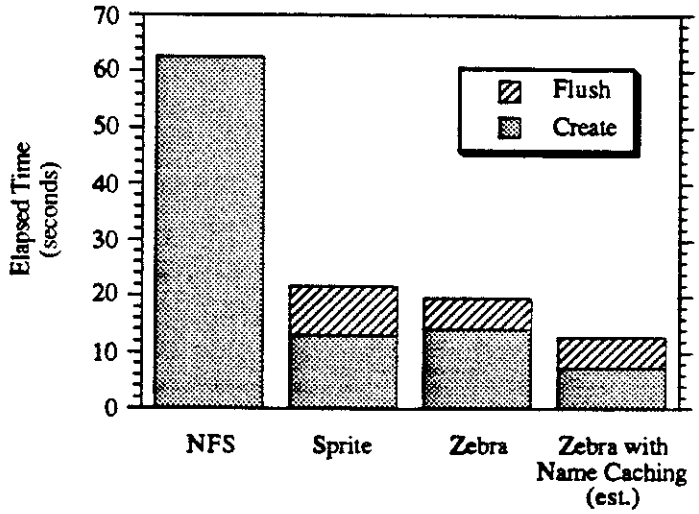

Figure 8. Performance for small writes. A single client created 2048 files, each 1 Kbyte in length, then flushed all the files to a single server. The elapsed time is divided into two components: the times to open and close the files and write the data, and the time to flush the data to disk. For NFS, each file was flushed as it was closed.

Figure 8 shows the elapsed time to write small files with a single client and single server. Although Zebra is substantially faster than NFS for this benchmark, it is only slightly faster than Sprite. The main reason for this is that neither Zebra nor Sprite caches naming information; each open and close requires a separate message exchange with the file server, and most of the time is spent in these exchanges. The rightmost bar in the figure estimates the Zebra time if name caching were implemented; this estimation is derived from the performance of running the same benchmark directly on a Sprite file server. Zebra is somewhat faster than Sprite in the second phase of the benchmark where the new files are flushed to disk. Both systems merge the small files into large blocks for writing, but Sprite doesn't do it until the data has reached the server: each file is transferred over the network in a separate message exchange. Zebra batches the files together before transferring over the network, which is more efficient.

9 Related Work

Most of the key ideas in Zebra were derived from prior work in disk arrays and log-structured file systems. However, there are many other related projects in the areas of striping and availability.

RAID-II [Lee92] and DataMesh [Wilkes90] both use RAID technology to build a high-performance file server. RAID-II uses a dedicated high-bandwidth data path between the network and the disk array to bypass the slow memory system of the server host. DataMesh is an array of processor/disk nodes connected by a high-performance interconnect, much like a parallel machine with a disk on each node. In both of these systems the striping is internal to the server, whereas in Zebra the clients participate in striping files.

Several other striping file systems have been built over the last several years. Some, such as HPFS [Poston88] stripe across local disks; others, such as CFS [Pierce89] and

Bridge [Dibble90] stripe across I/O nodes in a parallel computer; and a few, such as Swift [Cabrera91] stripe across servers in a network file system. All of these systems use per-file striping, so they work best with large files. As far as we know, none of these systems uses parity in its striping mechanisms; Cabrera mentions the possibility of incorporating parity into Swift but does not resolve the potential problems with small files and atomic parity updates. In fact, it was the apparent difficulty of using the Swift design with small files and parity that led us to the design for Zebra.

There have also been several recent research efforts to improve the availability of network file systems, such as Locus [Walker83], Coda [Satyanarayanan90], Deceit [Siegel90], Ficus [Guy90] and Harp [Liskov91]. All of these systems replicate data by storing complete copies, which has higher storage and update costs than Zebra's parity scheme. Harp uses write-behind logs with uninterruptible power supplies to avoid synchronous disk operations and thereby reduce the update overhead. In addition, some of the systems, such as Locus and Coda, use the replicas to improve performance by allowing a client to access the nearest replica; Zebra's parity approach does not permit this optimization.

The create and delete deltas used by Zebra are very similar to the active and deleted sublists used in the Grapevine mail system to manage entries in a registration database [Birrell82]. Grapevine used timestamps whereas Zebra uses version numbers, but they each allow the system to establish an order between different sources of information and to recover from crashes.

10 Conclusion

Zebra takes two ideas that were originally developed for managing disk subsystems, striping with parity and log-structured file systems, and applies them to network file systems. The result is a network file system with several attractive properties:

Performance. Data can be read and written at rates much higher than any single disk or server could support. The high bandwidth is available not just for reading and writing large files, but also for writing small files and potentially even for reading small files under certain conditions.

Scalability. New disks or servers can be added incrementally to increase the system's bandwidth and capacity. Zebra's stripe cleaner automatically reorganizes data over time to take advantage of the additional bandwidth.

Cost-effective servers. Storage servers do not need to be high-performance machines or have special-purpose hardware, since the performance of the system can be increased by adding more servers. Zebra transfers information to storage servers in large stripe fragments and servers need not interpret the contents of the stripes, so the server implementation is simple and has low overheads.

Availability. By combining ideas from RAID and LFS, Zebra can use simple mechanisms to manage parity for each stripe. The system can continue operation while one of the storage servers is unavailable and can reconstruct lost data in the event of a total failure of a server or disk.

Simplicity. Zebra adds very little complexity over the mechanisms already present in a network file system that uses logging for its disk structures. The deltas provide a simple way to maintain consistency among the components of the system.

Zebra makes it possible to choose among a variety of system configurations to optimize performance, cost, and availability. We think the best configuration is probably one with several inexpensive off-the-shelf workstations as storage servers, each with as many disks as its memory system and I/O interfaces can support. If a single high-performance file server is used instead, it still makes sense to use Zebra because it allows the clients to stripe across the server's disks. Finally, Zebra makes it possible to stripe data across several high-performance file servers, providing truly massive bandwidth to satisfy the I/O demands of even the fastest supercomputers.

11 References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff and John K. Ousterhout, "Measurements of a Distributed File System", *Proc. of the 13th Symp. on Operating Systems Principles*, October 1991, 198-212.

[Bernstein81] Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, June 1981, 185-222.

[Birrell82] Andrew D. Birrell, Roy Levin, Roger M. Needham and Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing", *Comm. of the ACM 25*, 4 (April 1982), 260-274.

[Cabrera91] Luis-Felipe Cabrera and Darrell D. E. Long, "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates", *Computing Systems 4*, 4 (Fall 1991), 405-436.

[Chen90] Peter M. Chen and David A. Patterson, "Maximizing Performance in a Striped Disk Array", *Annual Int. Symp. of Computer Architecture*, May 1990, 322-331.

[Chutani92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason and Robert N. Sidebotham, "The Episode File System", *Proc. of the Winter 1992 USENIX Conf.*, Jan. 1992, 43-60.

[Dibble88] Peter C. Dibble, Michael L. Scott and Carla Schlatter Ellis, "Bridge: A High- Performance File System for Parallel Processors", *Proc. of the 8th Int. Conf. on Distributed Computing Systems*, 1988, 154-161.

[Floyd89] R. Floyd and C. Ellis, "Directory Reference Patterns in Hierarchical File systems", *IEEE Trans. on Knowledge and Data Engineering 1*, 2 (June 1989), 238-247.

[Guy90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek and Dieter Rothmeier, "Implementation of the Ficus Replicated File System", *Proc. of the Summer 1990 USENIX Conf.*, June 1990, 63-71.

[Hagmann87] Robert Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", *Proc. of the 11th Symp. on Operating Systems Principles*, Nov. 1987, 155-162.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West, "Scale and Performance in a Distributed File System", *ACM Trans. on Computer Systems 6*, 1 (Feb. 1988), 51-81.

[Lee92] Edward K. Lee, Peter M. Chen, John H. Hartman, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson and David A. Patterson, "RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service", Technical Report UCB/CSD 92/672, UCBCS, Feb. 1992.

[Liskov91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira and Michael Williams, "Replication in the Harp File System", *Proc. of the 13th Symp. on Operating Systems Principles*, Oct. 1991, 226-238.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry, "A Fast File System for Unix", *ACM Trans. on Computer Systems 2*, 3 (Aug. 1984), 181-197.

[Nelson88] Michael N. Nelson, Brent B. Welch and John K. Ousterhout, "Caching in the Sprite Network File System", *ACM Trans. on Computer Systems 6*, 1 (Feb. 1988), 134-154.

[Ousterhout88] John Ousterhout, Andrew Cherenson, Fred Douglis, Mike Nelson and Brent Welch, "The Sprite Network Operating System", *IEEE Computer 21*, 2 (Feb. 1988), 23-36

[Patterson88] David A. Patterson, Garth Gibson and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. of the 1988 ACM Conf. on Management of Data*, June 1988, 109-116.

[Pierce89] Paul Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem", *Proc. of the Fourth Conf. on Hypercubes*, March 1989.

[Poston88] Alan Poston, "A High Performance File System for UNIX", June 1988.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proc. of the 13th Symp. on Operating Systems Principles*, October 1991, 1-15.

[Satyanarayanan90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, E. H. Siegel and D. C. Steere, "Coda: a highly available file system for a distributed workstation environment", *IEEE Trans. on Computers 39*, 4 (April 1990), 447-459.

[Seltzer93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick and Carl Staelin, "An Implementation of a Log-Structured File System for UNIX", *Proc. of the Winter 1993 USENIX Conf.*, Jan. 1993, 307-326.

[Shirriff92] Ken Shirriff and John Ousterhout, "A Trace-driven Analysis of Name and Attribute Caching in a Distributed File System", *Proc. of the Winter 1992 USENIX Conf.*, Jan. 1992, 315-331.

[Siegel90] Alex Siegel, Kenneth Birman and Keith Marzullo, "Deceit: A Flexible Distributed File System", *Proc. of the Summer 1990 USENIX Conf.*, June 1990, 51-61.

[Walker83] Bruce Walker, Gerald Popek, Robert English, Charles Kline and Greg Thiel, "The LOCUS Distributed Operating System", *Proc. of the 9th Symp. Operating Systems Principles*, Nov. 1983, 49-70.

[Wilkes91] John Wilkes, "DataMesh -- Parallel Storage Systems for the 1990s", *Digest of Papers, Eleventh IEEE Symp. on Mass Storage Systems*, Oct. 1991, 131-136.