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 increases throughput by striping file data across multiple servers. Rather than striping each file separately, Zebra forms all the new data from each client into a single stream, which it then stripes using an approach similar to a log-structured file system. This provides high performance for writes of small files as well as for reads and writes of large 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 4-5 times the throughput of the standard Sprite file system or NFS for large files and a 20%-3x improvement for writing small files.

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 parity information in each stripe, 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 of a single server, including its memory bandwidth and the speed of its processor, network interface, I/O busses, and disks. It is possible to split a file system among multiple servers but 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 lie on a single server, making that server a hot spot.

In the future, new styles of computing such as multi-media and parallel computation are likely to demand much greater throughput than today's applications, making the limitations of a single server even more severe. 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 easily exceed the bandwidth of a file server if they all run video applications simultaneously, and the problems will become much worse when video resolution increases with the arrival of HDTV. Another example is parallel applications. Several 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 traditional supercomputers, which cannot be handled by today's network file servers.

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 borrowed from log-structured file systems (LFS) [Rosenblum91]. Each client forms its new data for all files into a sequential log that it stripes across the storage servers. This allows even small files to benefit from striping. It also reduces network overhead, simplifies the storage servers, and spreads write traffic uniformly across the servers.

Zebra's style of striping also makes 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, allowing the stripe's data to be reconstructed in the event of a disk or server failure. Zebra can continue operation while a server is unavailable. Even if a disk is totally destroyed Zebra can reconstruct the lost data.

We have constructed a prototype implementation of Zebra as part of the Sprite operating system [Ousterhout88].

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.

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 achieves up to 4 Mbytes/second for a single client with four servers, which is 4-5 times the throughput of either NFS or the standard Sprite file system. For small files the Zebra prototype improves performance by more than a factor of 3 over NFS. The improvement over Sprite is only about 20%, however. This is because both Zebra and Sprite require the client to notify the file server of file opens and closes, and when writing small files these notifications dominate the running time. With the addition of file name caching to both systems we would expect Zebra to have even more of an advantage over Sprite.

The rest of the paper is organized as follows. Section 2 describes the RAID and log-structured-file-system technologies used in Zebra and introduces Zebra's logging approach. Section 3 describes the structure of Zebra, which consists of clients, storage servers, a file manager, and a stripe cleaner. Section 4 shows how the components of the system work together in normal operation; communication between the components is based on *deltas*, which describe block creations, updates, and deletions. Section 5 describes how Zebra restores consistency to its data structures after crashes, and Section 6 shows how the system provides service even while components are down. Section 7 gives the status of the Zebra prototype and presents some performance measurements. Section 8 discusses related work and Section 9 concludes.

2 Striping in Zebra

Zebra distributes file data over several file servers while ensuring that the loss of a single server does not affect the availability of the data. To do this Zebra borrows from two recent innovations in the management of disk storage systems: RAID technology (Redundant Arrays of Inexpensive Disks) [Patterson88], and log-structured file systems (LFS) [Rosenblum91]. RAID technology allows Zebra to provide scalable file access performance while using parity instead of redundant copies to guard against server failures. The log-structured approach simplifies the parity implementation, reduces the impact of managing and storing parity, and allows clients to batch together small writes to improve server efficiency.

2.1 RAID

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 and fast 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 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 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 larger than 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 up to eight disks, each with a bandwidth of 1-2 Mbytes/second, but the SCSI bus has a total bandwidth of only 2-10 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 the Zebra prototype these copies can only proceed at about 6-8 Mbytes/second. The

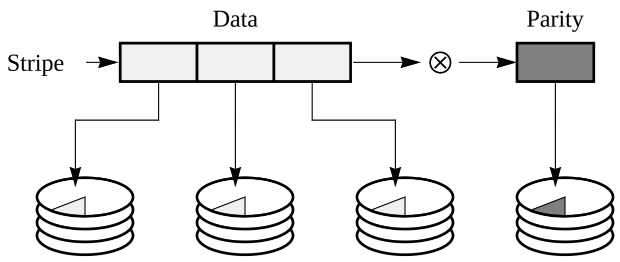

Figure 1. **Striping with parity**. The storage space of a RAID disk array is divided into stripes, where each stripe contains a striping unit on each disk of the array. All but one of the striping units hold data; the other striping unit holds parity information that can be used to recover after a disk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

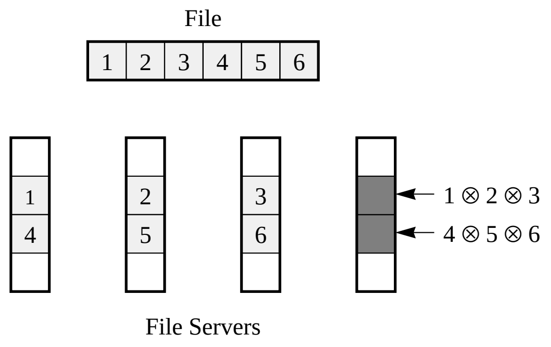

Figure 2. **Per-file striping for a large file**. The file is divided up into stripe units that are distributed among the servers. Each stripe contains one parity fragment.

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.

In order to eliminate the performance bottlenecks multiple paths must exist between the source or sink of data and the disks so that different paths can be 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.

2.2 Per-File Striping in a Network File System

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*.

The most obvious 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 because each stripe contains data from only one file. While conceptually simple, per-file striping has two 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 files differently than 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

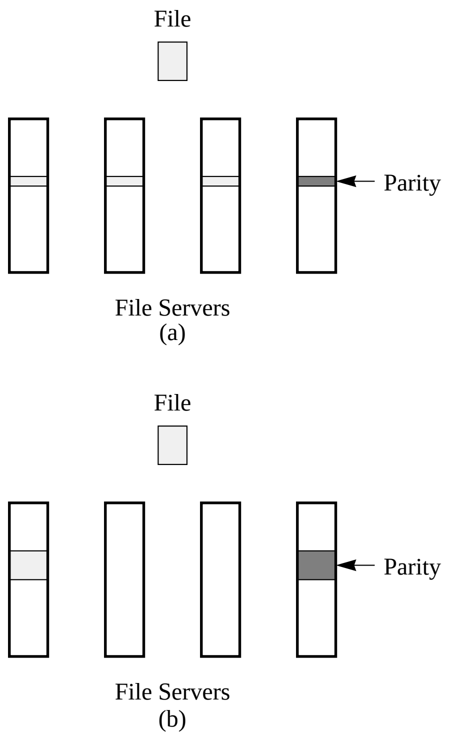

Figure 3. **Per-file striping for a small file**. In (a) the file is striped evenly across the servers, resulting in small fragments on each server. In (b) the entire file is placed on one server but the parity takes as much space as the file.

single server then its parity will consume as much space as the file itself, resulting in high storage overhead. In addition, the approach in Figure 3(b) can result in unbalanced disk utilization and server loading.

Second, 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.

2.3 Log-Structured File Systems and Per-Client Striping

Zebra solves the problems with per-file striping by applying techniques from log-structured file systems (LFS) [Rosenblum91]. LFS is a disk management technique that treats the disk like an append-only log. 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

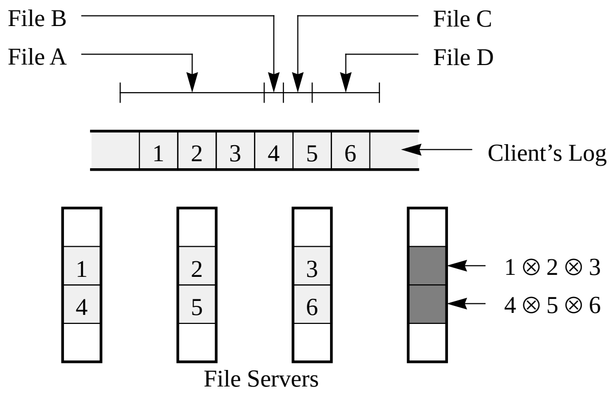

Figure 4. **Per-client striping in Zebra**. Each client forms its new file data into a single append-only log and stripes this log across the servers. In this example file A spans several servers while file B is stored entirely on a single server. Parity is computed for the log, not for individual files.

sequential transfers. LFS is particularly effective for writing small files, since it can write many files in a single transfer; in contrast, traditional file systems require at least two independent disk transfers for each file. Rosenblum reported a tenfold speedup over traditional file systems for writing small files. LFS is also well-suited for RAIDs because it batches small writes together into large sequential transfers and avoids the expensive parity updates associated with small random writes.

Zebra can be thought of as a log-structured network file system: whereas LFS uses the logging approach at the interface between a file server and its disks, Zebra uses the logging approach at the interface between a client and its servers. Figure 4 illustrates this approach, which we call *per-client* striping. Each Zebra client organizes its new file data into an append-only log, which it then stripes across the 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, and small writes are batched together by the log mechanism and written to the servers in large transfers; no special handling is needed for either case. 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 overhead because parity is not computed on a per-file basis. Furthermore, parity never needs to be updated because file data are never overwritten in place.

The above introduction to per-client striping leaves some unanswered questions. For example, how can files be shared between client workstations if each client is writing its own log? Zebra solves this problem by introducing a central *file manager*, separate from the storage servers, that manages metadata such as directories and file attributes and supervises interactions between clients. Also, how is free

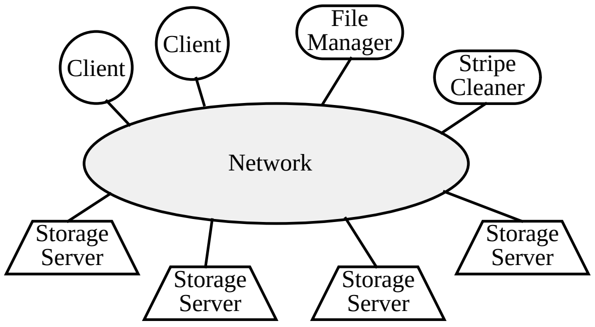

Figure 5: Zebra schematic. Clients run applications; storage servers store data. The file manager and the stripe cleaner can run on any machine in the system, although it is likely that one machine will run both of them. A storage server may also be a client.

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 section provides a more detailed discussion of these issues and several others.

3 Zebra 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. 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 4 then shows how the components work together to implement operations such as reading and writing files, and Sections 5 and 6 describe how Zebra deals with crashes.

We will describe Zebra under the assumption that there are several storage servers, each with a single disk. However, this need not be the case. For example, storage servers could each contain several disks managed as a RAID, thereby giving the appearance to clients of a single disk with higher capacity and throughput. It is also 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.

3.1 Clients

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 the storage servers, and return them to the application. As will be seen below, the file manager keeps track of where file data are stored and provides this information to clients when needed. When an application writes a file the client appends the new data to its log by creating new stripes to hold the data, computing the parity of the stripes, and writing the stripes to the storage servers.

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

3.2 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. The identifier for a fragment consists of an identifier for the client that wrote the fragment, a sequence number that identifies the stripe uniquely among all those written by the client, and an offset for the fragment within its stripe. All fragments in Zebra are the same size, which should be chosen large enough to make network and disk transfers efficient. In the Zebra prototype we use 512-Kbyte fragments.

Storage servers provide five operations:

Store a fragment. This operation allocates space for the fragment, writes the fragment to disk, and records the fragment identifier and disk location for use in subsequent accesses. The operation is synchronous: it does not complete until the fragment is safely on disk. The fragment must not already exist unless it is a parity fragment, in which case the new copy of the fragment replaces the old. This is done in a non-overwrite manner to avoid corruption in the event of a crash.

Append to an existing fragment. This operation is similar to storing a 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 (this can happen, for example, if an application invokes the `fsync` system call to force data to disk). Appends are implemented atomically so that a crash during an append cannot cause the previous contents of the fragment 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; a fragment identifier, 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 ends of the clients' logs after a crash.

Stripes are immutable once they are complete. A stripe may be created with a sequence of append operations, but non-parity fragments are never overwritten and once the stripe is complete it is never modified except to delete the entire stripe. A parity fragment, however, can be overwritten if data are appended to a partial stripe (see Section 4.2).

3.3 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 are stored, 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.

In the Zebra prototype we implemented the file manager using a Sprite file server with a log-structured file system. For each Zebra file there is one file in the file manager's file system, and the "data" in this file are an array of block pointers that indicate where the blocks of data for the Zebra file are stored. This allows Zebra to use almost all of the existing Sprite network file protocols without modification. Clients open, read, and cache Zebra metadata in the same manner that they cache "regular" Sprite files. 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.

The performance of the file manager is a concern because it is a centralized resource. In our implementation clients must contact the file manager on each open and close, so communication with the file manager is a performance bottleneck when clients are accessing small files. We believe that this problem can be solved by caching naming information on clients so that the file manager need not be contacted for most opens and closes. Client-level name caching has been used successfully in the AFS file system [Howard88] and Shirriff found that a name cache occupying only 40 Kbytes of a client's memory can produce a hit rate of 97% [Shirriff92]. 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.

The centralized nature of the file manager also makes its reliability a concern; this issue is addressed in Section 6.

3.4 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 are deleted or because the blocks are 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. 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 servers 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 first identifies stripes with large amounts of free space, then it reads the remaining live blocks out of the stripes 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. Section 4.5 describes the cleaning algorithm in more detail.

4 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.

4.1 Communication via Deltas

A client's log 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 changes to the blocks in a file, and are used to communicate these changes between the clients, the file manager, and the stripe cleaner. For example, a client puts a delta into its log when it writes a file block, and the file manager subsequently reads the delta to update the metadata for that block. Deltas contain the following information:

File identifier: a unique identifier for a file, analogous to an i-number in UNIX file systems.

File version: identifies the point in time when the change described by the delta occurred. A file's version number increments whenever a block in the file is written or deleted. The version numbers allow deltas in different logs to be ordered during crash recovery.

Block number: identifies a particular block by its position within the file.

Old block pointer: gives the fragment identifier and offset of the block's old storage location. If this delta is for a new block then the old block pointer has a special null value.

New block pointer: gives the fragment identifier and offset for the block's new storage location. If this delta is for a block deletion then the new block pointer has a special null value.

Deltas are created whenever blocks are added to a file, deleted from a file, or overwritten. All of these are called *update deltas*. Deltas are also created by the stripe cleaner when it copies live blocks out of stripes; these are called *cleaner deltas*. Lastly, *reject deltas* are created by the file manager to resolve races between stripe cleaning and file updates. All of these deltas will be described in more detail in the rest of the paper.

Deltas provide a simple and reliable way for the various system components to communicate changes to files. Since deltas are stored in the client logs and the logs are reliable, each component can be sure that any delta it writes will not be lost. When a client modifies a block of a file it only needs to write the block and the update delta to the log to ensure that both the file manager and the stripe cleaner learn of the modification. After crashes the file manager and stripe cleaner replay deltas from the client logs to recover their state.

4.2 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 they are placed in the client's file cache. The dirty data aren't written to a server until either (a) they reach 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 data be written to disk, or (d) the file manager requests that 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 need to be written at all [Baker91].

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. For each file block written the client also puts an update delta into its log and increments the file's version number. In the Zebra prototype file deletion and truncation are handled by the file manager, so it generates deltas for these operations and increments the file version numbers appropriately. In a system with name caching the deltas for deletion and truncation would be generated by clients.

To benefit from the multiple storage servers it is important for a client to transfer fragments to all of the storage servers concurrently. We added support for asynchronous remote procedure calls to Sprite to allow clients to do this. A client can also transfer the next stripe fragment to a storage server while the server is writing the current stripe fragment to disk, so that both the network and the disk are kept busy. The client computes the parity as it writes the fragments and at the end of each stripe the client writes the parity to complete the stripe. In the Zebra prototype the client also sends the stripe's deltas to the file manager and stripe cleaner. This improves performance by avoiding disk accesses for the file manager and stripe cleaner to read the deltas from the log, but it isn't necessary for correct operation. If the client crashes 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 invokes `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 be lost only if both a disk is destroyed and the client crashes). For even greater protection 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 always available after a crash.

The rate at which applications invoke `fsync` will have a large impact on Zebra's performance (or any other file system's) because `fsync`'s require synchronous disk operations. Baker et. al [Baker92b] found that under a transaction processing workload up to 90% of the segments written on an LFS file system were partial segments caused by an `fsync.`Such a workload would have poor performance on Zebra as well. Fortunately, they found that on non-transaction processing workloads `fsync` accounted for less than 20% of the segments written.

4.3 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 in the Zebra prototype. First the client must fetch the block pointers from the file manager, then it reads the file data from the storage servers. 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. In the prototype this extra RPC takes 2 ms if the file manager has the block pointers cached, and 19.5 ms otherwise. As many as 2048 block pointers can be returned by the RPC, allowing all of the block pointers for files up to 8 Mbytes in size to be fetched in a single RPC. Zebra clients cache both block pointers and data, so this information is only fetched on the first access to a file; name caching would eliminate most of the open and close RPCs as well.

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: 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 small files by reading 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, this approach might improve read performance. We speculate that such locality exists but we have not attempted to verify its existence or capitalize on it in Zebra.

4.4 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, a client could write a file that is cached on another client; if the second client subsequently reads the file, it must discard its stale cached data and fetch the new data. We chose to use the Sprite approach to consistency, which involves flushing or disabling caches 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 then 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.

4.5 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 delta the cleaner increments the utilization of the stripe containing the new block (if any), and decrements the utilization of the stripe that contained the old block (if any). 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 Zebra files. Note that a single update or cleaner delta can affect two different stripes; a copy of the delta is appended to the status files for both stripes.

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 blocks that haven't yet been deleted. Without the stripe status files this step would be much more difficult, since the deltas that cause blocks to become free could be spread throughout the stripes in the file system.

Once the live blocks have been identified the stripe cleaner, which executes as a user-level process, copies them to a new stripe using a special kernel call. The kernel call reads one or more blocks from storage servers, appends them to its client log, and writes the new log contents to the storage servers. For each block a cleaner delta is included in the client's log. The kernel call for cleaning blocks has the same effect as 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 last-modified times or version numbers for files, and (d) the deltas that it generates are marked as cleaner deltas instead of update deltas.

One concern about the stripe cleaner is how much of the system's resources it will consume in copying blocks. We do not have measurements of Zebra under real workloads, but we expect the cleaning costs for Zebra to be comparable to those for other log-structured file systems. In a transaction-processing benchmark on a nearly full disk Seltzer found that cleaning accounted for 60-80% of all write traffic and significantly affected system throughput [Seltzer93]. However, in a software development benchmark that is more typical of workstation workloads Seltzer found cleaning costs to be negligible. Rosenblum measured production usage of LFS on Sprite for several months and found that only 2-7% of the data in stripes that were cleaned were live and needed to be copied [Rosenblum91]. Based on these measurements we believe that the cleaning overhead will be low for typical workstation workloads but more work is needed to reduce the overheads for transaction-processing workloads.

4.6 Conflicts Between Cleaning and File Access

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 could modify the block after the cleaner reads the old copy but before the cleaner rewrites the block, in which case the new data would 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 files to prevent them from being modified until after cleaning was finished. Unfortunately, this produced lock convoys that effectively halted all normal file accesses during cleaning and resulted in significant pauses.

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. Instead the stripe cleaner just copies the block and issues a cleaner delta, assuming optimistically that its information about the block is correct and the block hasn't been updated recently. If in fact the block was updated while the cleaner was cleaning it, an update delta will be generated by the client that made the change. Regardless of the order in which these deltas arrive at the file manager, the file manager makes sure that the final pointer for the block reflects the update delta, not the cleaner delta. This approach results in wasted work by the cleaner in the unusual case where a conflict occurs, but it avoids synchronization in the common case where there is no conflict.

Type of Delta	**Block Pointer Matches?**	**Update Pointer?**	**Issue Reject Delta?**
Update	Yes	Yes	No
Cleaner	Yes	Yes	No
Update	No	Yes	Yes
Cleaner	No	No	Yes

Table 1: File manager delta processing. When a delta arrives at the file manager, the old block pointer in the delta is compared with the current block pointer. If they do not match (the bottom two scenarios) then a conflict has occurred.

The file manager detects conflicts by comparing the old block pointer in each incoming delta with the block pointer stored in the file manager's metadata; if they are different it means that the block was simultaneously cleaned and updated. Table 1 shows the four scenarios that can occur. The first two scenarios represent the cases where there is no conflict: the delta's old block pointer matches the file manager's current block pointer, so the file manager updates its block pointer with the new block pointer in the delta. If an update delta arrives with an old block pointer that doesn't match, it can only mean that the block was cleaned (any other update to the block is prevented by the cache consistency protocol); the file manager updates its block pointer with the new block pointer from the delta. If a cleaner delta arrives with an old block pointer that doesn't match, it means that the block has already been updated so the cleaned copy is irrelevant: the cleaner delta is ignored.

In both of the cases where the file manager detects a conflict it generates a reject delta, which is placed in the client log for its machine. The old block pointer in the reject delta refers to the cleaned copy of the block and the new pointer is null to indicate that this block is now free. The reject delta is used by the stripe cleaner to keep track of stripe usage; without it the stripe cleaner would have no way of knowing that the block generated by the cleaner is unused.

It is also possible for an application to read a block at the same time that it is being cleaned. For example, suppose that a client has retrieved a block pointer from the file manager but the block is moved 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 the pointer and fetches an up-to-date version from the file manager.

4.7 Adding a 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 clients will stripe their logs 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 be 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.

5 Restoring Consistency After Crashes

There are two general issues that Zebra must address when a client or server machine crashes: consistency and availability. If a crash occurs in the middle of an operation then data structures may be left in a partially-modified state after the crash. For example, the file manager might crash before processing all of the deltas written by clients; when it reboots its metadata will not be up-to-date with respect to information in the clients' logs. This section describes how Zebra restores internal consistency to its data structures after crashes. The second issue is availability, which refers to the system's ability to continue operation even while a component is down. Zebra's approach to availability is described in Section 6.

In many respects the consistency issues in Zebra are the same as in other network file systems. For example, the file manager will have to restore consistency to all of its structures on disk. 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 must also recover the information that 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 that follow.

The solutions to all of the consistency issues are based on logging and checkpoints. 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.

The combination of these two 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 the 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.

5.1 Internal Stripe Consistency

When a client crashes it is possible for fragments to be missing from stripes that were in the process of being written. The file manager detects client crashes and recovers on behalf of the client: it queries the storage servers to identify the end of the client's log and verifies that any stripes that could have been affected by the crash are complete. If a stripe is missing a single fragment then the missing data can be reconstructed using the other stripes in the fragment. If a stripe is missing more than one fragment then it is discarded along with any subsequent stripes in the same client's log, effectively truncates the client's log to the last recoverable stripe. This means that data being written at the time of a crash can be lost or partially written, just as in other file systems that maintain UNIX semantics.

When a storage server crashes and recovers, two forms of stripe inconsistency are possible. First, if a stripe fragment was being written at the time of the crash then it might not have been completely written. To detect incomplete stripe fragments, Zebra stores a simple checksum for each fragment. After a storage server reboots it verifies the checksums for fragments written around the time of the crash and discards any that are incomplete.

The second inconsistency after a storage server crash is that it won't contain fragments for new stripes written while it was down. After the storage server reboots it queries other storage servers to find out what new stripes were written. Then it reconstructs the missing fragments as described in Section 6.2 and writes them to disk. The prototype does not yet do this reconstruction.

5.2 Stripes vs. Metadata

The file manager must maintain 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 metadata accordingly. During normal operation the file manager keeps track of its current position in each client's log and at periodic intervals it forces the metadata to disk and writes a checkpoint file that contains the current positions. If a client crashes, the file manager checks with the storage servers to find the end of the client's log and make sure it has processed all of the deltas in the log. If the file manager crashes, then when it reboots it processes all of the deltas that appear in the client logs after the positions stored in the last checkpoint, thereby bringing the metadata up-to-date. A checkpoint is relatively small (a few hundred bytes) since all it contains is current log positions for each client, but it does have a performance impact because the metadata is flushed before it is written. Decreasing the checkpoint interval improves the file manager's recovery time at the expense of normal operation; we anticipate that a checkpoint interval on the order of several minutes will provide acceptable recovery time without significantly affecting the system performance.

There are two complications in replaying deltas, both of which are solved with version numbers. The first complication is that some of the deltas may have already been processed and applied to the metadata. This will happen if the file manager crashes after it writes metadata out to disk but before it writes a new checkpoint. If an update delta is encountered that has already been applied then its version number will be less than that of the file, and it is ignored. As in normal operation, a cleaner delta is applied only if its old block pointer matches the file manager's current block pointer.

The second complication is that a file could have been modified by several different clients, resulting in deltas for the file in several client logs. The file manager must replay the deltas for each file in the same order that they were originally generated. If the file manager encounters a delta during replay whose version number is greater than the file's version number, it means that there are deltas in some other client log that must be replayed first. In this case the file manager must delay the processing of the delta until all the intervening deltas have been processed from the other client logs.

5.3 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 checkpoints its state to disk at regular intervals. 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 are flushed before writing the checkpoint.

When the stripe cleaner restarts after a crash, it reads in the utilizations and log positions, 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. These duplicates are easily weeded out when the cleaner processes the status files.

6 Availability

Our goal for Zebra is for the system to continue to provide service even if some of its machines have crashed. A single failure of either a storage server, the file manager, or the stripe cleaner should not prevent clients from accessing files, neither should any number of client failures affect the remaining clients. Each of the system components is discussed separately in the sections below. The prototype does not yet implement all of these features, as noted.

6.1 Client Crashes

The only way that one client can prevent other clients from accessing files is through the cache consistency protocol: if a client has a file open and cached then other clients' access to the file is restricted to prevent inconsistencies. After a client crash the file manager closes all the open files on the client, thus allowing those files to be cached by other clients.

6.2 Storage Manager Crashes

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, a 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; the storage manager will reconstruct them when it reboots, as described in Section 5.1. In the prototype clients are capable of reconstruction, but only under manual control. Clients do not yet automatically reconstruct fragments when a server crashes.

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.

6.3 File Manager Crashes

The file manager is a critical resource for the entire system because it manages all of the file system metadata. If 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 destroy the file system. We believe that these problems can be eliminated by using the Zebra storage servers to store the file manager's metadata. Instead of using a local disk, the file manager writes the metadata to a virtual disk represented as a Zebra file. Updates to the metadata will be added to the file manager's client log as part of the virtual disk file and striped across the storage servers with parity, just like any other Zebra file. This provides higher performance for the metadata than storing it on a local disk, and also improves its availability and integrity. This approach also allows the file manager to 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 can be restarted on another machine. Of course, if the file manager crashes Zebra will be unavailable until the file manager restarts, but it should be possible to restart the file manager quickly [Baker92a].

We have not yet implemented this approach to improving the file manager's availability and integrity. A similar approach has been proposed by Cabrera and Long for the Swift file system [Cabrera91] for making its storage mediator highly available.

6.4 Stripe Cleaner Crashes

Crashes of the stripe cleaner are relatively easy to handle. The stripe cleaner need not be running in order for Zebra to provide service; all that is needed is for the cleaner to restart before disk space is exhausted. All of the stripe cleaner's state is stored in the Zebra file system, so if the stripe cleaner's machine becomes permanently unavailable the stripe cleaner can be restarted on a different machine.

7 Prototype Status and Performance

The implementation of the Zebra prototype began in April 1992. As of August 1993 Zebra supports all of the usual UNIX file operations, the cleaner is functional, and clients can write parity and reconstruct fragments. The file manager and cleaner both checkpoint their states and are able to recover after a failure. The prototype does not yet implement all of the crash recovery and availability features of Zebra, however. The metadata is not yet stored on the storage servers as described in Section 6.3, clients do not automatically reconstruct stripe fragments when a storage server crashes, storage servers do not reconstruct missing fragments after a crash, and the file manager and stripe cleaner are not automatically restarted. We have simplified the prototype by choosing not to implement name caching or support for concurrent write-sharing.

The rest of this section contains some preliminary performance measurements made with the prototype. The measurements show that Zebra provides a factor of 4-5 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 also provide substantial performance improvements for small writes.

For our measurements we used a cluster of DECstation-5000 Model 200 workstations connected by an FDDI ring (maximum bandwidth 100 Mbits/second). The workstations are rated at about 20 integer SPECmarks and each contained 32 Mbytes of memory. In our benchmarks 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/second but copies to or from disk controllers and FDDI interfaces run at only about 8 Mbytes/second. 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/second, but the SCSI bus and controller can only sustain about 1.6 Mbytes/second.

We had a total of eight workstations available for running these experiments. The minimum configuration we tested consisted of one client, one storage server, and one file manager. In the maximum configuration there were three clients, four storage servers and one file manager. During the measurements the file manager did not generate checkpoints, nor was the stripe cleaner running. Each data point was collected by running the benchmark 10 times and averaging the results.

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 by all clients 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 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/second. As servers are added in the single-client case Zebra's performance increases by more than a factor of 2 to 2.4 Mbytes/second with four servers. The non-linear speedup in Figure 6 occurs because the benchmark runs in two phases: in the first phase the application fills the kernel's file cache by writing the file, and in the second phase the client's kernel flushes its cache by transferring stripes to the servers. These phases are not overlapped and only the second phase benefits from additional storage servers. When we measured the second phase alone we found that the throughput scales nearly linearly from 1.1 Mbytes/second with one server to 3.8 Mbytes/second with four servers, at which point the client's FDDI interface saturates. Performance with two or

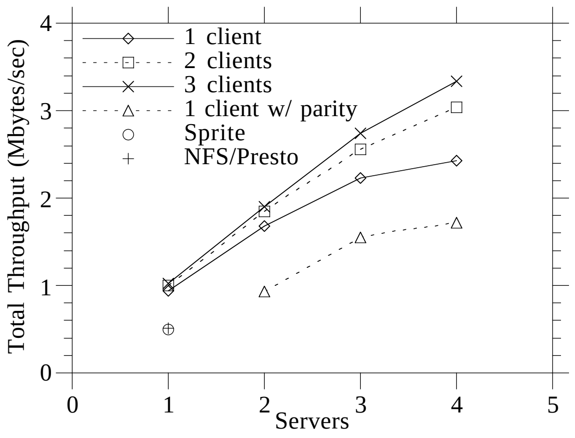

Figure 6. **Total system throughput for large file writes**. Each client ran a single application that wrote a 12-Mbyte file and then flushed the file to disk. In multi-server configurations data were striped across all the servers with a fragment size of 512 Kbytes. Parity was only computed for the line labeled "1 client w/ parity".

more clients is limited entirely by the servers, so it scales linearly with the number of servers.

Figure 6 also shows the throughput for a single client when it generates and writes parity; the throughput is measured in terms of useful file data not including parity. Zebra incurs almost no overhead for parity aside from the obvious overhead of writing more data to more servers. In the best case Zebra's throughput with two servers and parity should be the same as the throughput with one server and no parity, since it is writing one byte of parity for each byte of data; the performance in Figure 6 is only slightly less than this. Ideally, Zebra's throughput with four servers and parity should be the same as the throughput with three servers and no parity. In reality it is somewhat less than this because the client CPU is saturated in the former but not in the latter.

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 1.6 Mbytes/second Thus a single client can read a file at 1.6 Mbytes/second from a single server, and three clients can achieve a total bandwidth of 5.2 Mbytes/second with four servers. Two servers can saturate a single client, however, causing the single client curve in Figure 7 to level off at 2.8 Mbytes/second. At that speed the client is spending most of its time copying data between the application, the file cache, and the network. This overhead could be reduced significantly by modifying the Sprite kernel to use the FDDI interface's DMA capability to transfer incoming network packets directly into the file cache, rather than into an intermediate network buffer.

The performance of reads that require reconstruction is shown in the line labeled "1 client (recon)" in Figure 7. In this test one of the storage servers was unavailable and the client had to reconstruct any stripe fragments stored on that server by reading all of the other fragments in each stripe and computing their parity. With two servers the throughput during reconstruction is only slightly less than in normal operation with a single server; this is because a parity block

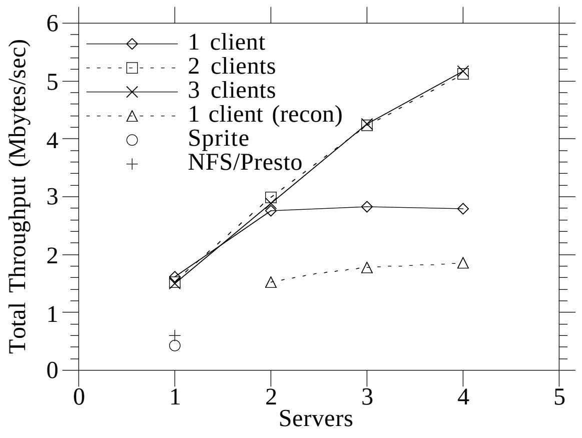

Figure 7. **Throughput for large file reads**. Each client ran a single application that read a 12-Mbyte file. In multi-server configurations data were striped across all the servers with a fragment size of 512 Kbytes. The line labeled "1 client (recon)" shows reconstruction performance: one server was unavailable and the client had to reconstruct the missing stripe fragments. For example, the system represented by the left-most point had two servers, one of which was unavailable.

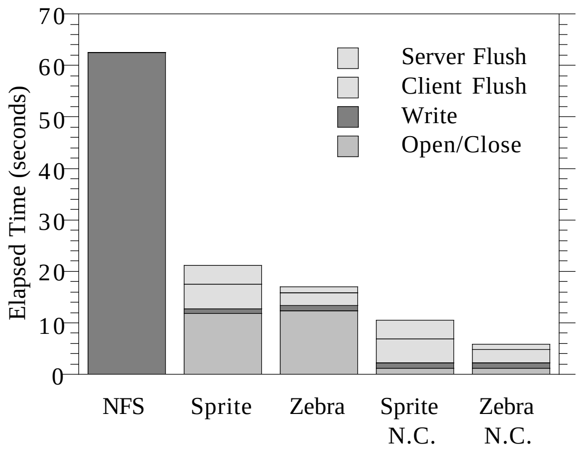

Figure 8. **Performance for small writes**. A single client created 2048 files, each 1 Kbyte in length, then flushed all the files to a single server. The elapsed time is divided into four components: the time to open and close the files, the time for the application to write the data, the time for the client to flush its cache, and the time for the server to flush its cache to disk. For NFS, each file was flushed as it was closed. The two rightmost bars are estimates for Sprite and Zebra if name caching were implemented.

in a two server system is a mirror image of its data block and therefore reconstruction doesn't require any additional computation by the client. The throughput doesn't increase much with additional servers because the client CPU has saturated due to additional copying and exclusive-or operations to reconstruct the missing data.

Figure 8 shows the elapsed time for a single client to write small files. In the NFS and Sprite tests the client was writing to a single file server, while the Zebra test used one storage server and one file manager. Although Zebra is substantially faster than NFS for this benchmark, it is only

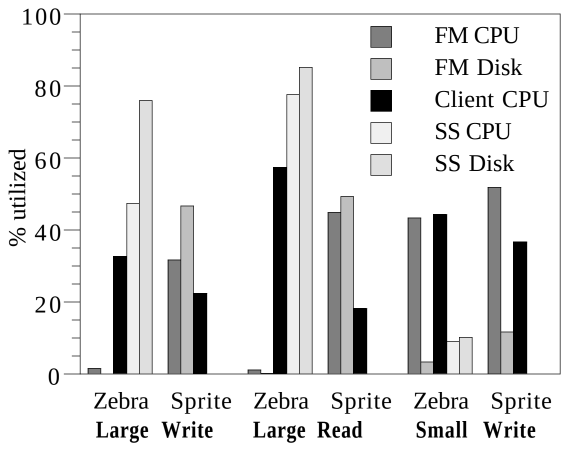

Figure 9: **Resource utilizations**. Utilizations of the file manager (FM) CPU and disk, client CPU, and storage server (SS) CPU and disk during the previous three benchmarks. The Zebra system consisted of a single client, a single file manager, and a single storage server; the Sprite system consisted of a single client and a single file server, which serves as both file manager and storage server. Parity was not computed.

about 20% faster than Sprite. The main reason for this is that neither Zebra nor Sprite caches naming information; each open and close requires a separate RPC to either the file server or file manager, and the figure shows that most of the time is spent in these RPCs. The rightmost bars in the figure estimate the times for Sprite and Zebra if name caching were implemented; the estimates were made by running the same benchmark directly on a Sprite file server. Zebra is significantly faster than Sprite during the cache-flush portion of the benchmark. Both systems merge the small files into large blocks for writing, but Sprite doesn't do it until the data have 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.

Figure 9 shows the utilizations of various system components during the benchmarks, both for Zebra and for Sprite. For large reads and writes the Zebra file manager's CPU and disk are almost idle; the system could scale to dozens of storage servers before the file manager becomes a performance bottleneck. In comparison to Sprite, Zebra has higher utilizations of the client CPU, server CPU, and server disk; this causes Zebra to complete the benchmark faster.

For small writes both Zebra and Sprite spend most of their time in synchronous RPCs to open and close files. In both systems the sum of client CPU utilization and file manager CPU utilization is nearly 100%; it cannot exceed 100% because the RPCs do not allow much overlap in processing between the two CPUs. In both Zebra and Sprite it appears that the server CPU will saturate with the addition of a second client; without name caching the server CPU will be a performance bottleneck.

8 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], DataMesh [Wilkes92], and TickerTAIP [Cao93] all use RAID technology to build high-performance file servers. 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. TickerTAIP is a refinement of DataMesh that focuses on distributing the functions of the traditionally centralized RAID controller across multiple processors, thus removing the controller as a single point of failure. In all of these systems the striping is internal to the server, whereas in Zebra the clients participate in striping files.

RADD (Redundant Array of Distributed Disks) [Schloss90] is similar to RAID in that it uses parity to withstand the loss of a disk, but it differs by separating the disks geographically to decrease the likelihood of losing multiple disks. Furthermore, RADD does not stripe data; the data stored on each disk are logically independent, thus RADD does not improve the performance of individual data accesses.

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 sfs [LoVerso93] and Bridge [Dibble90] stripe across I/O nodes in a parallel computer; but to our knowledge only one, Swift [Cabrera91], stripes across servers in a network file system. All of these systems use per-file striping, so they work best with large files. Swift's performance while reading and writing large files improves nearly linearly as the number of servers increases to three, but the CPUs and disks for Swift are much slower than those for Zebra so its absolute performance is lower than Zebra's. A per-file parity mechanism is planned for Swift, although it does not appear to resolve the potential problems with small files and atomic parity updates. The implementation of this mechanism is currently in progress and performance measurements should be forthcoming.

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 results in 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.

Another approach to highly available file service is to design file servers that can quickly reboot after a software failure [Baker92a]. The idea is to reboot the file server so quickly that file service is not interrupted. This alternative does not require redundant copies or parity, but neither does it allow the system to continue operation in the event of a hardware failure.

Zebra borrows its log structure from LFS [Rosenblum91], a high-performance write-optimized file system. A recent paper by Seltzer et. al [Seltzer93] has shown that adding extents to FFS [McKusick84] results in a file system (called EFS) that has comparable performance to LFS on large reads and writes. However, EFS does not improve performance for small files as does LFS and therefore Zebra, nor does it address the parity and striping issues presented by a striped network file system.

The create and delete deltas used by Zebra are 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.

9 Conclusions

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. Large files are read or written 4-5 times as fast as other network file systems and small files are written 20%-3x faster.

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 the servers need not interpret the contents of stripes, so the server implementation is simple and efficient.

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. Deltas provide a simple way to maintain consistency among the components of the system.

There are at least four areas where we think Zebra could benefit from additional work:

Name caching. Without name caching, Zebra provides only about a 20% speedup for small writes in comparison to a non-striped Sprite file system. We think that a system with name caching would provide a much greater speedup.

Transaction processing. We expect Zebra to work well on the same workloads as LFS, which includes most workstation applications. However, there is little experience with LFS in a transaction processing environment and Seltzer's measurements suggest that there may be performance problems [Seltzer93]. More work is needed to understand the problems and see if there are simple solutions.

Metadata. It was convenient in the Zebra prototype to use a file in an existing file system to store the block pointers for each Zebra file, but this approach suffers from a number of inefficiencies. We think that the system could be improved if the metadata structures were redesigned from scratch with Zebra in mind.

Small reads. It would be interesting to verify whether there is enough locality in small file reads for prefetching of whole stripes to provide a substantial performance improvement.

Overall we believe that Zebra offers higher throughput, availability, and scalability than today's network file systems at the cost of only a small increase in system complexity.

10 Acknowledgments

We are grateful to our paper shepherd Paul Leach and the anonymous referees for their advice on improving this paper. Felipe Cabrera, Ann Drapeau, Ken Shirriff, Bruce Montague, and Mary Baker provided useful comments on various drafts of the paper. Ken Lutz, Peter Chen, Peter Belleau, and Ares Ho built the timer boards that proved invaluable in debugging the system and running the experiments.

11 References

[Baker91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, Measurements of a Distributed File System, *Proc. of the 13th Symp. on Operating Sys. Prin. (SOSP)*, Oct. 1991, 198-212. Published as *ACM SIGOPS Operating Systems Review 25*, 5.

[Baker92a] M. Baker and M. Sullivan, The Recovery Box: Using Fast Recovery to Provide High Availability, *Proc. of the Summer 1992 USENIX Conf.*, June 1992, 31-43.

[Baker92b] M. Baker, S. Asami, E. Deprit, and J. Ousterhout, Non-Volatile Memory for Fast, Reliable File Systems, *Proc. of the Fifth Int. Conf. on Arch. Support for Prog. Lang. and Operating Sys. (ASPLOS)*, Oct. 1992, 10-22.

[Bernstein81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys 13*, 2 (June 1981), 185-222.

[Birrell82] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, Grapevine: An Exercise in Distributed Computing, *COMM of the ACM 25*, 4 (Apr. 1982), 260-274.

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

[Cao93] P. Cao, S. B. Lim, S. Venkataraman, and J. Wilkes, The TickerTAIP parallel RAID architecture, *Proc. of the 20th Annual Int. Symp. of Computer Arch.*, May 1993, 52-63.

[Chen90] P. M. Chen and D. A. Patterson, Maximizing Performance in a Striped Disk Array, *Proc. of the 17th Annual Int. Symp. of Computer Arch.*, May 1990, 322-331.

[Chutani92] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham, The Episode File System, *Proc. of the Winter 1992 USENIX Conf.*, Jan. 1992, 43-60.

[Dibble88] P. C. Dibble, M. L. Scott, and C. S. Ellis, Bridge: A High-Performance File System for Parallel Processors, *Proc. of the 8th Int. Conf. on Dist. Computing Sys. (ICDCS)*, 1988, 154-161.

[Guy90] R. G. Guy, J. S. Heidemann, W. Mak, T. W. P. Jr., G. J. Popek, and D. Rothmeier, Implementation of the Ficus Replicated File System, *Proc. of the Summer 1990 USENIX Conf.*, June 1990, 63-71.

[Hagmann87] R. Hagmann, Reimplementing the Cedar File System Using Logging and Group Commit, *Proc. of the 11th Symp. on Operating Sys. Prin. (SOSP)*, Nov. 1987, 155-162. Published as *ACM SIGOPS Operating Systems Review 21*, 5.

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

[Lee92] E. K. Lee, P. M. Chen, J. H. Hartman, A. L. C. Drapeau, E. L. Miller, R. H. Katz, G. A. Gibson, and D. A. Patterson, RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service, Tech. Rep. UCB/CSD 92/672, Computer Science Division, EECS Dept., UCB, Feb. 1992.

[Liskov91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, Replication in the Harp File System, *Proc. of the 13th Symp. on Operating Sys. Prin. (SOSP)*, Oct. 1991, 226-238. Published as *ACM SIGOPS Operating Systems Review 25*, 5.

[LoVerso93] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler, sfs: A Parallel File System for the CM-5, *Proc. of the Summer 1993 USENIX Conf.*, June 1993, 291-305.

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

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

[Ousterhout88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, The Sprite Network Operating System, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

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

[Pierce89] P. Pierce, A Concurrent File System for a Highly Parallel Mass Storage Subsystem, *Proc. of the Fourth Conference on Hypercubes*, Mar. 1989.

[Rosenblum91] M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, *Proc. of the 13th Symp. on Operating Sys. Prin. (SOSP)*, Oct. 1991, 1-15. Published as *ACM SIGOPS Operating Systems Review 25*, 5.

[Satyanarayanan90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. 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 (Apr. 1990), 447-459.

[Schloss90] G. A. Schloss and M. Stonebraker, Highly Redundant Management of Distributed Data, *Proc. of the Workshop on the Management of Replicated Data*, Nov. 1990, 91-95.

[Seltzer93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, An Implementation of a Log-Structured File System for UNIX, *Proc. of the Winter 1993 USENIX Conf.*, Jan. 1993, 307-326.

[Shirriff92] K. Shirriff and J. 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] A. Siegel, K. Birman, and K. Marzullo, Deceit: A Flexible Distributed File System, *Proc. of the Summer 1990 USENIX Conf.*, June 1990, 51-61.

[Walker83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, The LOCUS Distributed Operating System, *Proc. of the 9th Symp. on Operating Sys. Prin. (SOSP)*, Nov. 1983, 49-70. Published as *ACM SIGOPS Operating Systems Review 17*, 5.

[Wilkes92] J. Wilkes, DataMesh research project, phase 1, *Proc. of the USENIX File Systems Workshop*, May 1992, 63-69.