Simplifying Distributed File Systems Using a Shared Logical Disk

Robert A. Shillner and Edward W. Felten

Dept. of Computer Science
Princeton University
{*ras,felten*}*@cs.princeton.edu*

Abstract

We present a new approach to building non-centralized ("serverless") distributed file systems, using the *shared logical disk*, which provides the abstraction of a fault-tolerant, entry-consistent, shared array of persistent data blocks. By separating the storage management level from higher-level file system functions like metadata management, we significantly reduce the overall complexity of the file system's implementation.

We describe the implementation of a file system and an underlying shared logical disk, designed to run on workstation networks and on the SHRIMP multicomputer. The file system level is very similar to a conventional uniprocessor file system, relying on the underlying shared logical disk to handle all communication, caching, and coherence operations. The shared logical disk provides some simple hooks to allow the file system to express its fault-tolerance requirements. We believe the resulting design is much simpler than other non-centralized distributed file systems.

1 Introduction

This paper describes a new way to organize distributed file storage. Our method allows us to implement a non-centralized ("serverless") distributed file system in a much simpler way than was previously possible. Rather than implementing a monolithic distributed file system, we break the storage functionality into two layers: a *shared logical disk* which manages data layout, caching, coherence, and fault tolerance; and a file system layer which uses these features to present a conventional distributed file system view to users. The result is a major simplification of the file system implementation.

Our approach is based on picking some ideas from the work of others, and combining them with some new ideas, particularly in the area of fault tolerance. From de Jonge et al. we take the logical disk concept [8], which we extend to the distributed case. From Attanasio et al. we take the idea of striping a logical device across distributed physical devices [2], which we extend by adding caching and cache coherence. From Mann et al. we take the idea of expressing consistency requirements as ordering constraints on propagation of dirty writes to disk[14], which we extend by applying it to cache-to-cache transfers. The result of combining these ideas, and adding some support for fault-tolerance, is a base layer that makes it easy to write non-centralized distributed file systems.

1.1 Logical disk

The shared logical disk is based on concept of *logical disk* [8]. A logical disk is an array of fixed-size data blocks, which processes may use to stably store persistent data. Processes refer to the data blocks via logical block numbers; the logical disk maps these block numbers onto physical disk addresses. The logical disk is free to store the blocks at any physical location it desires, and it may change the location of blocks at any time. The logical disk implementation may even store the data on multiple disks or on remote disks; the mapping is transparent to the user. For example, de Jonge, et. al. [8] use a logical disk to store data in a log-structured fashion; except for the increased performance, the log-structuring is invisible to the layers above it. In this way, a logical disk can turn an ordinary file system into a log-structured file system, without modifying the on-disk data structures or any of the file system's code.

Another use of logical disks is for data protection in the case of a disk failure. A logical disk implementation could mirror the data on duplicate physical disks, so that if one disk fails, a backup copy is available. A logical disk could also stripe the data across multiple disks and store RAID-style parity blocks on an extra disk.

1.2 Shared logical disk

The *shared logical disk* is a distributed variation of the logical disk; it gives processes on different machines concurrent access to the data stored in a logical disk. For good performance, each node caches recently used logical blocks in memory, and a cache coherence protocol ensures that each node only sees up-to-date versions of each logical block.

Locking The clients of a shared logical disk use a locking mechanism to ensure cache coherence. Before a client reads or writes a block, it must acquire a lock on the block. There are two types of locks: read/write (also called write locks or exclusive locks) and read-only (also called read locks or nonexclusive locks). Multiple clients are allowed to hold read locks on the same block simultaneously, but if any client holds a write lock on a block, no other client may hold any lock on the block. Thus, a client holding a read lock on a block may safely read the block without fear of another client writing it. Likewise, a client holding a write lock on a block may safely write the block without fear of obliterating other clients' partial writes or of other clients reading the block in a partially-updated state.

API Figure 1 shows an API for a shared logical disk. In this API, a client calls `readlock` or `writelock` to obtain a lock on a block and a pointer to a block in the local cache. If the lock cannot be granted immediately (perhaps because some other client holds a conflicting lock, or because the block must be read in from disk), the lock call blocks until the lock becomes available. The client uses the returned pointer to read or modify the block in its place in the cache. When the client no longer needs the lock, it calls `unlock`. If the client writes on a block locked for reading, or if the client uses a pointer returned from `readlock` or `writelock` after it has unlocked the block, then the results are undefined. Typically data written in this unsafe fashion will be lost, or (worse) be written to another, unrelated block. Data read unsafely may be stale, or it may come from an unrelated block.

Typcally, the local caches are write-back rather than write-through. Thus, dirty blocks are ordinarily flushed to disk when they are evicted from cache by the block-replacement policy. A client can use `sync` to synchronously force the most recent version of a particular block to disk.

Dependences If clients do not use `sync`, dirty blocks may be flushed to disk in any order. Under normal circumstances this poses no problem, but if one of the nodes used by the system crashes, then some of the dirty blocks written by clients may be on disk, while others may have been lost in the crash. Some clients, such as filesystems and database, store complex data structures on disk; a crash could corrupt those structures by adversarially "withholding" certain writes from the disk while allowing others to go through. Single-node filesystems have traditionally worked around this problem by liberal use of `sync` and by checking their data structures for consistency at recovery time. While this has been acceptable for a single-node system, a distributed file system built on a shared logical disk would not want to stop and check the system for consistency each time any node crashes.

In respose to this situation, the shared logical disk includes the `make_dependence` call, which allows clients to specify the order in which two blocks will reach disk. This call allows certain clients to limit the damage that a crash can cause. For example, in a Unix-like filesystem, creating a file requires write to two blocks: the directory fragment which contains the entry for the new file, and the new file's inode. The directory fragment contains a pointer to (*i.e.,* disk address of) the inode; thus, if the updated directory reaches disk, but a crash prevents the inode from reaching disk, the directory structure will be inconsistent: a directory entry will point to a garbage block. However, if the client which creates the file writes the inode first, then makes a dependence from the inode to the directory, and finally writes the directory, then a crash cannot cause the directory structure to become inconsistent, because the directory

`void *readlock(int block)` acquires a read lock on `block` and returns a pointer to the block in the local cache.

`void *writelock(int block)` acts as above, but acquires a write lock.

`void unlock(int block)` unlocks a previously locked block.

`void sync(int block)` forces `block` to disk.

`void make_dependence(int blk1, int blk2)` ensures that `blk1` reaches disk before `blk2`.

Figure 1: Shared logical disk primitives.

cannot reach disk before the inode.[1] The client could achieve the same effect by calling `sync` on the inode block after writing it, but the dependence approach uses nonblocking delayed writes, while the `sync` approach must wait for the synchronous write of the inode to complete.

1.3 Benefits

Jonge, et. al. [8] note several advantages that a non-shared logical disk provides for single-node I/O. First, separating file layout (mapping of file blocks to logical blocks) from disk layout (mapping of logical blocks to disk blocks) leads to simpler file system design and implementation. In addition, when using a logical disk, a single file system can be used in many different hardware environments, with different logical disk implementations tuned for each environment. Finally, a logical disk can provide novel disk-management techniques, such as log-structuring, to improve file system performance. An additional possibility is that the logical blocks may be striped across many physical disks, increasing throughput by using the disks in parallel.

A shared logical disk provides several additional advantages. First, a shared logical disk eases the construction of a distributed file system. Because a shared logical disk provides cache-coherent access to data, each node can essentially run a normal single-node filesystem – the filesystem need only be modified to lock blocks before using them and to create dependences in a few key places. All of the communication occurs in the logical disk layer, and so the distributed filesystem itself is kept simple.

Implementing a fault-tolerant shared logical disk greatly simplifies the task of building a fault-tolerant distributed file system. By separating fault tolerance from the bulk of the file system code, and exposing only a simple set of fault-tolerance hooks to the file system, we can simplify the fault-tolerance mechanisms.

Finally, the shared logical disk simplifies the implementation of cooperative caching and other techniques that try to optimize the use of distributed memory for data caching and prefetching.

2 Design and implementation

We have completed an implementation of the shared logical disk primitives. We strove for simplicity in the design wherever possible: we avoided ambitious or speculative strategies in favor of more established, easier-to-implement algorithms. We expect to reimplement parts of the system over time as we collect performance data. This section presents our implementation of the shared logical disk, including design alternatives and some future directions.

2.1 Overall structure

We implement our shared logical disk software as a user-level library; several instances of the library cooperate to implement a shared logical disk. Each cooperating instance of the library stores a fraction of the shared logical disk in a device or Unix file[2]; each instance also maintains an in-core cache of recently used blocks. The library instances communicate via stream sockets, either over a TCP/IP internet or over the SHRIMP interconnect using a fast sockets library [7] developed for the SHRIMP multicomputer. The library runs on Linux PCs and uses a custom user-level threads package for the 80x86 processor.

We choose a library-level approach rather than a kernel-level approach for the usual reasons: it simplifies development and debugging, it is more portable, and it allows multiple instances of the library to be running on the same machine at the same time. One disadvantage of the library-level approach is that only programs linked with the library can take direct advantage of the shared logical disk.[3] Another disadvantage is that all disk accesses must go through the kernel, causing mode switch and copying overheads. In addition, because Linux currently lacks a raw device driver for SCSI disks, the kernel redundantly caches all disk blocks read by our library. Finally, Linux has no support for user-level asynchronous I/O, so each instance of the shared logical disk library can have only one outstanding disk request at any time, and while a request is outstanding, the library cannot perform any other actions.

Using sockets for communication simplifies the task of detecting and dealing with node crashes. The use of sockets also allows us to test the correctness of our software in a more stable (but slower) environment before porting to the in-development SHRIMP sys-

[1] In fact, calling `sync` on the directory causes the inode (and all other blocks on which the directory depends) to be flushed to disk first.

[2] Note that although any user can link a program to the library, the library will be able to successfully open the device or file only if the user has the appropriate permissions.

[3] In fact, only those program instances which are cooperating to implement a shared logical disk may be clients of that logical disk. This restriction eases crash recovery because the system knows all possible clients in advance. We expect that a subsequent shared logical disk implementation could lift this restriction.

tem.

Our implementation statically assigns the logical blocks to the servers in a round-robin fashion. We choose this design for its simplicity, as it eliminates the need to keep track of the locations of logical blocks. Dynamic mapping of logical blocks to physical locations can improve spatial locality through improved data layout; however, we did not feel that the added complexity was worthwhile for a first implementation. We hope to explore more ambitious mapping schemes in the future.

Our implementation also does not store any redundant data, again for the sake of simplicity. A subsequent shared logical disk implementation could support data mirroring, RAID-style parity [19] or a hierarchical combination of both [24].

2.2 Cache coherence

We use a token-based write-back cache coherence protocol similar to that used in some distributed file systems [13, 14, 17], except that in our system the unit of coherence is a logical disk block instead of a file. With this scheme, any node holding a read token for a block may safely acquire a read lock on the block without sending any messages; likewise, any node holding a write token for a block may acquire a read or write lock on the block without communicating.

Each node manages the tokens for the blocks it owns. Each node's token directory records which nodes hold tokens for each of the node's blocks. A client (non-owner) node asks for a block's token by sending a *request* message to the block's owner. When it is safe to do so, the owner replies with a *grant* message, sending the block's data if the requestor did not already have a token for the block. The owner may later send a *revoke* message to the client, asking that the client return its token. After waiting for its user to unlock the block, the client responds with a *relinquish* message, which includes the block's data if the token was a write token. A client must also send the owner an unsolicited relinquish message when a block ages out of the client's cache. When an owner receives a request for a token while other clients hold conflicting tokens, the owner revokes the conflicting tokens before granting the token to the requesting client. In our system, every cached block must have a token; the cache evicts any block whose token is revoked.

2.3 Fault Tolerance

Our implementation has modest fault tolerance. It can keep running even if several nodes crash and restart. When a node crashes, some data written by the crashing node just before the crash may be lost, but all other data is persistent. Whenever a node is down, the system makes a reasonable effort to provide service; if it cannot, some accesses may be stalled until one or more of the crashed nodes have recovered. When a crashed node restarts, it rejoins its peers and the system functions normally.

Our basic approach to fault tolerance is simple. The non-failed nodes continue their work as normal. When a thread needs to communicate with a failed node, the thread blocks until a connection can be established to the failed node. When a node recovers from a crash, it immediately exchanges information with every other node so as to put the system into a consistent state. As always, the details are the tricky part of fault tolerance. The following sections describe how fault tolerance interacts with various aspects of the design. Note that although, for simplicity, we describe the fault tolerance strategy in terms of node failures, it also works correctly in the presence of network partitions.

Communication and Fault Tolerance Some operations require request/reply communication between two nodes. All such operations are coded so that the requesting node re-sends its request if the connection between the two nodes breaks before the reply arrives. Since all of the request/reply operations are idempotent, no harm can come of the occasional redundant request.

Data Loss and Fault Tolerance In a perfect world, no data would be lost when a node crashes. However, performance requires some form of write-behind, so failures cause data to be lost. Our system imposes the rule that when a node fails, some data written just before the crash may be lost. The exact window of vulnerability is an adjustable parameter, currently set to thirty seconds.

The shared logical disk library implements write-behind by running a special thread on each node. This thread periodically calls `sync` on all dirty blocks owned by the local node.

Tokens and Fault Tolerance When a node fails, it loses two kinds of information about tokens. First, it forgets about any tokens it was holding. Second, it forgets who holds tokens for the blocks it owns. When a failed node recovers, it exchanges information with every other node in order to establish a consistent state. In the exchange, each node informs the other of all tokens it holds on blocks owned by the other node. Each node then adjusts its state to reflect the

message received. Since a recovering node believes it holds no tokens, this exchange of information effectively causes a recovering node to relinquish any tokens it held when it failed.

Dependences and Fault Tolerance The interaction between dependences and fault tolerance is subtle. When we introduced dependences above, we gave a simplified definition. Here is the full definition:

> *make_dependence(A,B)* establishes the rule that all updates to A made before[4] the call are visible to every node before any updates to B made after the call are visible. This rule must hold in the presence of arbitrary failures.

If there were no failures, the cache coherence mechanism would be enough to enforce this rule. In the presence of failures, however, we must be more careful. First, we cannot write B to disk until A has been written to disk.

Second, until A is written to disk, no node may access B without first getting a valid copy of A. Otherwise, the system could get into an untenable state. All copies of A might be lost due to failures. Then, if the node that had accessed B tried to access A, it would be required to get an up-to-date version of A, which would be impossible to provide.

Our shared logical disk software implements dependences by arranging dependent blocks into a directed acyclic graph (DAG). If block `b2` depends on `b1`, then the DAG contains an edge from `b1` to `b2`. `make_dependence(b1, b2)` adds into the DAG an edge from `b1` to `b2`.

When the system writes a block to the physical disk, it deletes all of the block's out-edges. The system enforces the first rule by never writing a block to disk if it has any in-edges. When the system needs to sync a block which does have in-edges, the system syncs each of the blocks with edges pointing to the first block, possibly recurring backward along several more edges. Since syncing a block deletes its out-edges, eventually the first block will make it to disk.

The second rule is enforced by writing A to disk before giving any other node a copy of B. We could get better performance by sending a copy of A along with every copy of B, but for this first implementation we choose to start with a simpler solution.

The strategies described so far work together to provide fault tolerance for the shared logical disk's internal algorithms. Higher-level libraries can implement their own fault tolerance by making proper use of dependences.

[4] For the purpose of this rule, "before" and "after" are defined according to the partial order defined by entry consistency, not according to wall-clock time.

3 Shared logical disk performance

We have made a number of preliminary performance measurements of our shared logical disk prototype. The numbers reflect the initial state of the implementation; we expect the performance to improve as we tune the prototype to fit our hardware and software environment.

3.1 Apparatus

We took performance measurements of the shared logical disk running on two nodes of the Princeton SHRIMP multicomputer [5, 10]. SHRIMP consists of a number of ordinary Linux Pentium PCs connected by an Intel Paragon backplane. SHRIMP uses hardware support to provide protected, low-latency, user-level communication. The SHRIMP hardware has a raw user-to-user latency of about four microseconds; the SHRIMP stream sockets library, which we use, has a latency of about 12 μsec, and a peak bandwidth of about 12 MBytes/sec. Each SHRIMP node has a 60 Mhz Pentium processor, 36 MB of RAM, and a 4 GB Seagate ST15150ND disk drive.

3.2 Measurements

Figure 2 shows the bandwidth of a single node reading and writing blocks from local and remote disks, plotted for various block sizes, from 512 bytes to 64 kbytes. There are six curves, showing read and write behavior for each of three cases: accessing local disk from user space without using the shared logical disk software, accessing local disk using the shared logical disk, and accessing remote disk.

In the local case, bandwidth is roughly 4 MBytes/sec for all block sizes of 4 Kbytes or larger. Adding the shared logical disk layer does not affect performance much in the local case.

For writes, bandwidth increases linearly until the track size of our disk is reached. This corresponds to the case of requiring one disk revolution per write call. Using the shared logical disk adds no noticable overhead on local writes.

In the remote case, performance seems to be limited by internode communication, since reads and writes achieve about the same rate of roughly 2 MBytes/sec.

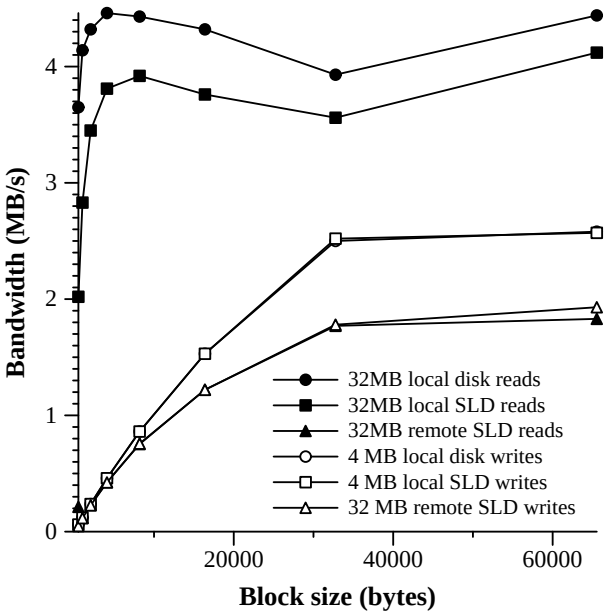

Figure 2: Reading and writing disk. Curves marked "SLD" are for tests using the shared logical disk; other curves rea The middle "curve" is actually two curves which are almost identical: writing local disk directly, and writing local disk via the shared logical disk layer.

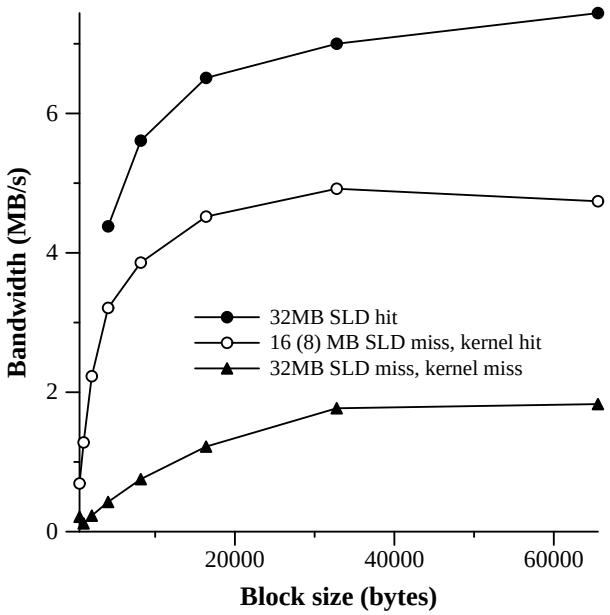

Figure 3: Cache effects for remote reads. These three curves show the effects of the shared logical disk's cache and the Linux kernel cache on remote reads. The bottom curve is the same as the bottom curve in Figure 2.

Figure 3 shows the effect of caching on remote reading. The bottom curve is the same as the bottom curve in Figure 2: it shows the performance of reading from remote disk. Recall that, since the shared logical disk is a user-level library, the Linux kernel redundantly caches blocks from the disk; the middle curve of Figure 3 shows the performance of reading from the Linux buffer cache of a remote machine. The top curve shows the results of hitting in the shared logical disk's cache on a remote machine. Due to immature SHRIMP support software, we cannot yet obtain this curve's data for block sizes smaller than 4 kbytes.

3.3 Tuning

There are many ways we could improve the performance of our shared logical disk in the future. First, we could take advantage of the SHRIMP hardware by doing data transfers directly from kernel buffers into remote memory; in some cases SHRIMP would allow us to DMA data directly into remote memory.

Second, we could eliminate much of the kernel overhead by simplifying the read path. If the shared logical disk daemon is a trusted process, it can be allowed to pin its cache pages into memory; then the kernel could transfer data directly to and from those pinned pages, saving a kernel-level copy.

Finally, of course, many performance-improving transformations that are used in other systems, such as log-structuring, are viable in our design. We see no fundamental reason why the performance of a carefully tuned shared logical disk-based system should be different than that of a monolithic system.

4 Distributed file system

We used the shared logical disk to implement a non-centralized distributed file system. Chief among this file system's characteristics is its close resemblance to an ordinary single-node file system; multiple instances of this file system on different nodes safely share the data stored in a shared logical disk. The shared logical disk provides all of the data communication, synchronization, caching, coherence and fault tolerance support. This section describes the design and implementation of the distributed file system.

4.1 Overview

The distributed file system, like the shared logical disk, is a user-level library. The file system library provides a collection of entry points which mimic the system calls of the Unix file system. A collection of running programs linked with the library share a directory hierarchy which any of the programs may read or modify. The file system stores its data on a shared logical disk, and thus the data are distributed among many physical disks.

The distributed file system uses the `readlock` and `writelock` primitives to safely coordinate actions performed on different nodes. The file system instances communicate only through data stored in logical disk blocks; they do not use any outside channels. Likewise, the only synchronization is the implicit in the locking mechanism.

The design of the file system closely matches that of existing single-node file systems. Only three modifications are necessary to distribute a centralized file system in this way. First, the file system must lock blocks before using them, as dictated by the shared logical disk's entry consistency model. Second, the file system must not cache data above the shared logical disk layer while it is not holding a lock on the block from which the data came. If it did, then its cached copies would grow stale when some other node modified the logical blocks backing the cached data. Thus, there can be no in-core inode table. Instead, the file system must acquire a lock on an inode's block each time it wishes to read the inode, and the file system must make all inode modifications in place on the logical disk. However, this does not imply a performance penalty, because the shared logical disk caches recently-used blocks in local memory. The third necessary modification is that the file system must use `make_dependence` to specify an ordering for updates critical to the integrity of the on-disk data structures. A detailed explanation of this requirement appears in Section 4.3.

A file system modified in these ways becomes a noncentralized distributed file system when used with a shared logical disk. The file system is noncentralized because there is no central server responsible for handling any aspect of the file system; instead the individual file system instances lock and unlock blocks as needed to satisfy client requests. As long as ownership of the most frequently written blocks is distributed reasonably among the nodes, there is no central bottleneck.

The three modifications mentioned so far are enough to provide a correctly-functioning distributed file system. However, other modifications can be made to improve performance. For instance, the distributed file system must guard against false sharing of logical blocks between nodes. False sharing occurs when different nodes write (or read and write) unrelated items which reside in the same logical block. For example, false sharing occurs if different machines operate on two different inodes which happen to reside in the same logical block. Since the smallest cache coherence unit is an entire block, the shared logical disk cannot know whether two accesses to the same block are unrelated. The cache coherence protocol forces one node to wait while the other completes its access, then ships the entire block over to the first node so that it can perform its operations.

Genuine write-sharing can also be a performance problem in a distributed file system built on a shared logical disk. For instance, in the Unix file system [3], the superblock contains a pool of free blocks; when a process needs to allocate a block, it takes one from the pool in the superblock. A distributed file system on a shared logical disk should not use this scheme, because each time any node needs to allocate a block it must acquire an exclusive write lock on the superblock. Other examples of write-shared resources are the Unix file system's free-inode cache and the Fast File System's [15] block-use bitmaps.

4.2 On-disk data structures

Our on-disk data structures are similar to those found in the traditional Unix file system [3]. Each file has an inode, which contains some metadata (file size, file type, link count, owner, timestamps, etc) as well as pointers to the file's data blocks. Directories are kept in files; each directory entry consists of a file name and an inode number. Blocks which are not part of any file are kept in a freelist; each node in the freelist contains many pointers to free blocks plus a pointer to the next node on the freelist. The superblock keeps a pointer to the head of the freelist.

We make some changes to the basic Unix file system design in order to better take advantage of the characteristics of our shared logical disk implementation. For example, the Unix file system stores several inodes together in a block. To avoid false sharing between inodes, our file system expands the inode structure to fill an entire block. Since most Unix files are small, block-sized inodes waste much space. To compensate for this, the file system stores small files in the inode block itself [16]. If a small file later grows too large to fit in the inode block, the system converts the file to the normal storage method.

A consequence of keeping only one inode per block is that we do not need to statically partition the disk

into inodes and data blocks, as Unix does. We allocate both data blocks and inodes from the same freelist.

Another result of block-sized inodes is that an inode holds about as many pointers as an indirect block. Since the largest file accessible by an inode with an indirect pointer is only about twice as big as the largest direct-only file, our file system dispenses with the indirect pointer. Files too large for direct pointers use the inode's double-indirect pointer.

Another potential performance bottleneck in a standard file system is the free block pool stored in the superblock. To avoid the ping-pong effect of nodes write-locking the superblock on each block allocation, our file system instead uses one free block pool per node, each stored in a separate block. Each node allocates blocks from its own pool, and since the pools are not shared, each pool can remain in cache at its own node. When a node's pool empties, the node can acquire a lock on the superblock and refill its pool from the shared freelist.

The Unix file system's `unlink` operation does not immediately delete a open file. Instead, the file continues to exist on disk until no process has it open, at which point the system reclaims the file's disk blocks. Many programs use this feature for temporary files, and it is easy to implement in a centralized, single-node file system. Before deleting a file, the system checks the global open-file table to make sure no process has it open; when closing a file, the system checks the file's link count in the in-memory inode table.

For a distributed file system built on a shared logical disk, this feature is difficult to implement efficiently. While it is easy enough to check a file's link count (stored in the inode) when closing the file, there is no central open-file table to check when removing a directory link. The file system currently works around this problem by maintaining, in each file's inode, a count of the number times the file has been opened. `unlink` checks the open count on the inode when removing the last link; if it is zero, `unlink` deletes the file. Maintaining the open count implies that each open and close operation must obtain a write lock on the file's inode. Since write locks are exclusive, read-sharing of a file by different nodes causes cause the file's inode to bounce back and forth across the network. Applications can mitigate this problem by caching open files, because reading files which are already open does not modify the inode.

4.3 Fault tolerance

The distributed file system maintains fault tolerance using the dependence mechanism provided by the shared logical disk. The file system creates a dependence whenever it makes a structural change to its on-disk data structures.

For example, consider what happens when a new file is created. The system takes a block from the node's free block pool to hold the new file's inode. The system must write the free pool to disk to record the block's removal from the list of free blocks, and it must write the new file's parent directory block to record the block's addition to the directory tree. If the write to the free pool were lost but the write to the parent directory block went through, then the block would be seen as still free; it could be (incorrectly) allocated again. The file system prevents this problem by creating a dependence from the free pool to the parent directory block, ensuring that the free pool will be written to disk before the parent directory block.

A similar situation arises whenever the system moves a block from one linked structure to another. Granger and Patt summarize the ordering constraints with the following pair of rules: (1) always remove the block from the old structure before linking it into the new structure, and (2) never point to a block before it has been initialized. The file system establishes dependences to ensure that these updates occur in order, even in the presence of crashes.

Even after this precaution is taken, there is still a danger that a node failure can cause a space leak, because a block that was being moved from one structure to another is seen as removed from the old structure, but not seen as added to the new structure. Such a block appears nowhere in the file system's data structures — not even in the free list.

The file system deals with these occasional space leaks by garbage collection. Whenever the number of leaked blocks becomes significant, an operator runs a mark-and-sweep garbage collector over the file system's data structures to find any leaked blocks and add them to the free list. The current garbage collector implementation requires that no other file system activity occur while it is running, but we plan to implement a concurrent garbage collector in the future.

4.4 Scalable NFS server

A key application of a distributed file system running on a multicomputer such as SHRIMP is a scalable NFS server. Each SHRIMP node has a 10 Mb/s Ethernet port in addition to its connection to the internal SHRIMP interconnect. If each SHRIMP node runs an NFS server exporting the shared logical disk's file system, then clients on each SHRIMP node's Ethernet can access data on the file system via the un-

modified NFS protocol. Because the shared logical disk's file system is cache-coherent, the NFS clients see the same file system iamge. Each SHRIMP node can serve data to as many clients as the Ethernet's bandwidth allows, and total system throughput can scale as more SHRIMP nodes are added.

5 Related work

IBM's Virtual Shared Disk [2] allows multiple workstations to access remote disks as if they were connected locally; multiple (possibly distributed) physical disks may be grouped together to form a single virtual disk. VSD supports read-block and write-block operations; there is no caching or locking. Thus, VSD applications which desire caching must provide it themselves, and distributed VSD applications must synchronize via some external mechanism. The designers seem to expect distributed databases to be the most common application of VSD.

The shared logical disk's token-based cache-coherence protocol is based on similar protocols used in distributed file systems such as Sprite [18, 17], AFS [13] and Echo [4, 14]. While these systems vary in the details of the coherence protocol (particularly in cases of concurrent write-sharing), their basic idea is the same: clients acquire read and write tokens before accessing file data, and the servers revoke these tokens when other clients request conflicting tokens. An important difference between these systems and the shared logical disk is the granularity of coherence; in these systems the granularity is an entire file, while for the shared logical disk it is an individual disk block.

Cooperative caching schemes [6, 9] seek to use the disk caches of networked machines collectively. These designs perform better than a traditional memory hierarchy (local cache, server cache, server disk) because they can use the memories of idle clients. Aggressive strategies approximate a global least-recently-used cache replacement policy for all cached data. Although we believe the shared logical disk is not incompatible with cooperative caching, our current prototype does not implement it.

Soft updates [11] are a technique for safely using delayed writes for filesystem metadata. Like the shared logical disk's dependences, soft updates allow a filesystem to specify the order in which certain writes reach disk. A soft update is a modification of a cached metadata block (such as an inode or directory) which can be selectively rolled back when the block is written to disk in order to satisfy the filesystem's ordering constraints. Soft updates can ensure that the inode for a newly-created file is written to disk before its associated directory entry is written to disk, while still allowing other, unrelated modifications to the directory block to be written before the inode. The soft updates approach keeps track of individual metadata updates; in contrast, the shared logical disk records dependences at whole-block granularity. Soft updates' fine-grain approach avoids false dependences arising from unrelated updates to the same metadata block; however, the soft updates mechanism must have detailed knowledge of the file system's data structures. The shared logical disk's performance suffers from false dependencs, but it uses a much simpler mechanism and does not need any knowledge about file system data structures.

Both DataMesh [23] and xFS [1] seek to use a central, trusted core of machines as a scalable file server for a larger, less-trusted group of file system clients. In both cases, the core communicates with the external clients using a standard protocol, such as NFS [21], while core machines communicate internally using high-performance proprietary protocols tuned to the core environment.

Sawmill [22], Zebra [12] and xFS [1] are network file systems based on log-structuring [20] and RAID [19]. Sawmill is a centralized-server system which stores data on a high-bandwidth disk array; specialized hardware allows data to flow directly between the disk array and the network without going through the server's memory system. In Zebra, clients stripe data in parallel across many servers, but they must use a centralized file manager for all metadata access. xFS takes Zebra one step further by eliminating all centralized components. This "serverless" solution, while it eliminates central bottlenecks, seems overly complex.

6 Future work

The shared logical disk is a simple idea which can be taken in many different directions. We plan to continue active development of the shared logical disk and its applications as the SHRIMP environment matures.

6.1 Shared logical disk

As noted in Section 3.3, first among our priorities for the shared logical disk software is understanding and improving its performance. There are several other areas we intend to pursue. For example, we believe that the shared logical disk is well-suited for cooperative caching, and that its cache coherence protocol

can be modified to allow direct client-to-client transfer of cached data.

We plan to reimplement dependences. The current strategy forces blocks to disk to enforce the semantics of dependences. We plan to explore implementations based on copying dependent blocks rather than syncing them.

While the shared logical disk's cache-coherence facility is appropriate for distributed file systems, certain classes of applications do not require cache coherence. Databases and many out-of-core parallel codes synchronize externally and could therefore make do with simple `read` and `write` operations. If these operations yield higher performance than the corresponding cache-coherent operations (`readlock`, `writelock` and `unlock`), then the shared logical disk can be made to support them.

The shared logical disk can also be extended to transparently store redundant data, both for availability and for protection from media failures. Another area for future study is alternative logical-to-physical block mappings, both static (such as the current round-robin) and dynamic (such as a log-structured approach [8]). Finally, we may integrate the shared logical disk into the Linux kernel. IBM's Virtual Shared Disk [2] exists as a psuedo device driver with a device special file in `/dev`; we can do something similar. An important obstacle to this is that SHRIMP communication is currently defined only from user level, and enabling kernel-to-kernel SHRIMP communication may involve significant work.

6.2 Distributed file system

The current distributed file system is a proof-of-concept prototype, based on the outdated traditional Unix file system. In the future we expect to modify or reimplement the distributed file system to incorporate modern file system design principles, such as intelligent data layout, sub-block fragments and/or log structuring. We also intend to use the distributed file system to build a scalable NFS server for SHRIMP, possibly for department-wide use.

Currently, the file system's garbage collector is synchronous — it must be run when there is no other file system activity. We have a design for a concurrent garbage collector which we hope to implement.

An area for future research is designing file systems which use dependences to reduce disk write traffic. Our current file system generates dependence cycles more frequently than necessary, causing disk writes at inopportune times. Another possible area of study is a comparison of the performance of metadata logging versus the dependences-and-garbage-collection strategy we currently use.

7 Conclusions

Distributed file system design has traditionally been complex, particularly for systems which seek to eliminate centralized bottlenecks. The shared logical disk is a new abstraction for managing distributed stable storage, which greatly decreases the complexity required in a noncentralized distributed file system. The shared logical disk provides storage in a logical array of fixed-size data blocks, and it handles caching, coherence and fault tolerance. A locking mechanism provides entry consistency for the shared data blocks, and an update-ordering mechanism provides support for fault tolerance.

A distributed file system built on a shared logical disk works essentially like a traditional single-node file system. Only three modifications are necessary to distribute a centralized file system in this way. The file system (1) must lock blocks before using them, (2) must not cache data above the shared logical disk layer, and (3) must specify an ordering for updates critical to the integrity of the on-disk data structures.

We have developed a user-level library which implements a shared logical disk. Running programs linked with the library cooperate to implement a shared logical disk. The prototype is about 2800 lines of C and runs on the SHRIMP multicomputer. We have implemented a distributed file system library on top of the shared logical disk library; it closely resembles a traditional Unix file system and consists of about 1800 lines of C.

Initial performance results are encouraging. Since the SHRIMP hardware and software are still quite immature, we are confident that performance can be made still better. The idea of building distributed file systems with a shared logical disk shows great potential and we are eager to explore new implementations of both the shared logical disk and distributed file systems.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 109–126, 1995.

[2] C. R. Attanasio, M. Butrico, C. A. Polyzois, S. E. Smith, and J. L. Peterson. Design and implementation of a recoverable virtual shared disk. Technical

report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1994.

[3] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Research Report 111, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1993.

[5] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–153, 1994.

[6] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 267–280, 1994.

[7] Stefanos N. Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream sockets on SHRIMP. Submitted for publication.

[8] Wiedbren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 15–28, 1993.

[9] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 201–212, 1995.

[10] Edward W. Felten, Richard Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li. Early experience with message-passing on the SHRIMP multicomputer. In *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996. To appear.

[11] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 49–60, 1994.

[12] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 29–43, 1993.

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

[14] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[15] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[16] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Software – Practice and Experience*, 14(4):365–368, April 1984.

[17] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[18] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[19] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM SIGMOD*, pages 1–15, 1988.

[20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 1–15, 1991.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the 1985 Summer USENIX Technical Conference*, pages 119–130, 1985.

[22] Ken Shirriff and John Ousterhout. Sawmill: A high-bandwidth logging file system. In *Proceedings of the 1994 Summer USENIX Technical Conference*, pages 49–60, 1994.

[23] John Wilkes. DataMesh — parallel storage systems for the 1990s. In *Proceedings of the 11th IEEE Symposium on Mass Storage*, pages 131–136, 1991.

[24] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 96–108, 1995.