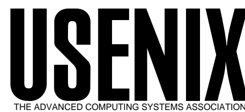
USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000



© 2000 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Scalability and Failure Recovery in a Linux Cluster File System

**Kenneth W. Preslan, Andrew Barry, Jonathan Brassow,
Michael Declerck, A.J. Lewis, Adam Manthei,
Ben Marzinski, Erling Nygaard, Seth Van Oort,
David Teigland, Mike Tilstra, Steven Whitehouse,
and Matthew O'Keefe**

Sistina Software, Inc.
1313 5th St. S.E.
Minneapolis, Minnesota 55414
+1-612-379-3951, okeefe@sistina.com

Abstract

In this paper we describe how we implemented journaling and recovery in the Global File System (GFS), a shared-disk, cluster file system for Linux. We also present our latest performance results for a 16-way Linux cluster.

1 Introduction

Traditional local file systems support a persistent name space by creating a mapping between blocks found on disk drives and a set of files, file names, and directories. These file systems view devices as local: devices are not shared so there is no need in the file system to enforce device sharing semantics. Instead, the focus is on aggressively caching and aggregating file system operations to improve performance by reducing the number of actual disk accesses required for each file system operation [1], [2].

New networking technologies allow multiple machines to share the same storage devices. File systems that allow these machines to simultaneously mount and access files on these shared devices are called *shared file systems* [3], [4], [5], [6], [7]. Shared file systems provide a server-less alternative to traditional distributed file systems where the server is the focus of all data sharing. As shown in Figure 1, machines attach directly to devices across a *storage area network* [8], [9], [10].

A shared file system approach based upon a shared network between storage devices and machines offers several advantages:

1. *Availability* is increased because if a single client fails, another client may continue to process its workload because it can access the failed client's files on the shared disk.
2. *Load balancing* a mixed workload among multiple clients sharing disks is simplified by the client's ability to quickly access any portion of the dataset on any of the disks.
3. *Pooling* storage devices into a unified disk volume equally accessible to all machines in the system is possible, which simplifies storage management.
4. *Scalability* in capacity, connectivity, and bandwidth can be achieved without the limitations inherent in network file systems like NFS designed with a centralized server.

In the following sections we describe GFS version 4 (which we will refer to simply as GFS in the remainder of this paper), the current implementation including the details of our journaling code, new scalability results, changes to the lock specification, and our plans for GFS version 5, including file system versioning with copy-on-write semantics to support on-line backups.

2 GFS Background

Previous versions of GFS are described in the following papers: [7], [6], [11]. In this section we provide a sum-

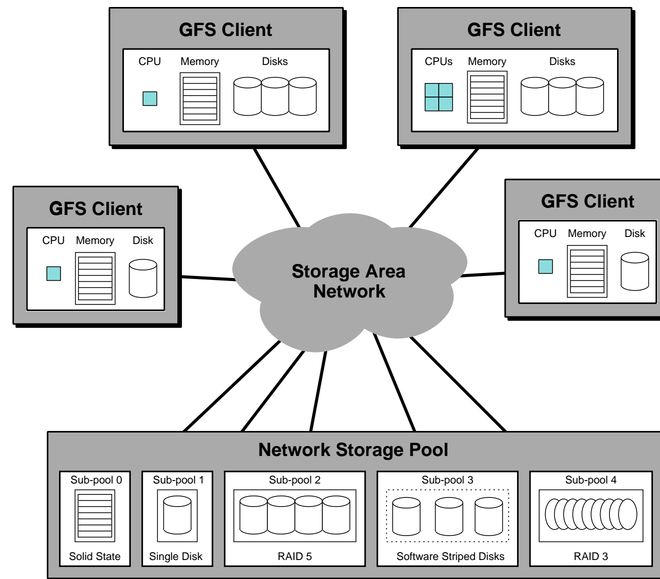


Figure 1: A Storage Area Network

mary of the key features of the Global File System.

2.1 DMEP

Device Memory Export Protocol (DMEP) is a mechanism used by GFS to synchronize client access to shared metadata. They help maintain metadata coherence when metadata is accessed by several clients. The DMEP SCSI command allows device to export memory to clients, and clients map lock state into these memory buffers. The lock state is contained in the storage devices (disks) and accessed with the SCSI DMEP command [12]. The DMEP command is independent of all other SCSI commands, so devices supporting the locks have no awareness of the nature of the resource that is locked (or even that the buffers are used to implement locks). The file system provides a mapping between file metadata and DMEP buffers.

Originally GFS used Dlocks SCSI command, which had the device maintaining the lock semantics as well as the lock state. The advantage of DMEP over Dlocks [13] is that the SCSI command is simpler, and the lock semantics may be modified on the client without affecting the SCSI command definition. This change was suggested by several SCSI disk vendors to ease implementation and create a more general SCSI synchronization primitive. All the semantics that used to be implemented by the SCSI devices as part of the Dlock command are now implemented on the client. The DMEP devices are just a reliable shared memory location.

In the event that a lock device is turned off and comes back on, all the state of the DMEP buffers on the device could be lost. Though it would be helpful if the locks were stored in some form of persistent storage, it is unreasonable to require it. Therefore, lock devices should not accept DMEP commands when they are first powered up. The devices should return failure results to all DMEP actions until a DMEP enable command is issued to the drive.

In this way, clients of the lock device are made aware that the locks on the lock device have been cleared, and can take action to deal with the situation. This is extremely important, because if machines assume they still hold locks on failed devices or on DMEP servers that have failed, then two machines may assume they both have exclusive access to a given lock. This inevitably leads to file system corruption.

The DMEP specification has been implemented as a server daemon called *memexpd* that can run on any UNIX machine, so that users need not have disk drives with special DMEP firmware to run GFS[14].

2.2 Lock Semantics

2.2.1 Expiration

In a shared disk environment, a failed client cannot be allowed to indefinitely hold whatever locks it held when it failed. Therefore, each client must periodically increment a counter in a DMEP buffer on the disk. If this counter (which the clients must monitor) isn't incremented for a given period of time, recovery functions can be started to free the lock state associated with that client. When a client fails to update its counter it is referred to as *timed-out*, and the act of updating the counter is often referred to as heartbeating the DMEP device.

2.2.2 Conversion Locks

The conversion lock is a simple, single-stage queue used to prevent writer starvation. In previous lock protocols used by GFS, one client may try to acquire an exclusive lock but fail because other clients are constantly acquiring and dropping the shared lock. If there is never a gap where no client is holding the shared lock, the writer requesting exclusive access never gets the lock. To correct this, when a client unsuccessfully tries to acquire a lock, and no other client already possesses that lock's conversion, the conversion is granted to the unsuccessful client. Once the conversion is acquired, no other clients can acquire the lock. All the current holders eventually unlock, and the conversion holder acquires the lock. All of a client's conversions are lost if the client expires.

2.3 Pool - A Linux Volume Driver

The Pool logical volume driver coalesces a heterogeneous collection of shared storage into a single logical volume. It was developed with GFS to provide simple logical device capabilities and to deliver DMEP commands to specific devices at the SCSI driver layer [15]. If GFS is used as a local file system where no locking is needed, then Pool is not required.

Pool also groups constituent devices into sub-pools. Subpools are an internal construction which does not affect the high level view of a pool[1] as a single storage device. This allows intelligent placement of data by the file system according to sub-pool characteristics. If one sub-pool contains very low latency devices, the file system could potentially place commonly referenced metadata there for better overall performance. There is not yet a GFS interface designed to allow this. Sub-pools are currently used in a GFS file system balancer [16]. The balancer moves files among sub-pools to spread data more evenly. Sub-pools now have an additional "type" designation to support GFS journaling. The file system requires that some sub-pools be reserved for journal space. Ordinary sub-pools will be specified as data space.

[1]The logical devices presented to the system by the Pool volume driver are called "pools".

2.4 File System Metadata

GFS distributes its metadata throughout the network storage pool rather than concentrating it all into a single superblock. Multiple resource groups are used to partition metadata, including data, dinode bitmaps and data blocks, into separate groups to increase file system scalability, avoid bottlenecks, and reduce the average size of typical metadata search operations. One or more resource groups may exist on a single device or a single resource group may include multiple devices.

Resource groups are similar to the Block Groups found in Linux's Ext2 file system. Like resource groups, block groups exploit parallelism and scalability by allowing multiple threads of a single computer to allocate and free data blocks; GFS resource groups allow multiple clients to do the same.

GFS also has a single block, the superblock, which contains summary metadata not distributed across resource groups, including miscellaneous accounting information such as the block size, the journal segment size, the dinode numbers of the two hidden dinodes and the root dinode, some lock protocol information, and versioning information.

Formerly, the superblock contained the number of clients mounted on the file system, bitmaps to calculate the unique identifiers for each client, the device on which the file system is mounted, and the file system block size. The superblock also once contained a static index of the resource groups which describes the location of each resource group and other configuration information. All this information has been moved to hidden dinodes (files).

There are two hidden dinodes:

1. *The resource index* – The list of locations, sizes, and glocks associated with each resource group

2. *The journal index* – The locations, sizes and glocks of the journals

This data is stored in files because it needs to be able to grow as the file system grows. In previous versions of GFS, we just allocated a static amount of space at the beginning of the file system for the Resource Index metadata, but this will cause problems when we expand the file system later. If this information is placed in a file, it is much easier to grow the file system at a later time, as the hidden metadata file can grow as well.

The Global File System uses Extendible Hashing [17], [7], [18] for its directory structure. Extendible Hashing (ExHash) provides a way of storing a directory's data so that any particular entry can be found very quickly. Large directories do not result in slow lookup performance.

2.5 Stuffed Dinodes

A GFS dinode takes up an entire file system block because sharing a single block to hold metadata used by multiple clients causes significant contention. To counter the resulting internal fragmentation we have implemented dinode stuffing which allows both file system information and real data to be included in the dinode file system block. If the file size is larger than this data section the dinode stores an array of pointers to data blocks or indirect data blocks. Otherwise the portion of a file system block remaining after dinode file system information is stored is used to hold file system data. Clients access stuffed files with only one block request, a feature particularly useful for directory lookups since each directory in the pathname requires one directory file read.

GFS assigns dinode numbers based on the disk address of each dinode. Directories contain file names and accompanying inode numbers. Once the GFS lookup operation matches a file name, GFS locates the dinode using the associated inode number. By assigning disk addresses to inode numbers GFS dynamically allocates dinodes from the pool of free blocks.

2.6 Flat File Structure

GFS uses a flat pointer tree structure as shown in Figure 2. Each pointer in the dinode points to the same height of metadata tree. (All the pointers are direct pointers, or they are all indirect, or they are all double indirect, and so on.) The height of the tree grows as large as necessary to hold the file.

The more conventional UFS file system's dinode has a fixed number of direct pointers, one indirect pointer, one double indirect pointer, and one triple indirect pointer. This means that there is a limit on how big a UFS file can grow. However, the UFS dinode pointer tree requires fewer indirections for small files. Other alternatives include extent-based allocation such as SGI's EFS file system or the B-tree approach of SGI's XFS file system [19]. The current structure of the GFS metadata is an implementation choice and these alternatives are worth exploration in future versions of GFS.

3 Improvements in GFS Version 4

We describe some of the recent improvements to GFS in the following sections.

3.1 Abstract Kernel Interfaces

We have abstracted the kernel interfaces above GFS, to the file-system-independent layer, and below GFS, to the block device drivers, to enhance GFS's portability. We hope to complete a port to FreeBSD sometime in 2001.

3.2 Fibre Channel in Linux

Until the summer of 1999, Fibre Channel (FC) support in Linux was limited to a single machine connected to a few drives on a loop. However, progress has been made in the quality of Fibre Channel fabric drivers and chipsets available on Linux. In particular, QLogic's QLA2100 and QLA2200 chips are supported in Linux, with multiple GPL'ed drivers written by QLogic and independent open source software developers. In testing in our laboratory with large Fabrics (32 ports) and large numbers of drives and GFS clients, the Fibre Channel hardware and software worked fine in static environments. Other FC card vendors like Interphase, JNI, Compaq, and Emulex are beginning to support Linux drivers for their cards.

However, it is possible to use GFS to share network disks exported through standard, IP-based network interfaces like Ethernet using Linux's Network Block Device software. In addition, new, fast, low-latency interfaces like

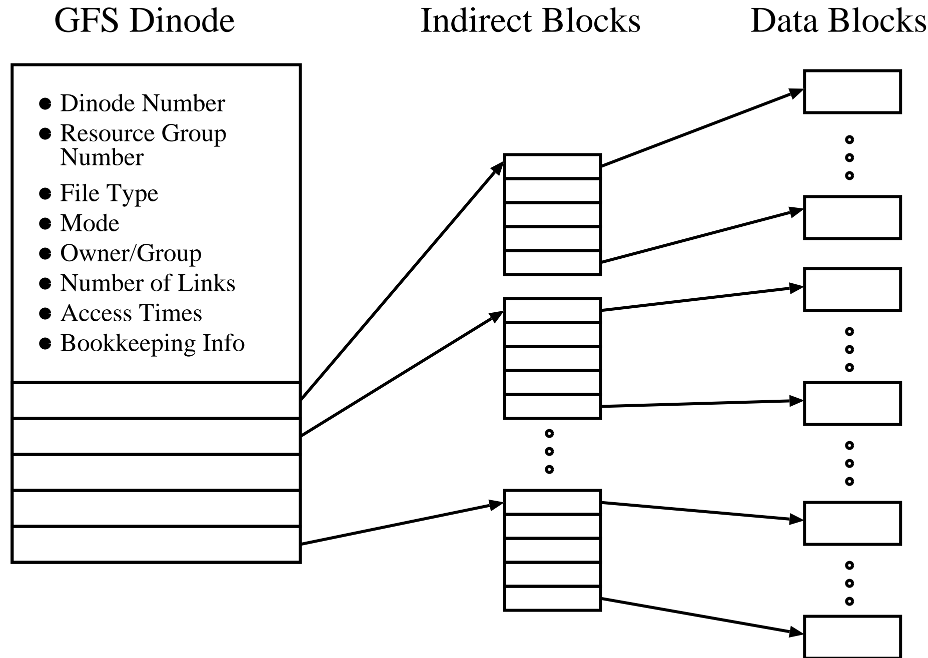


Figure 2: A GFS dinode. All pointers in the dinode have the same height in the metadata tree.

Myrinet combined with protocol layers like VIA hold the promise of high performance, media-independent storage networks.

3.3 Booting Linux from GFS and Context-Sensitive Symbolic Links

It has become possible to boot Linux from a GFS file system. In addition, GFS now supports context-sensitive symbolic links, so that Linux machines sharing a cluster disk can see the same file system image for most directories, but where convenient (such as /etc/???) can symbolically link to a machine-specific configuration file.

These two features help support a single system image by providing for a shared disk from which the machines in a cluster can boot up Linux, yet through context-sensitive symbolic links each machine can still maintain locally-defined configuration files. This simplifies system administration, especially in large clusters, where maintaining a consistent kernel image across hundreds of machines is a difficult task.

3.4 Global Synchronization in GFS

The lock semantics used in previous versions of GFS were tied directly to the SCSI Dlock command. This tight coupling was unnecessary, as the lock usage in GFS could be abstracted so that GFS machines could exploit any global lock space available to all machines. GFS-4 supports an abstract lock module that can exploit almost any globally accessible lock space, not just DMEP. This is important because it allows GFS cluster architects to buy any disks they like, not just disks that contain DMEP firmware.

A GFS lock module can implement callbacks to allow metadata caching and to improve lock acquisition latencies as shown in Figure 3. When client 2 needs a lock exclusively that is already held by client 1, client 2 first sends it's normal DMEP SCSI request to the disk drive (step 1 in the figure). This request fails and returns the list of holders, which happens to be client 1 (step 2). Client 2 sends a IP callback to client 1, asking 1 to give up the lock (step 3). Client 1 syncs all dirty (modified) data and metadata buffers associated with that lock to disk (step 4), and releases the lock. Client 2 may then acquire the lock (step 5).

Because clients can communicate with each other, they may hold locks indefinitely if no other clients choose to

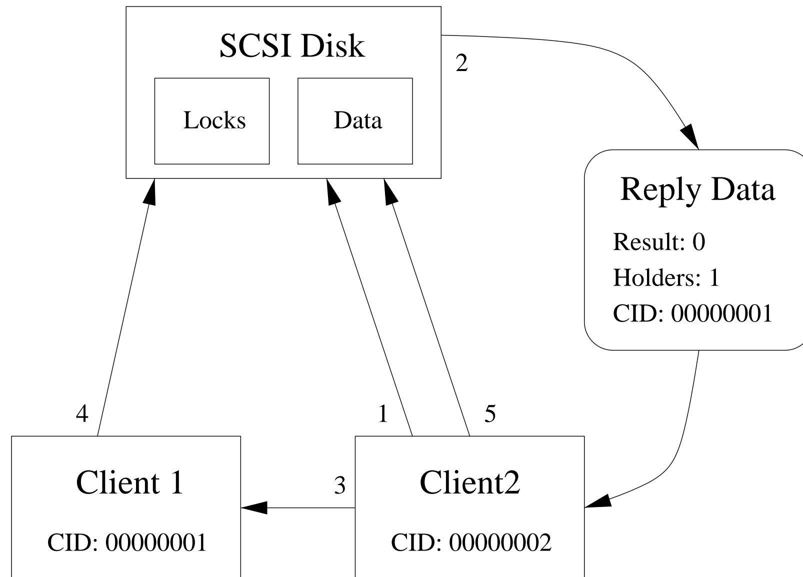


Figure 3: Callbacks on glocks in GFS

read from inodes associated with dlocks that are held. As long as a client holds a lock, it may cache any writes associated with the lock. Caching allows GFS to approach the performance of a local disk file system; our goal is to keep GFS within 10-15% of the performance of the best local, journaled Linux file systems across all workloads, including small file workloads.

In GFS-4, write caching is write-back, not write-through. GFS uses Global Locks (glocks). GFS uses interchangeable locking modules, some of which map glocks to DMEP buffers. Other locking methods, such as a distributed lock manager [9] or a centralized lock server, can also be used. GFS sees the Glocks as being in one of three states:

1. *Not Held* – This machine doesn't hold the Glock. It may or may not be held by another machine.
2. *Held* – this machine holds the Glock, but there is no current process using the lock. Data in the machine's buffers can be newer than the data on disk and there can be incore transactions for that glock. If another machine asks for the lock, the current holder will sync all the dirty transactions and buffers to disk and release the lock.
3. *Held + Locked* – the machine holds the Glock and there is a process currently using the lock. There can be newer data in the buffers than on disk. If another machine asks for the lock, the request is ignored temporarily, and is acted upon later. The lock is not released until the process drops the Glock down to the Held state.

When a GFS file system writes data, the file system moves the Glock into the Held+Locked state, acquiring the lock exclusively, if it was not already held. If another process is writing to that lock, and the Glock is already Held+Locked, the second process must wait until the Glock is dropped back down to Held.

The Write is then done asynchronously. The I/O isn't necessarily written to disk, but the cache buffer is marked dirty. The Glock is moved back to the Held state. This is the end of the write sequence.

The Buffers remain dirty until either bdflush or a sync causes the transactions and buffers to be synced to disk, or until another machine asks for the lock, at which point the data is synced to disk and the Glock is dropped to Not Held and the lock is released. This is important because it allows a GFS client to hold a Glock until another machine asks for it, and service multiple requests for the same Glock without making a separate lock request for each process.

3.5 GFS and Fibre Channel Documentation in Linux

We have developed documentation for GFS over the last year. Linux HOWTOs on GFS and Fi-

bre Channel can be found at the GFS web page: http://www.globalfilesystem.org. In addition, there are conventional man pages for all the GFS and Pool Volume Manager utility routines, including mkfs, ptool, passemble, and pinfo[14].

4 File System Journaling and Recovery in GFS

To improve performance, most local file systems cache file system data and metadata so that it is unnecessary to constantly touch the disk as file system operations are performed. This optimization is critical to achieving good performance as the latency of disk accesses is 5 to 6 orders of magnitude greater than memory latencies. However, by not synchronously updating the metadata each time a file system operation modifies that metadata, there is a risk that the file system may be inconsistent if the machine crashes.

For example, when removing a file from a directory, the file name is first removed from the directory, then the file dinode and related indirect and data blocks are removed. If the machine crashes just after the file name is removed from the directory, then the file dinode and other file system blocks associated with that file can no longer be used by other files. These disk blocks are now erroneously now marked as in use. This is what is meant by an inconsistency in the file system.

When a single machine crashes, a traditional means of recovery has been to run a file system check routine (fsck) that checks for and repairs these kinds of inconsistencies. The problem with file system check routines is that (a) they are slow because they take time proportional to the size of the file system, (b) the file system must be off-line while the fsck is being performed and, therefore, this technique is unacceptable for shared file systems. Instead, GFS uses a technique known as file system journaling to avoid fsck's altogether and reduce recovery time and increase availability.

4.1 The Transaction Manager

Journaling uses transactions for operations that change the metadata state. These operations must be atomic, so that the file system moves from one consistent on-disk state to another consistent on-disk state. These transactions generally correspond to VFS operations such as create, mkdir, write, unlink, etc. With transactions, the file system metadata can always be quickly returned to a consistent state.

A GFS journaling transaction is composed of the metadata blocks changed during an atomic operation. Each journal entry has one or more locks associated with it, corresponding to the metadata protected by the particular lock. For example, a creat() transaction would contain locks for the directory, the new dinode, and the allocation bitmaps. Some parts of a transaction may not directly correspond to on-disk metadata.

All metadata blocks contain a generation number that is incremented each time it is changed, and that is used in recovery.

A transaction is created in the following sequence of steps:

1. start transaction
2. acquire the necessary Glocks
3. check conditions required for the transaction
4. pin the in-core metadata buffers associated with the transaction (i.e., don't allow them to be written to disk)
5. modify the metadata
6. pass the Glocks to the transaction
7. commit the transaction by passing it to the Log Manager

To represent the transaction to be committed to the log, the Log Manager is passed a structure which contains a list of metadata buffers. Each buffer knows its Glock number. Passing this structure represents a commit to the in-core log.

4.2 The Log Manager

The Log Manager is separate from the transaction module. It takes metadata to be written from the transaction module and writes it to disk. The Transaction Manager pins, while the Log Manager unpins. The Log Manager also manages the Active Items List, and detects and deals with Log wrap-around.

For a shared disk file system, having multiple clients share a single journal would be too complex and inefficient. Instead, as in Frangipani [4], each GFS client gets its own journal space, that is protected by one lock that is acquired at mount time and released at unmount (or crash) time. Each journal can be on its own disk for greater parallelism. Each journal must be visible to all clients for recovery.

In-core log entries are committed asynchronously to the on-disk log. The Log Manager follows these steps:

1. get the transaction from the Transaction Manager
2. wait and combine this transaction with others (asynchronous logging)
3. perform the on-disk commit
4. put all metadata in the Active Items List
5. unpin the metadata
6. later, when the metadata is on disk, remove it from the Active Items List

Recall that all transactions are linked to one or more Glocks, and that Glocks may be requested by other machines during a callback operation. Hence, callbacks may result in particular transactions being pushed out of the in-core log and written to the on-disk log. Before a Glock is released to another machine, the following steps must be taken:

1. transactions dependent on that Glock must be flushed to the log
2. the in-place metadata buffers must be synced
3. the in-place data buffers must be synced

Only transactions directly or indirectly dependent on the the requested Glock need to be flushed. A journal entry is dependent on a Glock if either (a) it references that Glock directly, or (b) it has Glocks in common with transactions which reference that Glock directly.

For example, in Figure 4, five transactions in sequential order (starting with 1) are shown, along with the Glocks upon which each transaction is dependent. If Glock 6 is requested by another machine, transactions 1, 2, and 5 must be flushed to the on-disk log in order. (Because transactions involving overlapping glocks are combined as they are commited to the in-core log, transaction 3 will be written out as well. It's not strictly necessary, though.) Then the in-place metadata and data buffers must be synced for Glock 6, and finally Glock 6 is released.

4.3 Recovery

Journal recovery is initiated by clients in two cases:

- a mount time check shows that any of the clients were shutdown uncleanly or otherwise failed
- a locking module reports an expired client when it polls for expired machines

In each case, a recovery kernel thread is called with the expired client's ID. The machine then attempts to begin recovery by acquiring the journal lock of a failed client. A very dangerous special case can result when a client (known as a zombie) fails to heartbeat its locks, so the other machines think it is dead, but it is still alive; this could happen, for example, if for some reason the "failed" client temporarily was disconnected from the network. This is dangerous because the supposedly failed client's journal will be recovered by another client, which has a different view of the file system state. This "split-brain" problem will result in file system corruption. For this reason, the first step in recovery after acquiring the journal lock of a failed client is to prevent the failed machine from writing to the shared device. This operation is called I/O Fencing.

There are several methods for I/O Fencing.

- *Network Power Switch* – a machine connects to a power switch over IP and asks it to cycle the power on a failed client
- *X10* – a machine can piggy-back signals onto of the in-the-wall-power to cause another machine's power to be cycled.
- *Persistent Reservation* – Some SCSI devices implement a command that allows one machine ask the device to ignore another machine.
- *Fibre Channel Zoning* – Some Fibre Channel switches support the capability to prevent the failed client from accessing the disks.

Glock #

Transaction	2	3	6	8	10	11
1	X	X				
2		X	X	X		
3	X	X				
4					X	X
5			X	X		

X represents in-memory metadata buffers which will be written to the journal

Figure 4: Journal Write Ordering Imposed by Lock Dependencies During GFS Lock Callbacks

GFS allows arbitrary I/O fencing methods to disable the failed clients access to the shared storage devices. There are currently modules that support all of the above methods except Persistent Reservation. (Persistent Reservation is only just starting to appear on SCSI devices.)

Once a client fences out the fail machine and obtains its journal lock, journal recovery proceeds as follows: the tail (start) and head (end) entries of the journal are found. Partially-committed entries are ignored. For each journal entry, the recovery client tries to acquire all locks associated with that entry, and then determines whether to replay it, and does so if needed. All expired locks are marked as not expired for the failed client. At this point, the journal is marked as recovered.

The decision to replay an entry is based on the generation number in the metadata found in the entry. When these pieces of metadata are written to the log, their generation number is incremented. The journal entry is replayed if the generation numbers in the journal entry are larger or equal to the in-place metadata.

Note that machines in the GFS cluster can continue to work during recovery unless they need a lock held by a failed client.

4.4 Comparison to Alternative Journaling Implementations

The main difference between journaling in a local file system and GFS is that GFS must be able to flush out transactions in an order other than that in which they were created. A GFS client must be able to respond to callbacks on locks from other clients in the cluster. The client should then flush only the transactions that are dependent on that lock. This means that GFS doesn't automatically combine transactions as they are commited in-core. They are only combined if they share glocks.

4.5 Cluster Configuration

There are a number of identifiers that each member of the cluster needs to know about all the other members: Hostname, IP address, Journal Number, and the I/O fencing method. The mappings for each host in the cluster are stored outside the file system by the lock module. For the DMEP lock module, these values are stored on a block device readable by all machines in the cluster.

4.6 Online Growing of File Systems

It is important to able to add more storage space to a running cluster. When A new feature of GFS 4 is that it now supports the ability to grow the file system on-line. One feature of FC is that disks can be added to a Storage Area Network (SAN). Once that is done the disks can be added to the file system's Pool. The process of growing the Pool happens in three steps:

1. Labels describing how the new disks fit into the Pool are written to the new disks.
2. An atomic write to the label on the first disk in the Pool sets the Pool to its new size.
3. At this point, the Pool can be reassembled and the new space can be used.

One interesting feature of Pool is that step 3 doesn't need to happen on all the machines right away. The reassembly to access the new storage space only happens when

that new space is needed. As the Pool driver in each machine maps block request to individual disks, it looks for block numbers that doesn't exist in the current Pool. When it sees that the block number being accessed is too large, the Pool labels are reread from the disks and the new, bigger Pool is reassembled. Then GFS can be informed that new space is available and it can be added to the file system. The new space takes the form of new Resource Groups at the end of the file system. The locations of these new resource groups are written to the end of the Resource Index hidden dinode.

Again, the growth of the GFS file system is automatically detected by other machines in the cluster. During normal operation all the machines hold a shared lock on the Resource Index dinode. The process growing the file system acquires that lock exclusively when adding the new resource groups to the dinode. After that, each machine can reacquire its shared lock on the dinode and reread the new index. Each machine can then access the new storage space.

5 Scalability

Figure 5 shows one to sixteen GFS hosts being added to a constant size file system and each performing a workload of a million random operations. (These results are fore a previous non-journaled version of GFS.) These sixteen machines were connected across a Brocade Fabric fabric to 8 8-disk enclosures, each configured as a single 8-disk loop. The workload consisted of 50 percent reads, 25 percent appends/creates and 25 percent unlinks. Each machine was working in in its own directory and the directories were optimally placed across the file system. Notice that the scalability curve shows nearly perfect speedup. These new results compare favorably with the dismal scaling results obtained for the early versions of GFS [6], which didn't cache locks, file data, or file system metadata.

6 Conclusions and Future Work

In this paper, we described the GFS journaling and recovery implementation and other improvements in GFS version 4 (GFS-4). These include a lock abstraction and network block driver layer, which allow GFS to work with almost any global lock space or storage networking media. The new DMEP specification simplifies the work required by SCSI storage vendors, and allows the lock semantics to be refined over time. In addition, a variety of other changes to the file system metadata and pool volume manager have increased both performance and flexibility. Taken together, these changes mean that GFS can now enter a beta test phase as a prelude to production use. Early adopters who are interested in clustered file systems for Linux are encouraged to install and test GFS to help us validate its performance and robustness.

With the work on journaling and recovery complete, we intend to look at several new features for GFS. These include file system versioning for on-line snapshots of file system state using copy-on-write semantics. File system snapshots allow a slightly older version of the file system to be backed up on-line while the cluster continues to operate. This is important in high-availability systems.

References

[1] L. McVoy and S. Kleiman. Extent-like performance from a unix file system. In *Proceedings of the 1991 Winter USENIX Conference*, pages 33–43, Dallas, TX, June 1991.

[2] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice-Hall, 1996.

[3] Roy G. Davis. *VAXCluster Principles*. Digital Press, 1993.

[4] Chandramohan Thekkath, Timothy Mann, and Edward Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[5] Matthew T. O'Keefe. Shared file systems and fibre channel. In *The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*, pages 1–16, College Park, Maryland, March 1998.

[6] Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Tom Ruwart. The design and performance of a shared disk file system for IRIX. In *The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*, pages 41–56, College Park, Maryland, March 1998.

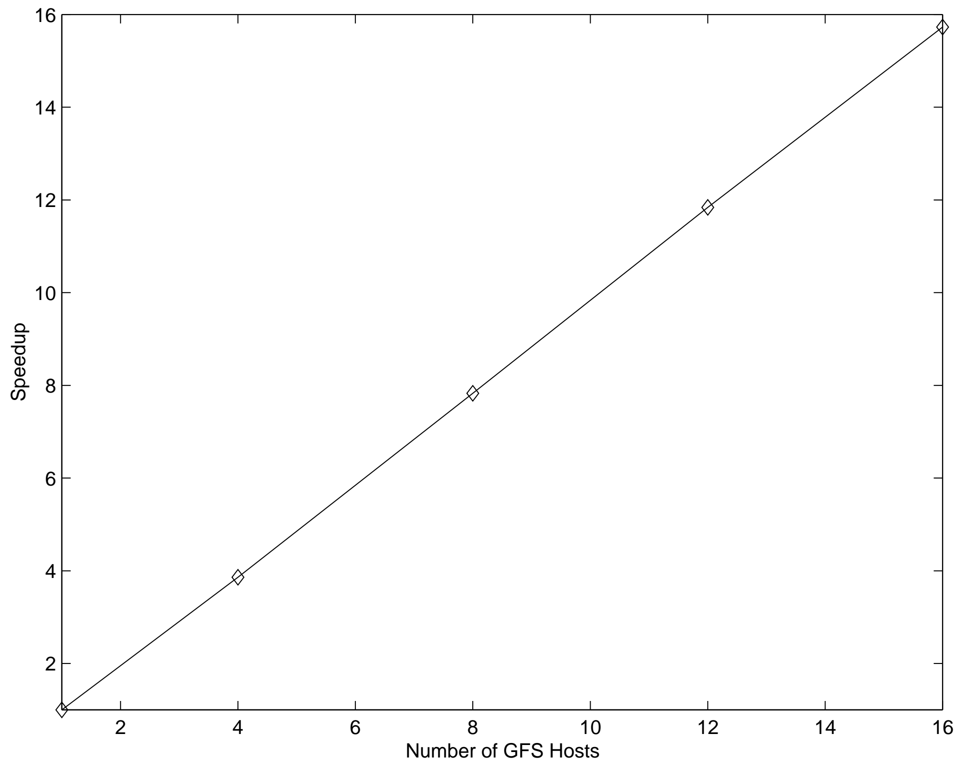


Figure 5: Sixteen machine speedup for independent operations

[7] Kenneth W. Preslan et al. A 64-bit, shared disk file system for linux. In *The Seventh NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Sixteenth IEEE Symposium on Mass Storage Systems*, pages 22–41, San Diego, CA, March 1999.

[8] Alan F. Benner. *Fibre Channel: Gigabit Communications and I/O for Computer Networks*. McGraw-Hill, 1996.

[9] N. Kronenberg, H. Levy, and W. Strecker. VAXClusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(3):130–146, May 1986.

[10] K. Matthews. Implementing a Shared File System on a HiPPi disk array. In *Fourteenth IEEE Symposium on Mass Storage Systems*, pages 77–88, September 1995.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Global File System. In *The Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, volume 2, pages 319–342, College Park, Maryland, March 1996.

[12] The GFS Group. Device memory export protocol specification. http:// www.globalfilesystem.org/ Pages/ dmep.html, August 2000.

[13] Ken Preslan et al. Dlock 0.9.6 specification. http:// www.globalfilesystem.org/ Pages/ dlock.html, May 2000.

[14] Mike Tilstra et al. The gfs howto. http://www.globalfilesystem.org /howtos/ gfs_howto.

[15] David Teigland. The pool driver: A volume driver for sans. Master's thesis, University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, October 1999. http://www.globalfilesystem.org/ Pages/ theses.html.

[16] Manish Agarwal. A Filesystem Balancer for GFS. Master's thesis, University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, June 1999. http://http://www.globalfilesystem.org /Pages/ theses.html.

[17] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing –

a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[18] Michael J. Folk, Bill Zoellick, and Greg Riccardi. *File Structures*. Addison-Wesley, March 1998.

[19] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, San Diego, CA, January 1996.

All GFS publications and source code can be found at `http://www.globalfilesystem.org`.