Implementing Journaling in a Linux Shared Disk File System

Kenneth W. Preslan - Sistina Software, Inc.

**Andrew Barry, Jonathan Brassow, Russell Cattelan,
Adam Manthei, Erling Nygaard, Seth Van Oort,
David Teigland, Mike Tilstra, and Matthew O'Keefe**

Parallel Computer Systems Laboratory, Dept. of ECE, University of Minnesota
200 Union Street SE, Minneapolis, MN 55455
+1-612-626-7180, okeefe@borg.umn.edu
University of Minnesota

Grant Erickson - Brocade Communications

Manish Agarwal - VERITAS Software

Abstract

In computer systems today, speed and responsiveness is often determined by network and storage subsystem performance. Faster, more scalable networking interfaces like Fibre Channel and Gigabit Ethernet provide the scaffolding from which higher performance computer systems implementations may be constructed, but new thinking is required about how machines interact with network-enabled storage devices.

In this paper we describe how we implemented journaling in the Global File System (GFS), a shared-disk, cluster file system for Linux. Our previous three papers on GFS at the Mass Storage Symposium discussed our first three GFS implementations, their performance, and the lessons learned. Our fourth paper describes, appropriately enough, the evolution of GFS version 3 to version 4, which supports journaling and recovery from client failures.

In addition, GFS scalability tests extending to 8 machines accessing 8 4-disk enclosures were conducted: these tests showed good scaling. We describe the GFS cluster infrastructure, which is necessary for proper recovery from machine and disk failures in a collection of machines sharing disks using GFS. Finally, we discuss the suitability of Linux for handling the big data requirements of supercomputing centers[1].

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

[1]The work by Grant Erickson and Manish Agarwal on GFS was performed while they were at the University of Minnesota.

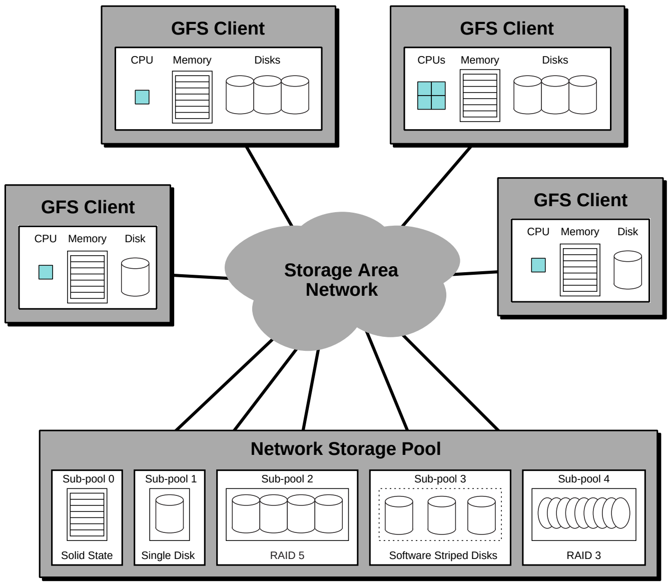

Figure 1: A Storage Area Network

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.

We began development of our own shared file system, known as GFS-1 (the Global File System, version 1), in the summer of 1995. At that time, we were primarily interested in exploiting Fibre Channel technology to post-process large scientific datasets [11] on Silicon Graphics (SGI) hardware. Allowing machines to share devices over a fast Fibre Channel network required that we write our own shared file system for IRIX (SGI's System V UNIX variant), and our initial efforts yielded a prototype described in [12]. This implementation used parallel SCSI disks and SCSI reserve and release commands for synchronization. Reserve and release locked the whole device, making it impossible to support simultaneous file metadata accesses to a disk. Clearly, this was unacceptable.

This bottleneck was removed in our second prototype, known as GFS-2, by developing a fine-grain lock command for SCSI. This prototype was described in our 1998 paper [6] and the associated thesis [13]; we also described our performance results across four clients using a Fibre Channel switch and RAID-3 disk arrays. Performance did not scale past three clients due to lock contention and the lack of client caching. In addition, very large files were required for good performance and scalability because neither metadata (or locks) nor file data were cached on the clients.

By the spring of 1998, we began porting our code to the open source Linux operating system. We did this for several reasons, but the primary one was that IRIX is closed source, making it very difficult to cleanly integrate GFS into the kernel. Also, Linux had recently acquired 64-bit, SMP, and floating-point support on Digital Equipment Corporation (DEC) Alpha platforms that were adequate for our computing needs.

In addition, we shed our narrow focus on large data applications and broadened our efforts to design a general-purpose file system that scaled from a single desktop machine to large clusters of machines enabled for device sharing. Because kernel source was available, we could finally support metadata and file data caching, but this required changes to the lock specification, detailed in the 0.9.4 device lock specification [14], [15].

The GFS port to Linux involved a complete re-write of GFS-2, resulting in a new version we call GFS-3 [7]. In addition to support for caching, GFS-3 supported leases on locks which time out if a GFS machine fails to heartbeat the lock. Client IDs are returned with the lock so that callbacks can be made to request that clients release metadata. GFS-3 used extendible hashing for the directory data structure to allow large numbers of files in a single directory. GFS-3's scalability has been measured up to 8 machines and 8 disk units with no measurable interference between machines making independent file accesses and disk requests across the storage network. At this point, we believe that there are no significant limits to GFS scalability.

However, though GFS-3 fixed most of the performance and scalability issues that had arisen for the previous versions of GFS, it did not address a critical requirement for shared disk cluster file systems: fault-tolerance. A production-quality shared file system must be able to withstand machine, network, or shared disk failures without disrupting continued cluster processing [3]. To this end, we have now implemented file system journaling in the latest version of GFS, known as GFS-4. When a GFS machine fails and the failure is detected, the remaining clients in the cluster may recover for the failed machine by replaying its journal. With the addition of journaling, GFS is now ready for consideration for deployment in production environments, after a reasonable beta test phase. We discuss the use of Linux and GFS in processing large datasets later in this paper.

In the following sections we describe GFS-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-5, including file system resizing.

2 GFS Background

For a complete description of GFS-3 see [7], for GFS-2 see [6], and for GFS-1 see [12]. In this section we provide a summary of the key features of the Global File System.

2.1 Dlocks

Device Locks are mechanisms used by GFS to synchronize client access to shared metadata. They help maintain metadata coherence when metadata is cached by several clients. The locks are implemented on the storage devices (disks) and accessed with the SCSI device lock command we call *Dlock* [16], [17], [15]. The Dlock 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. The file system provides a mapping between files and Dlocks.

GFS-3 used Dlock version 0.9.4 [15], which included timeouts on a per lock basis, multiple reader/single writer semantics, and inclusion of lock-holding client ID information in the SCSI reply data. The latest, journaled version of GFS (version 4) uses Dlock version 0.9.5 [17], which includes some new features we describe in the following sections. Both the 0.9.4 and 0.9.5 Dlock specifications have been implemented as daemon processes that can executed on any machine on the network. A machine that runs a Dlock daemon process is called a Dlock server. A Dlock server can provide provide Dlock functionality in systems constructed with storage devices that do not have SCSI Dlock support [18].

2.1.1 Expiration

In a shared disk environment, a failed client cannot be allowed to indefinitely hold whatever locks it held when it failed. Therefore, each holder must continually update a timer on the disk. If this timer ever expires, other lock holders may begin error recovery functions to eventually free the lock. Expiration is alternately referred to as timing-out, and the act of updating the timer is often referred to as heartbeating the timer or the timer-device. In version 0.9.4, time-outs were per lock; in 0.9.5, they are per-client.

2.1.2 Client IDs

The Client ID is a unique identifier for each client. The client id is completely opaque to the Dlock device. In GFS the client ID is used both as an identifier and to store the IP address of the client, allowing inter-machine communication. The Client ID can be any arbitrary 32-bit number that uniquely identifies a machine.

2.1.3 Version Numbers

Associated with every lock is a version number. Whenever the data associated with a lock is changed, the version number is incremented. Clients may use cached data instead of re-reading from the disk as long as the version number on the dlock is unchanged since the data was last read. The drawback with version numbers is that a client must still read the version number (which is located on the dlock storage device or dlock server); this is often a high-latency operation (even simple SCSI commands that do not touch the disk often require at least 1 millisecond).

Version numbers are an optional dlock feature, and are are unused in GFS-4, which relies instead on callbacks to keep cached metadata consistent. Version numbers may be removed in a future version of the device lock specification.

2.1.4 Conversion Locks

The conversion lock is a simple one stage queue used to prevent writer starvation. In Dlock version 0.9.4, 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.

2.1.5 Enable

In the event that a lock device is turned off and comes back on, all the locks 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 dlock commands when they are first powered up. The devices should return failure results, with the enabled bit of the dlock reply data format cleared, to all dlock actions except refresh timer until a dlock enable 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 dlock 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.

2.2 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 Dlock commands to specific devices at the SCSI driver layer [19]. 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. Sub-pools are an internal construction which does not affect the high level view of a pool[2] 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 [20]. 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.

There are two other volume managers available in Linux. Linux LVM (logical volume manager) was developed by Heinz Mauelshagen [21] and provides traditional volume manager functionality including volume resizing, on-line addition and deletion of volumes (both physical and logical levels), and on-line reallocation of physical disk space. Work is in progress to develop a volume snapshotting capability in Linux LVM. There is also a software RAID driver called MD developed by Ingo Mulnar that supports RAID levels 0, 1, 4, and 5 [22]. Pool does not support software RAID or volume resizing and virtualization, while neither Linux LVM nor MD support multiple clients accessing the same volume. Finding ways to integrate the functionality found in these different volume managers would be very useful.

2.2.1 Modular Block Drivers

Device drivers are the collection of low-level functions in the OS used to access hardware devices. Drivers can provide various levels of functionality. They range from directly manipulating hardware registers to providing a more abstract view of devices, making them easier to program [2]. The "I/O subsystem" refers to the mid-level drivers and OS routines which come between hardware-specific drivers and upper level system calls.

Low-level, device-specific driver functions include:

- Device initialization and control
- Interrupt handling
- Transferring bytes of data into and out of buffers

Mid- and upper-level driver functions include:

- Device naming
- Buffering and caching
- Providing a consistent, programmable interface

In Linux, file systems and drivers may be written as kernel modules. This allows them to be dynamically installed and removed from the kernel. Although not required, this makes development much easier because the entire kernel need not be recompiled and restarted to make a change in a module [23].

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

Pool is a mid-level block driver built atop the SCSI and FC drivers. This means it conforms to the standard block driver interfaces, but remains at a higher level of abstraction. The following list describes the basic functionality provided by Pool:

- *init_module, cleanup_module* are the functions required by Linux to be a kernel module. They basically call pool_init when the driver is added and free memory when it is removed.

- *pool_init* is the function called when the Pool module is installed in the kernel. It registers Pool in the global table of block drivers. It also initializes data structures maintained by Pool describing currently managed pools. Registering a block driver in the kernel includes specifying a major device number, a name (pool), and a set of functions used to access devices of this type.

- *pool_open, pool_release* are called for a specific pool after the open and close system calls on the pool's device node. Usage counts are incremented or decremented. These would be called due to a mount or I/O on the device node.

- *pool_read, pool_write* are called after a read or write system call on the pool's device node. They pass requests to the Linux routines `block_read()` and `block_write()`. These are not regularly used since the file system calls lower level routines directly.

- *pool_ioctl* is used to request Dlocks, get listings of currently configured pools, add new pools, remove pools, get basic size information or control debugging.

The following functions are internal pool routines called due to pool_ioctl requests:

- *add_pool* is called during *passemble* to configure a new pool. Information describing the new pool is provided by *passemble* and used to allocate new structures in the Pool driver. The parameters describe the pool, sub-pools, underlying physical devices, Dlocks, and striping. Most of the code is OS-independent and is handled in *gen_pool*. All lower-level devices are opened at this stage to verify they can be used. The minor number selected for the new pool is returned to user space.

- *remove_pool* removes a specific pool by closing underlying devices and freeing data structures. Pools are removed when the -r(emove) option is specified in *passemble*.

- *list_pool, list_spool* return descriptions of currently managed pools. This information is needed by the tools *passemble* and *pinfo*. This is also how mkfs_gfs (make a gfs file system command) gets sub pool and Dlock information when creating a new file system.

The last function, *pool_map*, is the most interesting and central to Pool's purpose. It is used to map requests from a logical pool device and block number to a real device and block number. The map functions for Pool and other volume managers are called in an unusual way in Linux. This is the topic of the next section.

```
if (MAJOR(bh->b_rdev) == POOL_MAJOR)
        pool_map(&bh->b_rdev, &bh->b_rsector);

else if (MAJOR(bh->b_rdev) == MD_MAJOR)
        md_map(&bh->b_rdev, &bh->b_rsector);

else if (MAJOR(bh->b_rdev) == LVM_MAJOR)
        lvm_map(&bh->b_rdev, &bh->b_rsector);
```

Figure 2: Pseudocode of mapping functions called directly.

2.2.2 Block Mapping

All reads and writes to block devices occur in chunks defined by the file system block size (usually 4 or 8 KB). Each of these block I/O requests is defined by a `buffer_head` structure (`bh`). All the `bh`'s are managed by the I/O subsystem and a `bh` for a specific request is passed through OS layers from the file system [3] down to the low level disk driver. Two `bh` fields are especially important in specifying the block:

- *rdev*: the device for this request (major, minor numbers)
- *rsector*: the block number on the device rdev

When using a volume driver, a `bh` comes from the file system with `rdev` equal to the *logical* device and `rsector` equal to the logical block. When the `bh` reaches the specific disk driver, `rdev` and `rsector` must specify a *real* device and block number. The I/O subsystem routine which processes the `bh` below the file system is `ll_rw_block()`. Here is where the specific volume manager mapping function is called to change `rdev` and `rsector`.

To call the correct volume manager mapping function (`pool_map, md_map,` or `lvm_map`), the original major number of `rdev` is checked because each volume manager has a unique major number. When the specific volume manager is identified, its map function can be called directly. Pseudocode illustrating this is in Fig. 2.

The method of calling specific map functions can be improved by making the code more general. Every block driver has an identification structure in the global block driver table `blk_dev[]` indexed by major number. The function pointer `map_fn` is added to `blk_dev_struct` as seen in Fig. 3. The function pointers are initially set to NULL for every driver. If a block driver has a map function, like `pool_map`, it will set `map_fn` to that function in the driver init routine.

The pseudocode segment in Fig. 2 can then be simplified as seen in Fig. 4. The `map_fn` field is compared to NULL. If the pointer is set, the driver for this block device has defined a map function which is then called through the function pointer. The map function itself

[3]The Linux kernel is moving away from using the buffer cache at these levels as the page cache will be used for most I/O. The buffer cache will be used at lower levels only.

```
typedef void (request_fn_proc)(void);
typedef int (makerq_fn_proc)(struct buffer_head *, int rw);
typedef int (map_fn_proc)(kdev_t, kdev_t *, unsigned long *,
                          unsigned long);
typedef struct request ** (queue_proc)(kdev_t dev);

struct blk_dev_struct {
       request_fn_proc         *request_fn;
       makerq_fn_proc          *makerq_fn;
       map_fn_proc             *map_fn;
       queue_proc              *queue;
       void                    *data;
       struct request          *current_request;
       struct request   plug;
       struct tq_struct plug_tq;
};
```

Figure 3: Entries in block device switch table.

```
dev = blk_dev + major;

if (dev->map_fn && dev->map_fn (bh[i]->b_rdev,
                                &bh[i]->b_rdev,
                                &bh[i]->b_rsector,
                                bh[i]->b_size >> 9)) {
       printk (KERN_ERR ‘‘Bad map in ll_rw_block’’);
       goto sorry;
}
```

Figure 4: Generic mapping in ll_rw_block()

determines the new `rdev` and `rsector` by using the number and size of each subpool and sub-device. The two fields in the `bh` are then rewritten.

The same method used for the map function is also used for the `make_request` function which is less common and used by volume drivers when doing mirroring. The `map_fn` and `make_request` functions will probably be combined in the future. Using function pointers for map and request routines was originally designed by Chris Sabol in the GFS group who also worked on the initial Pool port and Pool under IRIX.

The Linux framework for block drivers described above is different from the standard approach in other OS's. In the more common method, each block driver defines a *strategy* function as the single I/O entry point in the block device switch table. The file system and any volume drivers always call the strategy function of a buffer's device. Within a volume driver's strategy routine, mapping is done before calling the next driver's strategy routine.

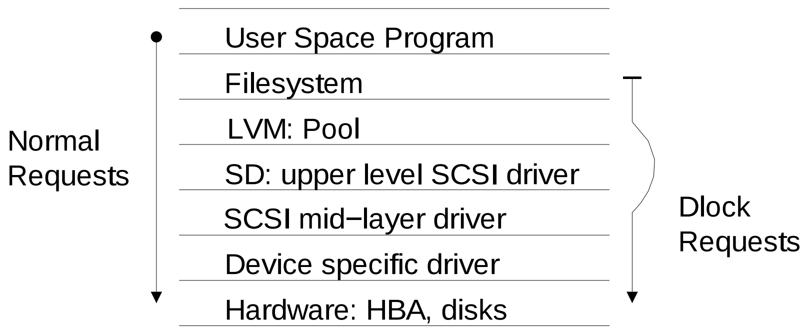

Figure 5: File System and driver layers

The Linux I/O subsystem was designed assuming no volume driver layer which is why mapping routines are called aside from the normal I/O path.

2.2.3 Dlock Support

A unique feature of the Pool driver required by the GFS locking layer is Dlock support. When a pool is configured, particular sub-devices are specified as supporting Dlocks. The Pool driver merges all the available Dlocks into a uniform Dlock space accessible by the GFS locking layer through pool_ioctl. When a Dlock is requested, Pool maps the logical Dlock number to an actual Dlock on a specific device. Because the highest level SCSI driver is not aware of the DLOCK command, Pool needs to construct the appropriate SCSI command descriptor block (CDB) and insert it into the SCSI mid-layer driver.

Pool's handling of Dlock commands is illustrated in Figure 5. There are three layers between the file system and the lowest level hardware driver. First is the volume manager which translates buffer addresses into real devices and block offsets. Not shown in the diagram is other I/O subsystem code which merges and queues buffer requests. Second is the upper level SCSI driver, SD, which breaks buffer requests into actual READ and WRITE SCSI commands. Next is the mid level driver which manages all SCSI devices and sends CDB's to the correct host bus adapters. Because SD deals only with transferring data with READ and WRITE it must be bypassed for other SCSI commands. Below this, the mid level driver sends many other CDB's and does not care about the specific command. This is naturally the level where DLOCK commands can be queued.

Interfacing Pool's Dlock code with the mid-layer SCSI driver[4] required a patch to the upper SD driver. SD maintains the array: `Scsi_Disk *rscsi_disks`. The array is indexed by minor numbers so the detailed structure of a SCSI device is easily accessible given the device number. Part of the `Scsi_Disk` structure is a required parameter to the `scsi_allocate_device()` routine. This allocate routine returns a `Scsi_Cmd` pointer which is passed into `scsi_do_cmd()`. The "do command" function sends the Dlock CDB. So a kernel patch (distributed with GFS and Pool) exporting the `rscsi_disks`

[4]The mid-layer SCSI driver code can be found in /usr/src/linux/drivers/scsi/scsi.c and the upper-level SCSI driver is /usr/src/linux/drivers/scsi/sd.c

symbol from the SD driver is required to access the appropriate `Scsi_Disk` given the minor number of the Dlock device.

Originally, the Pool driver handled all Dlock retries, timeouts, activity monitoring and resets. All the code implementing this was rewritten when moving to Linux, making it simpler and eliminating some incorrect behavior. Eventually as the Dlock specification became more complex, all these functions were moved into a separate locking module. In the current implementation, Pool only maps Dlock requests and sends them to devices.

2.3 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 client parallelism and 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 number of journals and resource groups, the dinode numbers of the three 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 three 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

3) The configuration space dinode – This holds configuration information that is used by the locking modules and transparent to GFS. The Dlock/Dlip modules use it to store namespace information about the cluster. (This is necessary for our ”Dlock” cluster infrastructure.)

There are four identifiers that each member of the cluster needs to know about all the other members: Hostname, IP address, Journal ID Number, and Client ID number. The quartets for each host in the cluster are stored in the config space dinode and passed to the lock module as it is initialized. What format the data is in is up to the lock module. The data will be written to the config file using a GFS ioctl call or standard write calls.

This data is stored in files because it needs to be able to grow as the filesystem grows. In previous versions of GFS, we just allocated a static amount of space at the beginning of the filesystem for the Resource Index metadata, but this will cause problems when we expand the filesystem 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 [24], [7], [25] 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.

2.4 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.

2.5 Flat File Structure

GFS uses a flat pointer tree structure as shown in Figure 6. 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 [26]. The current structure of the GFS metadata is an implementation choice and these alternatives are worth exploration in future versions of GFS.

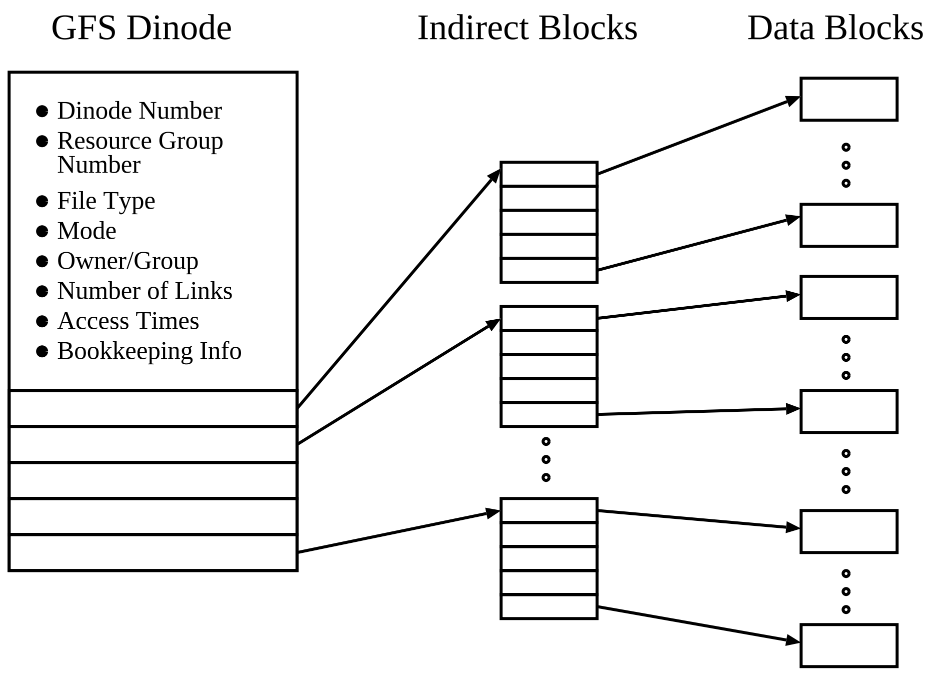

Figure 6: A GFS dinode. All pointers in the dinode have the same height in the metadata tree.

3 Improvements in GFS Version 4

Since our presentation at the IEEE/NASA Mass Storage Symposium last year, there have been many improvements to GFS. We describe some of these improvements 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.

3.2 Fibre Channel in Linux

Until the summer of 1999, Fibre Channel support in Linux was limited to a single machine connected to a few drives on a loop. However, significant 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 well-supported in Linux, with multiple GPL'ed drivers written by QLogic and independent open source software developers. During testing in our laboratory with large Fabrics (32 ports) and large numbers of drives and GFS clients, the Fibre Channel hardware and software has performed well. Recent reductions in adapter card and switch costs have made it possible to cost-effectively build large, Fibre-Channel-based storage networks in Linux.

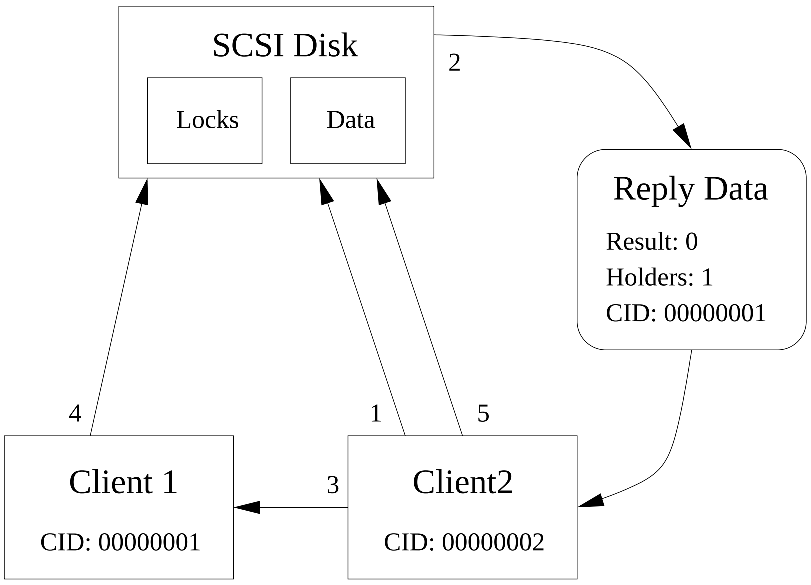

Figure 7: Callbacks on glocks in GFS

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 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 is possible to boot Linux from a GFS file system. In addition, GFS 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 provide building blocks for implementing 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 Dlocks. This is important because it allows GFS cluster architects to buy any disks they like, not just disks that contain Dlock firmware.

The GFS lock abstraction allows GFS clients to implement callbacks, as shown in Figure 7. When client 2 needs a dlock exclusively that is already held by client 1, client 2 first sends it's normal dlock SCSI request to the disk drive (step 1 in the figure). This request fails and returns the list of holder ClientIDs, which happens to be client 1 (step 2). Client 2 sends a callback to client 1, asking B to give up the lock (step 3). Client 1 syncs all dirty (modified) data and metadata buffers associated with that dlock to disk (step 4), and releases the dlock, incrementing the version number if any data has been written. Client A may then acquire the dlock (step 5).

Because clients can communicate with each other, they may hold dlocks indefinitely if no other clients choose to read from inodes associated with dlocks that are held. As long as a client holds a dlock, it may cache any writes associated with the dlock. 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 Linux file system systems across all workloads, including small file workloads.

In GFS-4, write caching is write-back, not write-through. GFS uses Global Locks (glocks), which may or may not be dlocks. GFS uses interchangeable locking modules, some of which map glocks to Dlocks. Other locking methods, such as a distributed lock manager [9] or a centralized lock server, can also be used. Our group has developed a centralized lock server known as the GLM (Global Locking Module) [18]. 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. If another machine asks for the lock, the current holder will sync all the dirty 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 Dlock 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 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 Dlock 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 dlock 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 Fibre 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[18].

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 file system 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.

Two types of metadata buffers are involved in transactions: primary and secondary. Primary metadata includes dinodes and resource group headers. They contain a generation number that is incremented each time they are changed, and that is used in recovery. There must always be one piece of primary metadata for each lock in the transaction. Secondary metadata includes indirect blocks, directory data, and directory leaf blocks; these blocks do not have a generation number.

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, and its type (Dinode, RG Header, or Secondary Metadata). 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 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 collect more transactions (asynchronous logging)

(3) perform the on-disk commit

(4) put all metadata in the Active Items List

(5) unpin the secondary metadata

(6) later, when the secondary metadata is on disk, remove it from the Active Items List

(7) unpin the primary metadata

(8) later, when the primary metadata is on disk, remove it from the Active Items List

Recall that all journal entries 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 journal entries 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) journal entries dependent on that Glock must be flushed to the log

(2) the in-place metadata buffers must be synced

		Glock #			
		2	3	6	8
Journal Entry	1	X	X		
	2		X	X	X
	3	X	X		
	4			X	X

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

Figure 8: Journal Write Ordering Imposed by Lock Dependencies During GFS Lock Callbacks

(3) the in-place data buffers must be synced

Only journal entries 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 earlier journal entries which reference that Glock directly.

For example, in Figure 8, four journal entries 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, journal entries 1, 2, and 4 must be flushed to the on-disk log in order. 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 several cases:

(a) a mount time check shows that any of the clients were shutdown uncleanly or otherwise failed

(b) a locking module reports an expired client when it polls for expired machines

(c) a client tries to acquire a Glock and the locking module reports that the last client to hold that Glock has expired

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 either (1) forcibly disable the failed client or, (2) fence out all IO from the client using the zoning feature of a Fibre Channel switch.

Once a client obtains the journal lock for a failed client, 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 primary 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 than 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 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 can't combine transactions into compound transactions until just before the transaction is committed to the disk.

When a GFS client unlinks a file from the directory structure, the file isn't actually deallocated until all clients have stopped using it. In order to determine which clients are using a given dinode, GFS must maintain an "nopen" count in each dinode. This is a counter of the clients that are using a dinode. When a client crashes, it leaves nopen references on all the dinodes that it was using. As part of recovery, the machine doing the recovery must determine which dinodes the failed client was using and decrement nopen count on those dinodes.

Hence, each GFS client maintains a list of all the dinodes it has nopen references on. Every time an inode is opened or closed, a marker is put in the journal describing the operation. Since the log can wrap many times during the time that a dinode is held by the client, this list is periodically re-logged in its journal.

GFS also has to label some metadata blocks with generation numbers that are incremented when transactions are committed. These generation numbers and the current state of the global locks are used to decide whether or not a given journal entry should be replayed during recovery.

As mentioned previously with respect to generation numbers, GFS has two types of metadata: Primary and Secondary. Primary metadata has version numbers and must be persistent on the disk – once the block is allocated as primary metadata, it can never be reused for real data or secondary metadata. Secondary metadata isn't subject to either of these two restraints.

One difference between Frangipani [4] and GFS is that Frangipani doesn't make a distinction between primary and secondary metadata. All Frangipani metadata is primary metadata. This is a good choice for Frangipani because of the unique nature of the Petal [27] block device underneath it. GFS is greatly simplified, however, by not having to maintain lists of unused indirect blocks, directory blocks, and other secondary metadata. The trade-off is that GFS has the extra constraint that secondary metadata must be flushed to the disk before any primary metadata for each compound transaction.

5 Performance Results

Figures 9 and 11 represent the current single client I/O bandwidth of Linux GFS (GFS-3 release Antimatter-Anteater was used for the tests. This release does not include journaling.) The tests were performed on a 533 MHz Alpha with 512 MB of RAM running Linux 2.2.13. The machine was connected to eight Seagate ST19171FC Fibre Channel drives on a loop with a Qlogic host adapter card. A 4,096-byte block size was used for these tests. (A block size of 8,192 bytes yields numbers that about 10 percent better, but this larger block size isn't available on all Linux architectures.) The Transfer Size axis represents the size of the file being transferred, whereas the Request Size represents the actual size of each file transfer request. So, for example, a 4096 MB file transferred using 4 requests yields a request size of 1024 MB and a transfer size of 4096 MB.

The bandwidth of first time creates, shown in Figure 9, peaks at around 55 MB/s. The read bandwidth shown in Figure 11 peaks at about 45 MB/s.

The GFS-3 single client performance can be compared to our previous GFS-2 single client performance numbers reported in [7] and shown in Figures 10 and 12. The machine configurations for these tests were essentially the same, although we used Linux kernel 2.2.0-pre7 for the GFS-2 tests. GFS-2 read bandwidth peaked out at 42 MB/s while GFS-3 reads peaked out at 48 MB/s. (notice the scale differences in the figures between GFS-2 and GFS-3 results).

Though the peak bandwidths are relatively close, notice how GFS-3 read performance is much better for smaller request sizes. For GFS-3 creates, the maximum performance is 50 MB/s versus 18 MB/s for GFS-2. As in the read case, GFS-3 create performance is much higher at smaller request sizes than GFS-2 create performance.

The significant GFS-3 performance advantage comes from the fact that GFS-2 had only read buffer caching, whereas GFS-3 has both read buffer and lock caching. GFS-2 would need to check the lock (a separate SCSI request) each time a buffer was read or written, whereas GFS-3 uses callbacks to enable lock caching. As long as no other client needs the lock, no lock request is made to disk during buffer accesses. The performance improvement is even more pronounced for writes, since writes can nearly always be cached. The actual write to disk only occurs later during a periodic sync operation, or when the buffer cache is pressured by the kernel to release buffer space to applications.

Figure 13 is a comparison of the extendible hashing directory structure in GFS-3 to the linear directory structure of Ext2. The test involved creating a million entry directory. Creates per second were measured at regular intervals as the directory was filled. The GFS curve levels off because of un-cached hash tables. Even for large directory sizes (10s of thousands directory entries), GFS can create 100s of files per second. Fast directory operations for directories with thousands of files are necessary to support applications with millions of small files.

Figure 14 shows one to four GFS-3 hosts being added to a constant size file system and each performing a workload of a million random operations. These four machines were connected across a Brocade Fabric switch to 4 4-disk enclosures, each configured as a single 4-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 direc-

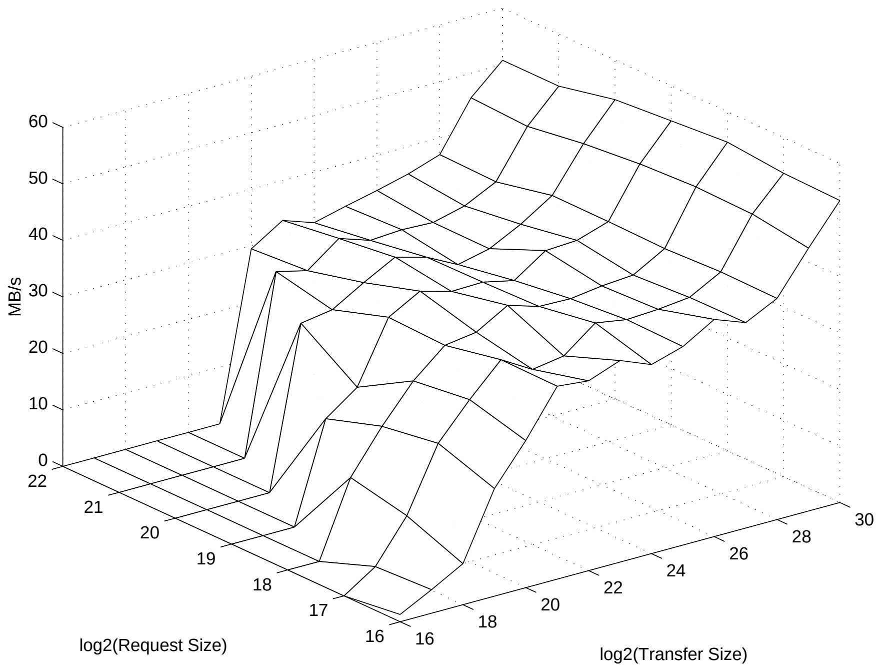

Figure 9: GFS-3 single machine create bandwidth

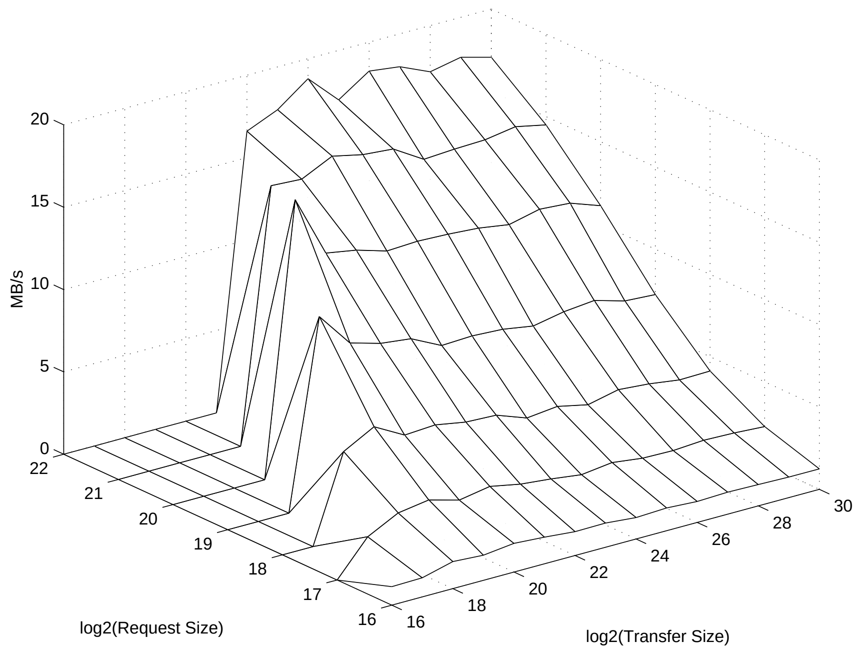

Figure 10: GFS-2 single machine create bandwidth

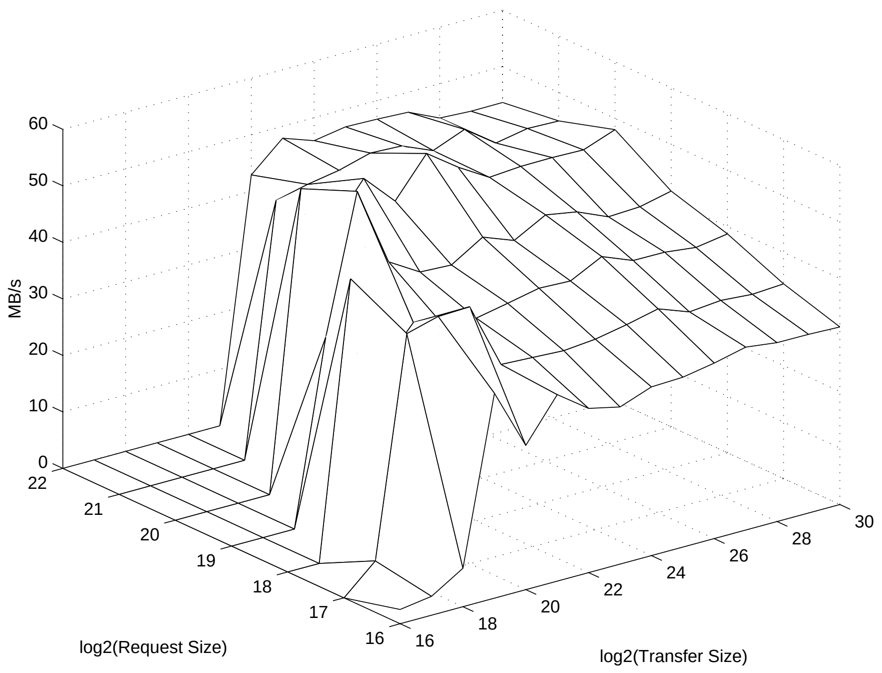

Figure 11: GFS-3 single machine read bandwidth

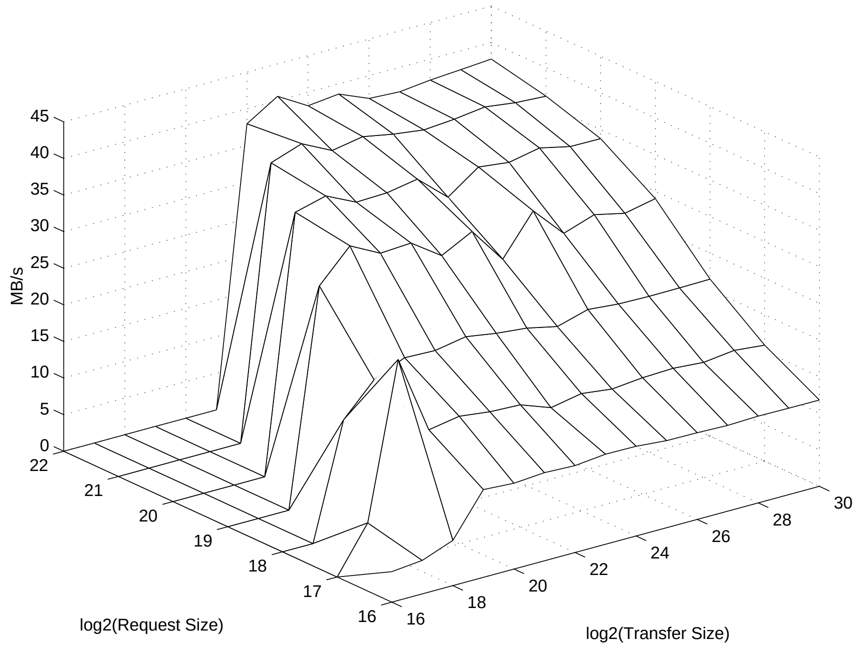

Figure 12: GFS-2 single machine read bandwidth

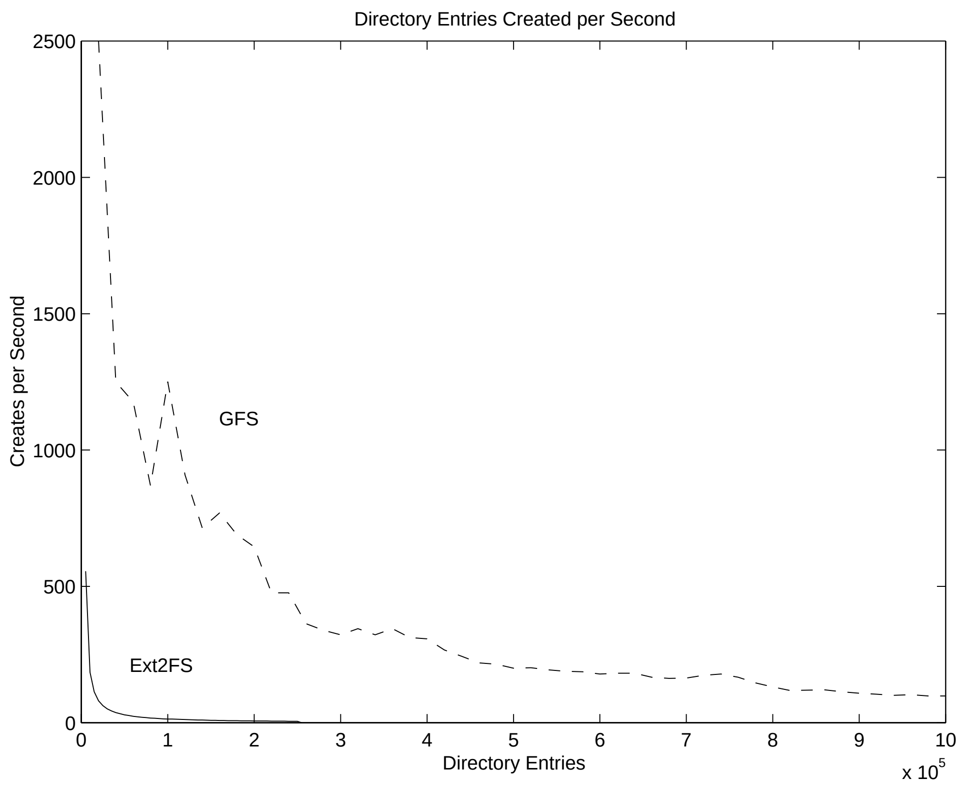

Figure 13: File creates per second versus directory size

tories were optimally placed across the file system. Notice that the scalability curve shows nearly perfect speedup. Similar results were achieved for an 8-way cluster. 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 Prospects for Linux in Big Data Environments

The ability of Linux to handle supercomputing workloads is improving rapidly. In the past year, Linux has gained 2 journaled file systems (ext3fs and Reiserfs) with two more on the horizon (XFS and GFS). SGI is porting its XFS file system to Linux. SGI has also open-sourced its OpenVault media management technology. Improvements in NFS client performance, virtual memory, SMP support, asynchronous and direct IO, and other areas will allow Linux to compete and surpass other UNIX implementations.

The open source nature of Linux provides better peer review on both architecture and code. Linux is free, and appears to be well on its way towards becoming the standard server operating system of the future. This means that most server applications will be ported to it in time, and that competition for Linux support and specialized services will develop.

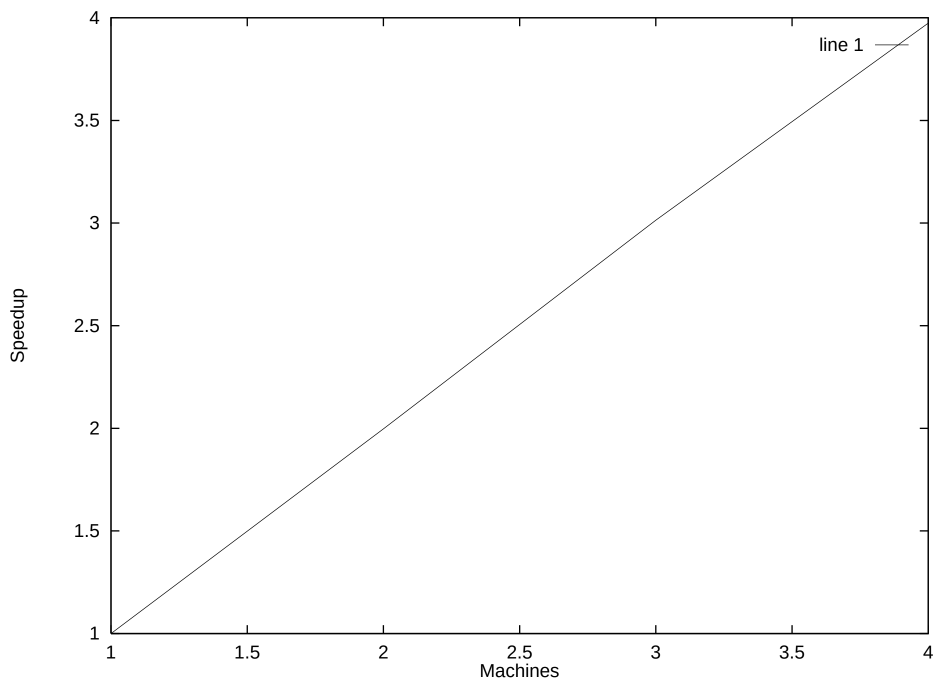

Figure 14: Four machine speedup for independent operations

7 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 lock specification (0.9.5) provides for better fairness and other improvements to support journaling and recovery. 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.

Once the work on journaling and recovery is complete, we intend to consider several new features for GFS. These may include file system versioning for on-line snapshots of file system state using copy-on-write semantics. File system snapshots allow an 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. Heinz Mauelshagen is implementing snapshotting in the Linux LVM volume manager [21], and so it may not be necessary to support this feature in GFS if we can use LVM to create GFS pools.

The ability to re-size the file system on-line is also very important, especially in storage area networks, where it will be quite common for new disks to be continually added to the SAN.

Finally, Larry McVoy, Peter Braam, and Stephen Tweedie are developing a scalable cluster infrastructure for Linux. This will include a Distributed Lock Manager (DLM) and mechanisms to detect and recover from client failures and cluster partitioning. This infrastructure could be very helpful in implementing recovery in GFS.

8 Acknowledgments

This work has been supported by NASA Ames Research Center through grant NAG2-1308, and by the Department of Defense High Performance Computing and Modernization Office (HPCMO) via Logicon contract number 9720954.

Many people have contributed code, ideas, and financial support to the GFS Group over the years. John Lekashman of NASA Ames provided both ideas and funding through NASA. Without John's support, we could never have gotten as far as we have with GFS.

Our industry sponsors include Brocade (Mark Sausville and Mike Lovas), Seagate (Mike Miller), EMC (Richard Wheeler), VERITAS Software (Alex Mirschenchenko), SGI (Jim Mostek and Dan Koren), and StorageTek (Aloke Guha and Jim Hughes). These sponsors helped make GFS possible with both funding and equipment.

Steve Soltis worked on early versions of GFS. Other people contributing ideas included Tom Ruwart, Alan Poston, Aaron Sawdey and Ben Gribstad. Jim Wayda of Dot Hill helped develop the 0.9.5 dlock specification, in particular the conversion lock approach. Tony Hecker of Seagate implemented the 0.9.4 specification in the latest Seagate Fibre Channel drives; Nate Larsen implemented th 0.6 dlock specification in an earlier generation of Seagate disk drives. Jim Coomes, Gerry Houlder, and Mike Miller of Seagate helped develop the device lock SCSI command.

Several Linux developers helped contribute implementation and architecture ideas to improve GFS, including Stephen Tweedie, Ted Ts'o, Peter Braam, and David Brower. Jim Mostek, Dan Koren, and Bill Earl of SGI helped us understand journaled file system implementations. Heinz Mauelshagen helped review this paper.

Chris Worley of Alta Technology has been our earliest and most aggressive early adopter, and we appreciate his GFS testing efforts. Tim Riker of Caldera, another aggressive early adopter, built a search engine using GFS. Paul Hayes, Laurie Miller and JD Harrington at the University of Minnesota used GFS to support their parallel electromagnetic simulations. Luke Lonergan and Greg Lindahl of HPTI installed GFS for evaluation on a Beowulf cluster at the Forecast Systems Lab in Boulder, Colorado. John Kaitschuck and Andy Hendrickson are evaluating a GFS cluster for supporting the NFS workload at Industrial Light and Magic. Tad Kollar is building a GFS cluster as part of a Beowulf machine at the NASA Glenn Research Center.

The authors would also like to acknowledge Jim Finlayson, who is on the Program Committee, for guiding the paper through to publication. Thanks also to Mike Declerck of NERSC for reviewing this paper and the GFS HOWTO, and for insights on the impact of real production use on several GFS features.

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.

[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] Aaron Sawdey, Matthew O'Keefe, and Wesley Jones. A general programming model for developing scalable ocean circulation applications. In *Proceedings of the 1996 ECMWF Workshop on the Use of Parallel Processors in Meteorology*, pages 209–225, Reading, England, November 1996.

[12] 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.

[13] Steven R. Soltis. *The Design and Implementation of a Distributed File System Based on Shared Network Storage*. PhD thesis, University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, August 1997.

[14] Matthew T. O'Keefe, Kenneth W. Preslan, Christopher J. Sabol, and Steven R. Soltis. X3T10 SCSI committee document T10/98-225R0 – Proposed SCSI Device Locks. http:// ftp.symbios.com/ ftp/ pub/ standards/ io/ x3t10/ document.98/ 98-225r0.pdf, September 1998.

[15] Ken Preslan et al. Dlock 0.9.4 specification. http:// www.globalfilesystem.org/Pages/dlock.html, December 1998.

[16] Kenneth W. Preslan et al. Scsi device locks version 0.9.5. In to appear in *The Eighth NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Seventeenth IEEE Symposium on Mass Storage Systems*, College Park, MD, March 2000.

[17] Ken Preslan et al. Dlock 0.9.5 specification. http:// www.globalfilesystem.org/Pages/dlock.html, December 1999.

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

[19] 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.

[20] 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.

[21] Heinz Mauelshagen. Linux lvm home page. http:// linux.msede.com/lvm.

[22] Ingo Mulnar. Linux software raid driver. http://ostenfeld.dk/j̃akob/Software-RAID.HOWTO/Software-RAID.HOWTO.html#toc4.

[23] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, 1998.

[24] 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.

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

[26] 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.

[27] Edward Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *ASP-LOS VII*, pages 84–92, San Diego, CA, October 1996.

[28] John Lekashman. Building and managing high performance, scalable, commodity mass storage systems. In *The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*, pages 175–179, College Park, Maryland, March 1998.

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