A 64-bit, Shared Disk File System for Linux

Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow,
Grant M. Erickson, Erling Nygaard, Christopher J. Sabol,
Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe

Parallel Computer Systems Laboratory
Department of Electrical and Computer Engineering
University of Minnesota

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 implementations may be constructed, but new thinking is required about how machines interact with network-enabled storage devices.

We have developed a Linux file system called GFS (the Global File System) that allows multiple Linux machines to access and share disk and tape devices on a Fibre Channel or SCSI storage network. We plan to extend GFS by transporting packetized SCSI commands over IP so that any GFS-enabled Linux machine can access shared network devices. GFS will perform well as a local file system, as a traditional network file system running over IP, and as a high-performance cluster file system running over storage networks like Fibre Channel. GFS device sharing provides a key cluster-enabling technology for Linux, helping to bring the availability, scalability, and load balancing benefits of clustering to Linux.

Our goal is to develop a scalable, (in number of clients and devices, capacity, connectivity, and bandwidth) server-less file system that integrates IP-based network attached storage (NAS) and Fibre-Channel-based storage area networks (SAN). We call this new architecture Storage Area InterNetworking (SAINT). It exploits the speed and device scalability of SAN clusters, and provides the client scalability and network interoperability of NAS appliances.

Our Linux port shows that the GFS architecture is portable across different platforms, and we are currently working on a port to NetBSD. The GFS code is open source (GPL) software freely available on the Internet at `http://gfs.lcse.umn.edu`.

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, hence 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]. 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*. [5], [6], [7], [8].

A shared file system approach based upon a 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 abil-

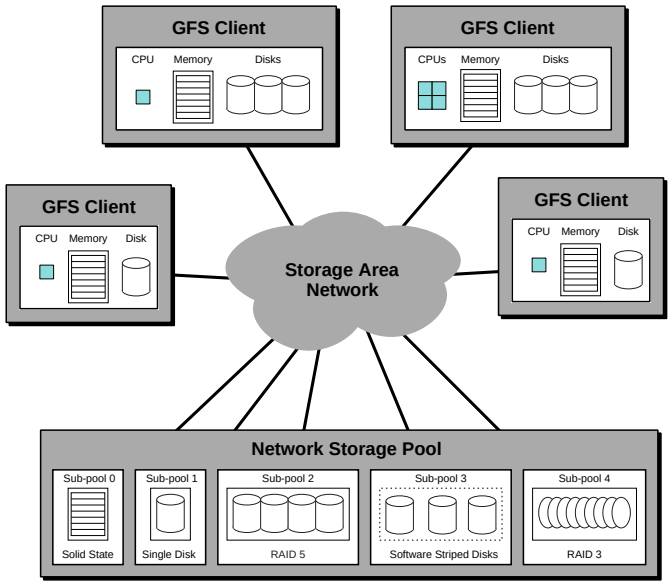

Figure 1: A Storage Area Network

ity 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.

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 [9] 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 [10]. 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 [4] and the associated thesis [11]; 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 heavy lock contention. In addition, very large files were required for good performance and scalability because neither metadata 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 and SMP support and on Digital Equipment Corporation (DEC) Alpha platforms was much faster and cheaper than our IRIX desktop machines.

In addition, we had shed our narrow focus on large data applications and had broadened our efforts to design a general-purpose file system that scaled from a single desktop machine to a large, heterogeneous network enabled for device sharing. Because we had kernel source we could finally support metadata and file data caching, but this required changes to the lock specification, detailed in [12]. This GFS port to Linux involved significant changes to GFS-2, so that we now refer to it as GFS-3. In the following sections we describe GFS-3 (which we will refer to as GFS in the remainder of this paper), the current implementation including the details of our Linux port, new scalable directory and file metadata data structures, preliminary performance results, and future work.

2 GFS Background

For a complete description of GFS-2 see [4] and for GFS-1 see [10]. In this section we provide a summary of the key features of the file system.

2.1 Dlocks

Device Locks are mechanisms used by GFS to facilitate mutual exclusion of file system metadata. They are also used to help maintain the coherence of the metadata when it is cached by several clients. The locks are implemented on the storage devices (disks) and accessed with the SCSI device lock command, *Dlock*. 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.

In the original specification [4], each Dlock is basically a test-and-set lock. A GFS client acquires a lock, reads data, modifies the data, writes the data back, and releases the lock. This allows the file system to complete operations on the metadata that are "atomic" with respect to other operations on the same metadata.

Each Dlock also has a "version number" associated with it. When a client wants to do a read-modify-write operation on a piece of metadata, it acquires the lock, does the read-modify-write, and releases the lock using the *unlock increment* action. When a client just wants to read metadata, it acquires the lock, reads the metadata, and releases the lock using the *unlock* action. If all clients follow this scheme, consistency can be checked by comparing the version number returned by a lock action with the value of the version number when the lock was previously held. If the version numbers are the same, no client modified the data protected by the lock and it is guaranteed to be valid. Version numbers were also used for caching in the distributed lock manager of the Vaxcluster [6].

2.2 The Network Storage Pool

The network storage pool (NSP) volume driver supports the abstraction of a single unified storage address space for GFS clients. The NSP is implemented in a device driver layer on top of the basic SCSI device and Fibre Channel drivers. This driver translates from the logical address space of the file system to the address space of each device. Subpools divide NSPs into groups of similar device types which inherit the physical attributes of the underlying devices and network connections.

2.3 Resource Groups

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 and 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. (The superblock may be replicated to improve performance and ease recovery.) This information includes 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 contains a static index of the resource groups which describes the location of each resource group and other configuration information.

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.

Consider a file system block size of 4 KB and assume the dinode header information requires 128 bytes. Without stuffing, a 1-byte file requires a total of 8 KB and at least 2 disk transfers to read the dinode and data block. With stuffing, a 1-byte file only requires 4 KB and one read request. The file can grow to 4 KB minus 128 bytes,

or 3,968 bytes, before GFS unstuffs the dinode.

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 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. The current structure of the GFS metadata is an implementation choice and these alternatives are worth exploration in future research.

3 GFS on Linux

Work on the Global File System started on SGI's IRIX operating system. IRIX is optimized for a big data environment and provides a lot of tools needed to develop GFS. The two things that IRIX lacks are kernel interface documentation and easily available kernel source.

In order for the file system module to interact with other parts of kernel, the writer of the file system needs to understand the interfaces to those other parts of the kernel. This understanding is easily achieved in one of two ways: reading the documentation of the interfaces or, if no documentation exists, reading the source that implements the interface. (Some have argued that source code is the only true documentation.) Neither of these options are available for IRIX.

A open source operating system, like Linux, is ideal for developing new kernel code. The source code is freely available. All kernel interfaces can be understood with a little bit of examination and cogitation. Because the source code is freely available, documentation can be written by third parties [13], [14]. We expect that whatever shortcomings Linux currently has with respect to manipulating large data sets will be overcome with time. GFS development is now focused primarily on Linux.

3.1 IRIX vs Linux

There are big differences between the IRIX and Linux Virtual File System (VFS) layers. IRIX uses the standard SVR4 VFS/Vnode interface, while Linux uses a homegrown approach.

3.1.1 VFS caching

Both VFS layers provide (roughly) the same set of system calls and make similar requests to the file system specific code. Their approach is different, though. The SVR4 VFS layer was planned to support networked file systems from the start [15]. In contrast, the Linux file system is more oriented towards optimizing local file systems.

The boundary between the IRIX/SVR4 VFS layer and the file system specific code is very clean. Every time the VFS layer needs information from the file system specific layer, it makes a function call to the file system dependent layer for that information. It remembers almost nothing about previous requests.

This is very good for a networked file system. One machine can change data in the file system without worrying about other machine's VFS layers caching that data. The VFS layer always asks the file system specific layer when it wants information. The file system specific layer can always provide the most up to date metadata.

The Linux VFS layer, on the other hand, knows a lot about the files it is accessing. It has its own copies of the file size, permissions, link count, etc. For local file systems, this works great. All disks accesses go though the VFS layer anyway, so the VFS layer might as well cache the data as it goes by. Local file systems can be very quick because the VFS avoids the overhead of calling the necessary function and waiting for the file system specific layer to locate and encode the requested information. It just reads the data from its own copy. Local file systems can also be simpler than their SVR4 counterparts. The Linux VFS layer does permission checking automatically. The writer of the local file system doesn't need to be as concerned with the intricate details of how UNIX manages permissions.

However, this makes designing and implementing a network file system more difficult in Linux. Uncontrolled caching in a networked file system, especially a shared-disk file system, can result in data inconsistencies between

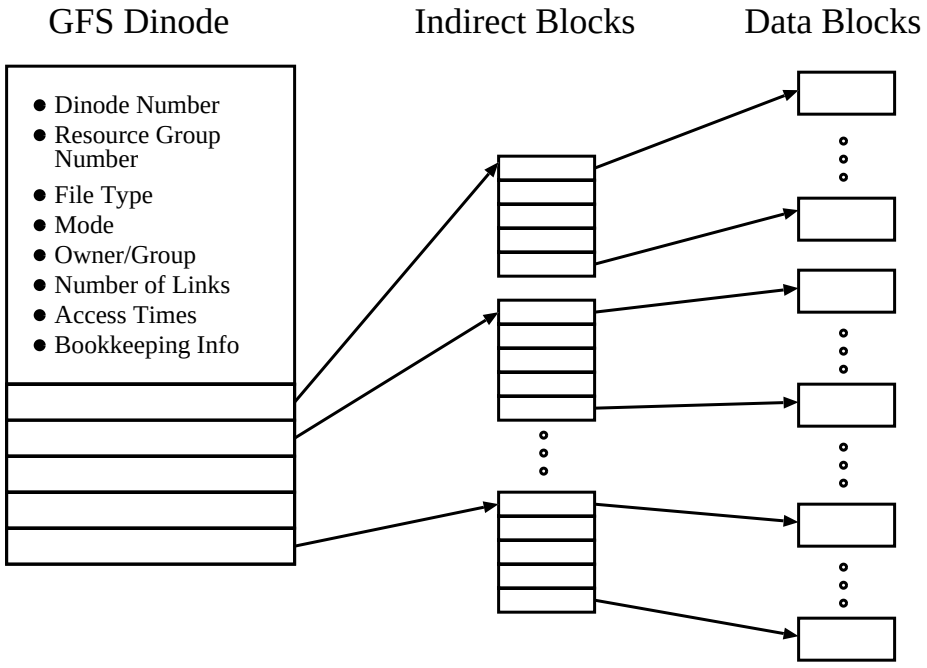

Figure 2: A GFS dinode. All pointers in the dinode have the same height in the metadata tree.

machines. The Linux VFS layer does provide some calls to the file system specific code to prevent the use of stale data, but it is awkward and not nearly as efficient as the SVR4 VFS. There are many places where the Linux VFS checks on file metadata (permissions, etc.) before it calls the file system specific code. In most cases, that data has to be rechecked in the file system specific code to avoid race conditions. SVR4 VFS doesn't make the check in the first place, so it is only done once (in the file system specific layer).

Linux is open source software and the development group is open to new ideas. One of the goals of the GFS group is to affect change in the Linux VFS layer that makes it more efficient for all file systems in general and GFS in particular.

3.1.2 Direct I/O

IRIX has something called Direct I/O. It allows a file system to copy data directly to and from the disk and a user buffer. This eliminates a memory copy, as normal buffered I/O reads data off the disk into the buffer cache and then copies it into the user buffer. Direct I/O can offer significant speed improvements in large file access.

At this time, Linux doesn't have an equivalent to Direct I/O. All disk I/O must pass through the buffer cache. Stephen Tweedie has written a patch for Linux that implements Direct I/O [16], but it won't go into the official kernel until version 2.3.

3.2 Linux Fibre Channel support

Currently, one Fibre Channel (FC) [17] driver is available for Linux. It was written at the University of New Hamphshire (UNH) InterOperability Lab for the Qlogic ISP2100 [18]. The driver is integrated into the Linux driver hierarchy so all attached FC devices appear in the file system as standard SCSI (sd) devices. The preinstalled firmware on the QLA2100 adapter currently supports only FC loops. Updated firmware with fabric support may be obtained. To date, we have not yet tested with the fabric-supporting firmware. New work on the driver will also be required to support fabrics. (A good description of Fibre Channel technology, including both loops and fabrics, can be found in the book by Benner [17].)

4 File System Improvements

Many improvements have been made to the file system and metadata structures described in [4] and [10]. These changes will, we believe, dramatically increase GFS's scalability.

4.1 Directories and Extendible Hashing

One of the places where traditional file systems don't perform well is large directories. Most early file systems (and a surprising number of modern ones) store directories as an unsorted linear list of directory entries. This is satisfactory for small directories, but it becomes too slow for big ones. On average, the file system must search through half of the directory to find any one entry. Not only is this costly in CPU time, but it causes excessive I/O to the disk. Large directories can take up megabytes of disk space.

The Global File System uses Extendible Hashing [19], [20] 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 amounts of searching aren't required.

For example, assuming a 4096-byte block size and 280-byte directory entries, a GFS ExHash directory can hold up to about 1700 files and find any file in one block read. A ExHash directory that contains up to about 910,000 files can find any file in two block reads. This compares to searching through (on average) half of the 62,000 blocks necessary to hold the directory if an unsorted linear list is used. Section 4.1.3 goes over how these numbers we arrived determined.

4.1.1 How does ExHash work?

The basis of ExHash is a multi-bit hash of each filename. A subset of the bits in the hash is used as an index into a hash table. The pointers of the hash table point to "leaf blocks" that contain the directory entries themselves. A particular entry in an ExHash directory is found using the following steps: Compute the 32-bit hash of filename, take the left X bits from the hash, look up the leaf block disk address in the hash table, read the leaf block, and search the leaf block for the directory entry.

The trick to this hashing scheme is that the hash table can grow in size as entries are added. The hash table is always a power-of-two in size. This power-of-two determines how many bits of the hash are used as an index. When the hash table becomes too small to hold the number of directory entries it needs it doubles in size, and one more bit of the hash is used. This allows a very large number of directory entries to be added without having to resort to linked lists of leaf blocks.

Instead of allocating one leaf block per hash table entry, leaf blocks can be pointed to by multiple hash table entries. This allows hash table pointers to share leaf blocks with other hash table pointers that have small numbers of directory entries. This makes ExHash memory efficient.

Leaf blocks always have a power-of-two number of hash table pointers pointing to them. When a directory entry is added to a full leaf block, the leaf block is split into two separate leaf blocks. Half of the hash table pointers point to the original leaf block and half point to the new leaf block. Directory entries are distributed between the two leaf blocks so that they are pointed to by the correct element of the hash table. When a leaf block needs to be split but it is only pointed to by one hash table pointer, the size of the hash table is doubled. There are then two pointers to the leaf block and the leaf can be split normally.

An example ExHash directory can be see in Figures 3–5. It shows a small hash table with a size of four. The first two bits of each file's hash are used in the table lookup. In this example, each leaf block holds three directory entries.

The GFS hash tables are stored as regular file data in the directory and accessed using the standard vnode read/write routines. This allows the hash table to grow to arbitrary sizes and still preserve optimal access time to any particular pointer. The leaf blocks are stored outside of the regular file metadata in blocks that are allocated by the directory routines.

GFS has two modes of directory operation. When the number of directory entries is small enough to fit in the part of the dinode normally reserved for metadata pointers, they are stuffed in that area, just as small regular files are stuffed in the dinode. The dinode is already in memory if the directory is open, so zero reads are required to find any entry.

4.1.2 Growing the Hash Table

When the directory can no longer be stuffed it is converted to an ExHash regime. The hash table starts out being half the size of a file system block and is stuffed into the dinode. All that is necessary to find any entry is to look up the leaf block address in the stuffed hash table and read in the leaf block. Because the dinode (and hash table) are already in memory, any directory entry can be found in one block read.

When the hash table doubles, it can't be stuffed anymore and is stored in the file data associated with the directory. From then on any entry can be found in 1 read of a hash table block, plus the number of indirect blocks between the dinode and the hash table, plus 1 read for the leaf block.

It is probably a good idea to keep the hash table from growing too big. Since the hash is 32 bits, the hash table could potentially take up 2^{32} bytes multiplied by the size of a leaf block pointer (8 bytes). It is possible (but

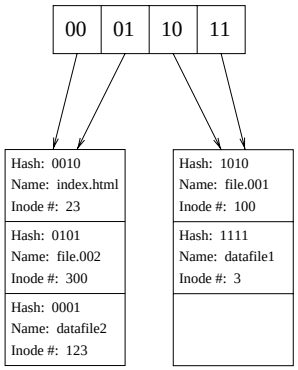

Figure 3: An ExHash directory: The hash table has a size of four and there are two leaf blocks.

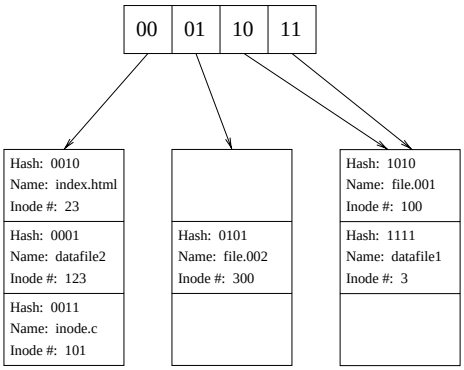

Figure 4: The directory from Figure 3 after the file “inode.c” was added. The addition forced the leftmost leaf block to be split.

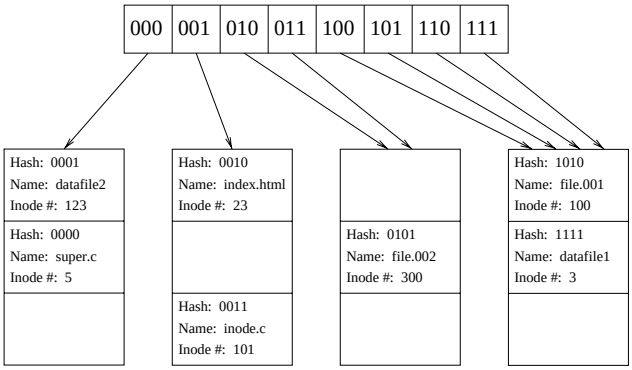

Figure 5: The directory from Figure 4 after the file “super.c” was added. The addition forced the hash table to be doubled and the leftmost leaf block to be split.

extremely unlikely) that a set of files with the same hash could be added to the directory. This would cause the hash table to grow to it's maximum size very quickly. Since 32 gigabytes is much too large for a directory (at least for now), GFS has a compile-time constant that prevents the hash table from growing beyond a certain size (16 megabytes by default). If the directory needs to grow beyond this size, leaf blocks are chained together in linked lists. This increases the average access time for directory lookups, but very slowly, requiring about one more disk access every time the size of the directory doubles. (At this point the directory holds about 29 million entries, so doubling it isn't very likely).

Adding and deleting directory entries takes the same amount of time as a directory search. Reading the entire contents of a directory is just as slow as in a linear directory because every block must be read. Every other operation is much faster than it would be compared to a directory stored in a linear fashion.

4.1.3 Access Times

Assuming a 4096-byte block size and 280-byte directory entries, a GFS ExHash directory can hold up to about 1700 files and find any file in one block read. In this case, the hash table is stuffed. A stuffed hash table has half as many pointers as can fit in a FS block. Pointers are eight bytes, so 512 pointers can fit in a FS block. Then the hash table has 256 entries. Each leaf block has a 32 byte header, so the number of directory entries that can fit in a leaf block is $\frac{4096-32}{280} = 14.5$. Assuming that an ExHash directory is about half full before the hash table doubles (a more than fair assumption), it could hold $\frac{1}{2} \times 14 \times 256 = 1792$ entries. The dinode (and consequently the hash table) is always in memory if the directory is open, so all that needs to be done is read the appropriate leaf block, i.e. one read.

Using similar logic and the knowledge that a GFS directory with a 4096-byte block size can grow to 131072 hash table entries before any indirect blocks are needed to store the table, it's possible to figure out how man entries can be access in two block reads: $\frac{1}{2} \times 14 \times 131072 = 917504$

4.1.4 Comparison with B-Trees

A comparison with B-trees [21] is warranted. B-trees are a common method of organizing directories in which blocks of directory entries are laid out in a tree pattern. Each block has pointers to $2d + 1$ other blocks, where d is the order of the tree.

The tree is arranged in such a way that a search for an entry involves starting at the root and following a branch to its leaf. The entry will be found at one of the blocks along the path. The search time (in terms of the number of block reads) is between one and $O(log_d(n))$. It is more likely to be closer to the upper bound because there are more entries close to the leaves.

As a comparison, assume a GFS directory with a block size of 4096 bytes and a directory entry size of 280 bytes. Directory entry lookup times for a ExHash directory with up to 469 million entries is three block reads (this includes one indirect metadata block, one hash block and one leaf block). A comparable B-Tree directory would have a height of at least six. The ExHash directory would be at least twice as fast, on average. Access times for ExHash consistently beat B-trees for all ranges of directory sizes.

The number of I/Os necessary to insert an entry into the directory is also another measure of efficiency. The common case for both ExHash and B-trees is an addition to a block with no overflow (i.e., there is enough room in the block to hold the entry). For an ExHash directory that takes the same amount of time as a search. B-trees always add entries to their leaves, so that means when adding a node the whole height of the tree must be traversed.

The more complicated case occurs when the block that should receive the entry has no room for it. A ExHash directory splits the leaf block in two. This involves accesses to the two leaf blocks and changing some of the pointers in the hash table. A B-tree overflow involves shifting entries to neighbor and parent blocks and possibly allocating a new block. The operations for both schemes are roughly comparable.

The most costly case for ExHash is when leaf block can't be split and the hash table needs to be doubled. The hash table must be read into memory and written back at twice it's previous size. (Then the leaf block is split.) As the hash becomes bigger, this becomes more and more expensive but fortunately, less and less frequent. The pathological case for B-trees is the successive overflow from child block to parent block all the way back to the root block, causing the tree to grow a block in height. There are also less pathological cases where the restructuring doesn't go all the way back to the root, but only part way up the tree. In these worst-case scenarios, ExHash is probably a little bit worse than B-Trees, but in most cases it will be faster.

Space efficiency is another concern. What percentage of the space used by the directory is actually used to hold directory entries? B-trees are always guaranteed to be at least half full. There are variants that are guaranteed to

be two-thirds full. They also have very little organization overhead. Every block used in the directory holds directory entries. The only organizational data stored are the $n + 1$ data pointers stored in each block for every n entries.

There is no limit on what percentage of a directory's allocated space is actually used by directory entries. The space efficiency of the directory is dependent on the quality of the hash function (see section 4.1.5). ExHash directories also have blocks that don't hold directory entries (i.e., the hash table). This lowers the space efficiency of the directory somewhat, but the hash table generally takes up less than one percent of the size of the leaf blocks.

We feel that, on average, Extendible Hashing provides a better method of directory organization that B-trees. It is quicker than B-trees but a little bit more space-hungry. Given the steep increase in available disk space and the much slower decrease in access times, we feel that this is the right trade-off.

4.1.5 The Hash Function

The hash function used in this algorithm is important. It must provide a uniform distribution of the hashed keys. If the distribution isn't uniform enough, the directory will be less space efficient. Lower space efficiency translates into slower access times due to bigger hash tables, more indirect blocks between the inode and the hash table data, and therefore more block reads per lookup.

One important thing to consider is the set of keys used in the hash. Common hash functions are designed to work well on words from a dictionary. Keys from this key space don't have too much in common with each other. The keys that GFS's hash function has to deal with are much more structured. One good example is a numerical time-domain electromagnetics simulation program that dumps out thousands of files named after the time step that generated them ("timestep.00001", "timestep.00002", "timestep.00003", and so on). Dates are another common element in the filenames of files that exist in large directories. Most common "fold and hash" functions [20], [22] fall apart for key spaces that overlap so much.

GFS uses a 32-bit cyclic redundancy check (CRC) for its hash function. Because CRCs are designed to detect single-bit errors, they provide a much more uniform distribution of hashed keys. The small number of bits that change between two names that differ by one digit produce a large change in the CRC. Our results show that CRCs perform significantly better than other hashes for "filename type" keys and they even do better for "dictionary type" key spaces. As far as the authors know, the approach of using a CRC as the hash function for for an extendible hash directory hasn't been used before.

The GFS directory code was instrumented so that it can provide measurements of the space efficiency of directories as they are filled. The measure is defined as:

$$\text{eff} = \frac{\text{Number Of Entries}}{\text{Number Of Blocks}(\text{Entries Per Block})} \quad (1)$$

The efficiency number indicates the fraction of the space allocated for the directory that actually contains directory entries.

As a test, a directory was filled with 45,402 files named after dictionary words. This was done twice. Once when GFS was using a conventional hash (from [20]) and once using a CRC hash. The results can be seen in figure 6. The *EntriesPerBlock* value from Equation 1 is about 14.6.

The efficiency results for the conventional hash were good until about the 27,000th entry. At this point, the conventional hash produced too many almost identical hashed keys. The new entry caused a leaf to overflow. Because the hashed keys were almost identical, the entries weren't split evenly between the two new leaves. In fact, they all were put into one leaf. This overflowed the leaf again and caused a second split. This overflowing continued until there was only one pointer from the hash table to the leaf. At this point, the overflowing caused the hash table to be doubled. Then the leaf could be split, but the split resulted in another overflow, which caused the hash table to double again, and so on.

In short, the hash table ballooned up to its maximum size of 16 MB. All the space required by the hash table reduced the space efficiency of the directory. In contrast, the CRC hash provided a more uniform distribution of hash keys. This increased the efficiency of the CRC hash directory and made it more consistent. It also prevents the cascading overflow situation. Directories with millions of entries have been created using the CRC hash and repeated overflow has never been a problem.

When the hash table swells to its maximum size, the access time for directory lookups is also increased. When the hash table is large, finding an entry in the hash table requires indirect block accesses. (Recall that the hash table is stored as regular file data.) Also, the overflow of a leaf with one pointer is handled by making a linked list from that leaf. So maximally sized hash tables not only reduces a directories space efficiency, it also reduces its speed.

The same thing was done with 45,402 "filename type" names. (We used: file.0000000000, file.0000000001, file.0000000002, and so on.) The results can be seen in

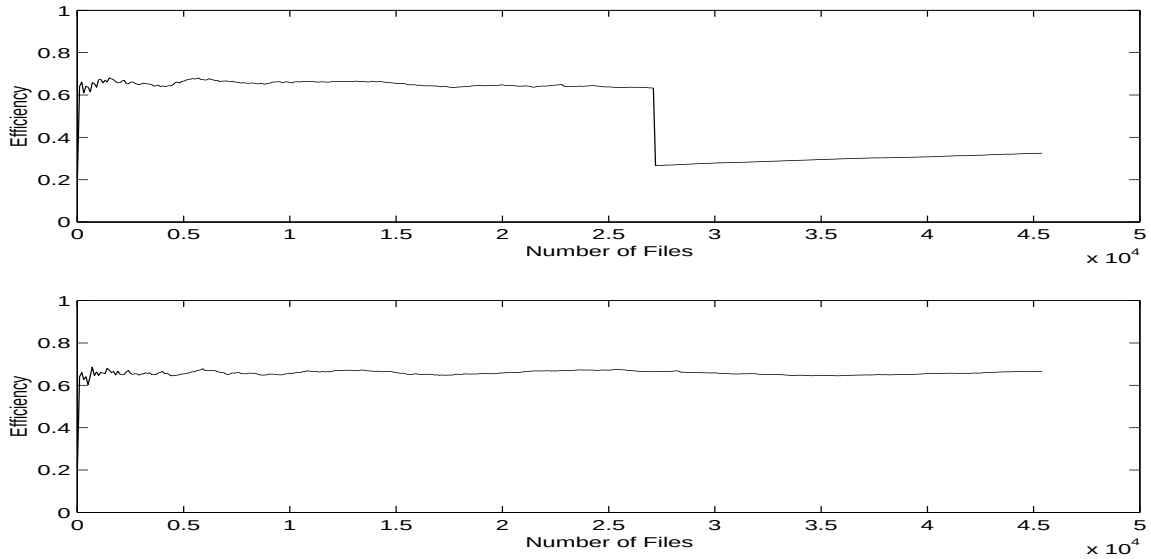

Figure 6: The space efficiency of a GFS ExHash directory as 45,402 files with names from the dictionary are created. The top graph shows the performance of a conventional hash. The bottom graph shows the performance of a CRC hash. The efficiency measure indicates the fraction of the space allocated for the directory that actually contains directory entries. (i.e. – An efficiency of one is optimal)

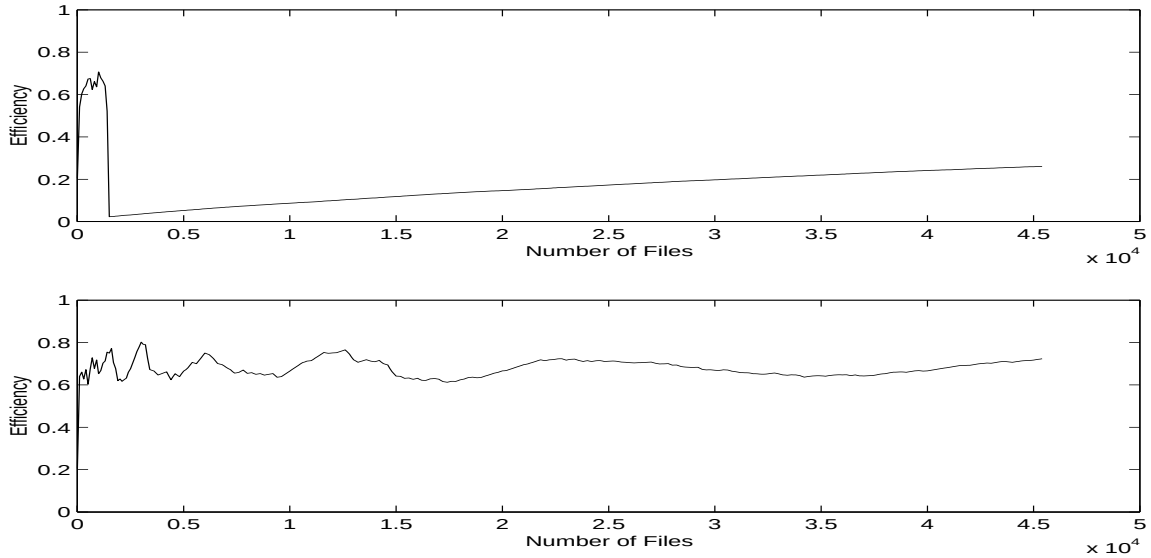

Figure 7: The space efficiency of a GFS ExHash directory as 45,402 files with names "file.0000000000" through "file.0000045401" were created. The top graph shows the performance of a conventional hash. The bottom graph shows the performance of a CRC hash.

Figure 7. The conventional hash caused the hash table to double to maximum size within the first 1500 entries. The efficiency of the CRC hash oscillates more than it did for the dictionary words, but the hash table remains a reasonable size.

In summary, the CRC hash function produces much better results than the conventional hash function. The CRC hash function is also fast. It is implemented in a few basic operations (AND, Shift, NOR, etc) and a lookup in a 1 KB lookup table. The time require to compute the CRC hash is comparable to conventional hashes.

4.2 GFS Consistency

Great care must be taken when metadata is accessed and updated. If the proper Dlocks aren't held at the right time, metadata and data corruption can easily result. Much of the recent GFS work has focused on making sure that locks are held in all the right places.

This new locking has also increased the potential for deadlock. There are many places where the file system must hold two or more Dlocks to perform an operation. For example, the lookup operation requires two simultaneous locks. The lookup operation takes a directory and the name of a file in that directory and returns the inode for that file. Two locks must be acquired for this operation: one lock must be held while the directory is read and the file's inode number is determined. The other lock must be held while the inode is read. These two locks must be held at the same time or race conditions exist with other processes on the same machine doing lookups on the same file, and other processes and machines trying to unlink this file.

There are a few other places where two or more locks are held and deadlock can occur. Ordering the acquisition of the Dlocks is difficult because Dlocks are assigned arbitrarily to different parts of the directory structure. An order that would prevent deadlock for one part of the file system tree could cause deadlock in other parts.

GFS handles this problem by implementing a system of back-offs and retries. If a client is holding one Dlock and wants another, it tries to get the new lock for a certain amount of time. If it doesn't get the lock in this time, it assumes a deadlock condition exists. It releases the first lock, sleeps for a random amount of time, and then retries the whole operation. This avoids deadlock, but it isn't optimal. The new version of the Dlock protocol allows clients to talk to each other directly, so that a separate fairness and sharing protocol can be applied if necessary.

Another new feature is that processes can now recursively acquire Dlocks. This was implemented by adding a layer between the file system and the NSP volume driver that examines each Dlock command before it is issued. If the Dlock has already been acquired by a process with the same process ID, a counter is incremented and the command is passed back up to the file system as if it was issued to the lock device and succeeded. If the lock is held by another process, the requesting process is put to sleep until the first process releases the lock. If the Dlock isn't currently held, the command is passed down to the pool device and is issued to the actual Dlock device.

When an unlock command is issued, the counter is decremented. When it reaches zero, the unlock command is passed down to pool and issued to the device.

An interesting and useful side effect of this algorithm is that it prevents multiple simultaneous lock requests to a lock device from the same machine. If one process has a Dlock and another process wants the same Dlock, the second process sleeps on a semaphore waiting for the first process to finish. This minimizes the amount of traffic on the network.

This recursive Dlock layer will be very important in the next generation GFS. In this new version, GFS will hold Dlocks much longer that it does now. This allows write caching and minimizes the effects of Dlock latency. Recursive Dlocks allow these locks to be held longer with minimal changes to the code.

To enable caching, when a Dlock is first acquired, the "Number of times locked" counter is set to 2 (instead of the usual 1). From this point forward the code acquires and releases locks as it normally would. The difference is that the lock and release command are all internal to the file system and don't access the lock device. When the file system needs to release the lock on the lock device, it calls the unlock routine one more time. This decrements the "Number of times locked" counter to zero and the unlock command is issued to the lock device.

4.3 Using the Buffer Cache

The buffer cache is an important component of modern UNIX operating systems [23], [2]. To prevent excessive disk accesses the operating system saves recently used disk blocks in a section of memory called the "buffer cache". Future requests for data already in the buffer cache can be completed quickly since no disk access is required. If the requested data is not in the buffer cache, it is read from disk and then copied into the buffer cache as well as to the user program. This applies to both metadata and file blocks.

Unlike IRIX, Linux presently provides no "direct" disk access to bypass the buffer cache. Large disk reads on

IRIX can benefit significantly from no buffering because only one memory copy is performed in the OS rather than two. Ordinary usage on both platforms, however, is characterized by frequent small file reads. In this case performance is greatly enhanced by using the buffer cache instead of accessing the disk. Caching metadata blocks also improves performance for large file requests because of repeated indirect block references.

Using the buffer cache in GFS is complicated by the ability of multiple clients to access and cache the same disk blocks. When a client detects data has changed on disk (indicated by a new Dlock counter value), it needs to invalidate those blocks in its buffer cache so the new data will be re-read. Recent changes in GFS keep track of cached buffers associated with each Dlock so they can be invalidated when necessary. This allows use of the buffer cache for reads, providing data for repeated small file requests and speeding up large file accesses. Without this ability in the past, all buffers were immediately invalidated after a read. Caching of writes is more difficult and cannot be implemented in GFS until the latest Dlock specification is in use [12].

Linux uses any spare memory as buffer space and reclaims this space when memory demands are higher. This further complicates the issue as GFS must not attempt to invalidate a buffer which has been reclaimed by the OS. Inserting a function pointer in the buffer head allows a GFS routine to remove the link to the departing buffer head. These changes in the Linux buffer management routines were possible because the kernel source was available. IRIX source code would enable similar improvements to GFS IRIX in the future.

With support for buffering in place, block read-ahead can now be effective. In this approach, a block request results in the next several blocks also being read at the same time and stored in the buffer cache. Program locality makes subsequent block reads likely, and pre-fetching them can improve overall performance.

4.4 Free Space Management

The current implementation of free space management in GFS is based on the bitmap approach. For every file system block in a given resource group there is a single bit to represent whether the block is free or not. This method is space efficient but as the file system fills with data, a search through an increasing number of bits is required in order to find the necessary free space. This becomes more costly with respect to performance with every additional byte we need to check, and even more expensive when it is necessary to search through individual bits.

The new approach, using an extent-based scheme, can potentially cost more in terms of space but should provide better performance. Instead of keeping track of each file system block in a resource group, we restrict ourselves to the free blocks. For every group of free file system blocks in a resource group there will be an extent that keeps track of the starting block and the number of blocks in the group, as shown in Figure 8. When the file system is created, it has one extent in each resource group. When files are added only the starting address in the extent needs to be changed. As files are removed, if the space freed cannot be added to an existing extent, a new one must be added. If the file system becomes highly fragmented, the amount of space necessary to hold the extents may become large.

There are two distinct advantages to this method. First, there is no need to search through a mapping of blocks in order to find the blocks that are free. Since we already know the blocks we are tracking are free, our focus is to find a group of free blocks that is large enough to hold our entire file. The second advantage of this scheme is that we can give the block allocator a "goal" block, i.e., a block that we would like our new space to follow. This way we can attempt to group metadata and data together on disk. While this approach may require more time to search through the extents to find an extent that starts closely after the goal block, it has the potential to reduce disk latencies in the future.

4.5 The Network Storage Pool

The pool driver coalesces a heterogeneous collection of shared storage into a single logical volume called the Network Storage Pool. The pool driver is built atop the SCSI and Fibre Channel drivers and is similar to SGI's xlv and Linux's md. It allows striping across multiple devices and provides a pool of Dlocks for GFS, hiding the implementation details. Devices may be divided into subpools according to specific performance characteristics.

Recent changes to the pool driver adapted it to the modular Linux block driver interface. The device locking options in the pool driver have also evolved to support exponential back-off from failed lock requests, multiple attempts to acquire locks, and giving up on Dlocks. We are developing a Dlock server daemon that runs over any IP network. This capability will be helpful in GFS development and will help those without Dlock support in their disk drives. (Note that GFS can be run in single machine mode with Dlocks disabled, which means that GFS can function efficiently as just another Linux desktop file system using any kind of disk.) Other pool driver addi-

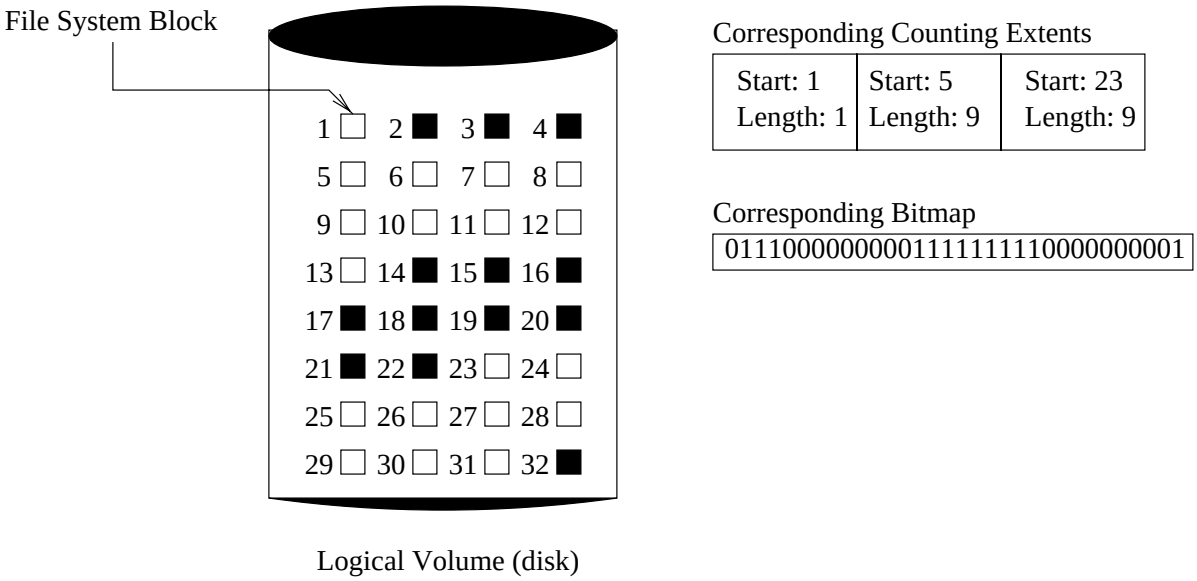

Figure 8: The extent and bitmaps encodings of 32 blocks, including free and in-use blocks.

tions include user-level tools to dynamically update pools in the kernel, and to dynamically create file systems based on pool parameters.

Ptool is a user-level tool which configures pool devices according to a parameter file edited by the user. The pool name, subpool definitions, subpool devices (individual disk partitions), striping sizes and scope, and Dlock devices can be specified in the parameter file. Labels containing all this information are written by *ptool* to the beginning of each disk partition used by the pool. *Ptool* needs to be run only once by one client for a pool to be created and accessible to all clients.

Passemble is the user level program which scans all the devices accessible to a client to determine what pools exist and can be used. All the labels on the shared devices (written by *ptool*) are read to construct the logical pool definitions. This information is then passed to the kernel which adds these definitions to its own list of managed pools. An important early improvement removed specific major and minor numbers from pool labels as various clients may identify devices differently. The requirement for a designated label partition has also been removed. New storage pools can be written, assembled and added to the kernel dynamically. *Passemble* needs to be run by each client at bootup and when a new pool has been created. Pool device files are also created and removed by *passemble* as storage pools are added and destroyed.

4.6 New Dlock Features

The new version of the Dlock protocol has features that allow GFS to perform better and more reliably. The new lock protocol is fully defined in [12]. The main additions are:

- **Dlocks Time Out**

  Each Dlock now has a timer associated with it. If the lock is left in the locked state for too long, the lock expires and is unlocked. A client that wishes to hold a lock for a long time can send "Touch Lock" commands that reset the timer on the lock.

  This new feature fixes one of the bigger performance problems in GFS. It allows the addition of write caching. Previously, clients had to discover failed clients by pinging the lock for some minimum time; the lock was reset manually if there was no activity. This meant there was a maximum amount of time that a lock could be held.

  In the new version, when Dlocks time out, the lock device determines which clients have failed. A client can hold a lock for a long period of time and be assured that no other client can read or write the data protected by the lock. This means that the client doesn't have to synchronously write back modified data so that extensive write-caching is now possible.

- **Client Identification Numbers are returned**

  Each GFS client in the new locking scheme is assigned a unique four-byte integer. The *Client ID* is passed to the Dlock device in the SCSI Command Descriptor Block of the Dlock command.

  Dlock commands that fail because the lock is held by another client return the Client IDs of the machines that are currently holding the lock. This allows out-of-band communication between clients while still keeping GFS Dlock-centric.

  This also helps clients hold locks for longer amounts of time. If a client wants a lock that is held by another client, it can use the returned Client ID to send a non-SCSI message to the client holding the lock. This message can either ask for the lock to be released or, perhaps, ask for authorization to do a third-party-transfer to or from the disk.

- **Reader/Writer Locks**

  Many pieces of data, especially metadata, are read often but written infrequently. This type of access pattern lends itself well to reader/writer locks. Readers acquire one of a multitude of reader locks. Writers acquire a writer lock that precludes both readers and other writers. Each Dlock in the new protocol is a reader/writer lock. This should help scalability in high traffic areas like the root directory.

5 Performance Results

Figures 9 through 11 represent the current single client I/O bandwidth of Linux GFS. The tests were performed on a 533 MHz Alpha with 512 MB of RAM running Linux 2.2.0-pre7. The machine was connected to eight Seagate ST19171FC Fibre Channel drives on a loop with a Qlogic QLA2100 host adapter card. GFS, at the time of this writing, has read caching implemented. Read Ahead and write caching have not yet been implement. 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 bandwidth of first time creates, shown in Figure 9, peaks at around 17.5 MB/s. For small request sizes, GFS performance leaves something to be desired at only about 1.3 MB/s. This will be greatly improved when write caching is implemented. Write caching will allow the small requests to be built up into bigger ones, and consequently improves performance. Right now, Dlocks are acquired and released for every request. The write caching implementation requires that Dlocks be held for long periods of time. Without the overhead of acquiring locks, small I/O request bandwidth should noticeably improve.

Figure 10 shows the bandwidth for preallocated writes on the same system. For preallocated writes, the metadata (and location of the data blocks) is quickly cached up and data is written out as fast as possible. Not having to allocate blocks increases the file system speed by about 25 percent. In the last IRIX release, preallocated writes were about twice as fast as creates [24]. The fact there there is such a small difference between creates and preallocated writes is a tribute to Linux GFS's new block allocation routines and read caching.

The read bandwidth shown in Figure 11 peaks at about 38 MB/s. This is faster than the buffered I/O of IRIX GFS (which peaked about 23 MB/s). The most noted improvement over IRIX GFS is in the small I/O request sizes. The bandwidth for small request sizes in IRIX GFS was about one to two megabytes per second. (The same was true for Linux GFS before read caching was implemented.) With read caching, small request I/O for Linux GFS is about 12 MB/s. We expect this numbers to improve even more when read ahead and the improved Dlock acquisition routines are implemented.

All these numbers show that Linux GFS buffered I/O is significantly better than IRIX GFS buffered I/O. However, IRIX GFS direct I/O offers a significant speedup. Peak bandwidth for large I/Os are 35 MB/s for creates and 55 MB/s for reads. (Direct I/O doesn't offer much of a speedup for small I/Os under the old IRIX GFS.) As stated in section 3.1.2, Linux does not yet have Direct I/O. Bandwidth for both reads and writes will increase when this hurdle is overcome.

6 Future Work

GFS has come a long way in the last three years, but it has a long way to go. Current and future work is described in the following sections.

6.1 Error Recovery

Error recovery is particularly important in a shared-disk file system. All the clients are directly manipulating the metadata, so the failure of any client could leave metadata in a inconsistent state. Furthermore, since there are so many machines accessing the disks, it is impractical for all of them to unmount and wait for a file system check (*fsck*) to complete every time a client dies. It is important that

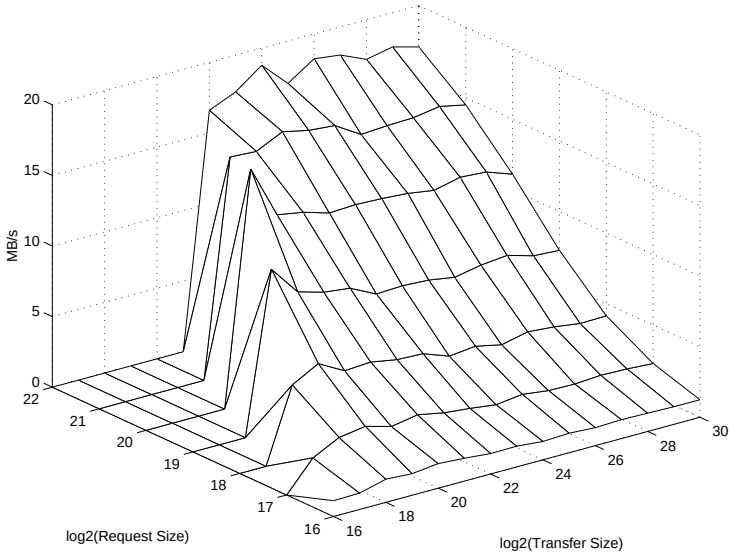

Figure 9: Linux GFS create bandwidth

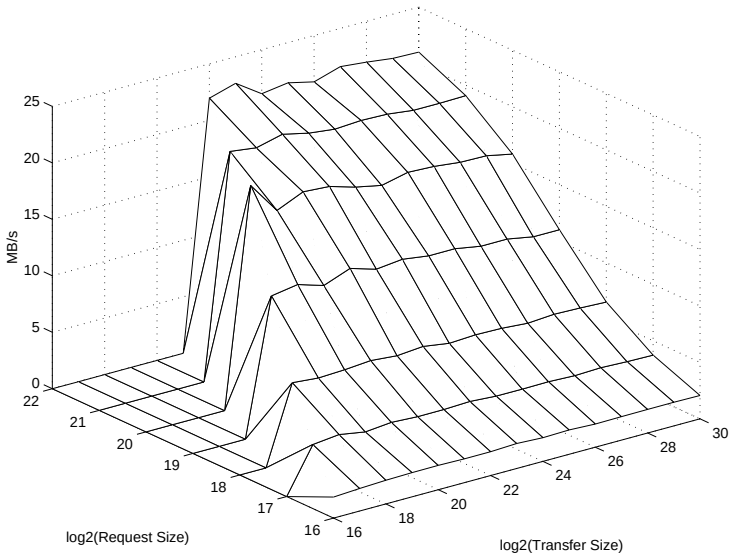

Figure 10: Linux GFS preallocated write bandwidth

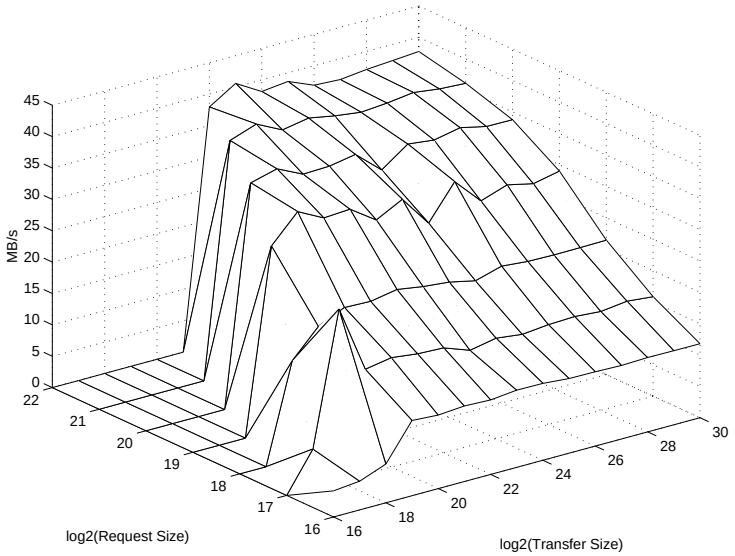

Figure 11: Linux GFS read bandwidth

the inconsistencies caused by a failed client are localized and easily repairable while the file system is online.

We are currently investigating a number of methods of recovery that will allow quick consistency checking. Snapshots, journaling, and logging are among them. The final version of the paper will describe a scalable metadata logging scheme for GFS.

6.2 OS pool sharing

The port of GFS to Linux has raised the obvious issue of sharing a file system among both IRIX and Linux clients. (as well as future platforms for GFS). There are no file system or pool driver barriers to doing this. Currently, to support both Alpha and x86 Linux clients, byte ordering and structure packing routines have been added to the file system and pool driver. The lack of a common disk partition format is the last problem to be solved before a GFS file system may be shared among differing operating systems.

For dedicated storage arrays, the easiest solution is just to have no partition format at all. When dealing with a storage pool made up of hundreds of disks, there is no point to dividing the disks into pieces. Grouping the disks in different ways provides the configure-ability that system architects require.

There are some situations, particularly involving SCSI over IP, where a cross-platform partition format would be useful. The Microsoft partition format is very common and would be a good choice if it wasn't so limited. Clearly, some other standard needs to be agreed on.

If GFS succeeds on Linux, it could provide a reference implementation for other programmers writing shared device file systems. If other operating systems use the GFS protocols and metadata for their shared device file system implementations, then interoperability can be achieved and incompatible shared file system formats avoided.

6.3 Growing File Systems

As devices are added to the storage network, the file system should be able to dynamically grow and use the new space. Enlarging the pool on which the file system resides is the first step. This is accomplished by making the new space an additional subpool. (Striping is confined to a subpool.) Passing the new data to the kernel and adding the subpool to the in-core structures is a simple process. The complexity arises in expanding *ptool* and *passemble* to dynamically change the pool defining labels and correctly assemble the pool definitions from new, in-use and unused devices belonging to multiple pools. At the file system level, a program needs to update the superblock and resource indexes on disk and prompt the file system on each client to reread this data so the new space will be used. We are working on making the changes to allow the file system to grow like this.

6.4 A GFS BSD Port

GFS is targeted at heterogeneous clusters of workstations. This commitment will continue with GFS ports to BSD UNIX.

6.5 SCSI over IP

By writing code that lets SCSI commands and data flow over IP networks, the notion of a storage area network is greatly expanded [25]. A shared device file system like GFS can access data that is spread over a much wider spectrum of hardware. Instead of network attached storage being limited to dedicated (and expensive) disks arrays, a computer can export its local disks to the IP network and essentially become a network attached storage device.

Since any machine can become a network attached storage device, upgrades to more conventional SAN hardware doesn't need to be as quick. A GFS installation can be created with commodity Ethernet hardware. As the demand for I/O bandwidth increases, Fibre Channel hardware can be added to create a Storage Area InterNetwork (See Figure 12). GFS accesses data with equal ease from the Ethernet or Fibre Channel networks, but the new hardware will be faster.

The key to SCSI over IP is two pieces of software, the client and the server. A server daemon waits for IP connections. When a connection is made, the daemon receives SCSI commands that are transmitted to it over the network. It then repackages those commands and sends them out across its local SCSI bus to a local disk. (It could also send them to a Fibre Channel disk it might be attached to.) It takes the response from that disk, packages it up, and sends it back out over IP.

The client presents an interface to the operating system that looks like a standard SCSI disk. When it gets a request from a higher level, it packages the command up and sends it across the network to the appropriate server machine. It then passes the response that comes back from the server up to the higher level.

The technology to package up parallel SCSI commands and send them over a serial line or network is already part of SCSI-3 [26]. All that is required is implementing the drivers. This should be straight forward. Van Meter has implemented just such a scheme and shown that it can achieve parallel SCSI speeds over fast Ethernet [25].

The server side can also be used to emulate SCSI commands. The server would look to see what type of SCSI command was being transfered. If it was is special command, the server daemon could handle it by itself and send a reply back to the client without ever talking to the disk. Other commands could be passed through to the disk.

The Dlock command could be implemented this way. The command is currently in the process of being standardized, but until it becomes wide spread in SCSI devices, the server daemon could emulate it.

7 Acknowledgments

Many people have contributed code and ideas to GFS over the years. The authors would like to thank the following people.

- From the **University of Minnesota**

  Benjamin I. Gribstad, Steven Hawkinson, Thomas M. Ruwart, Aaron Sawdey,

- From **Seagate Technology, Inc.**

  Dave Anderson, Jim Coomes, Gerry Houlder, Nate Larson, Michael Miller,

- From **NASA Ames Research Center**

  Alan Poston, John Lekashman

- From **Ciprico, Inc.**

  Edward A. Soltis

The authors would also like to acknowledge the valuable comments from the reviewer of this paper, Sam Coleman.

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] 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.

[4] 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*

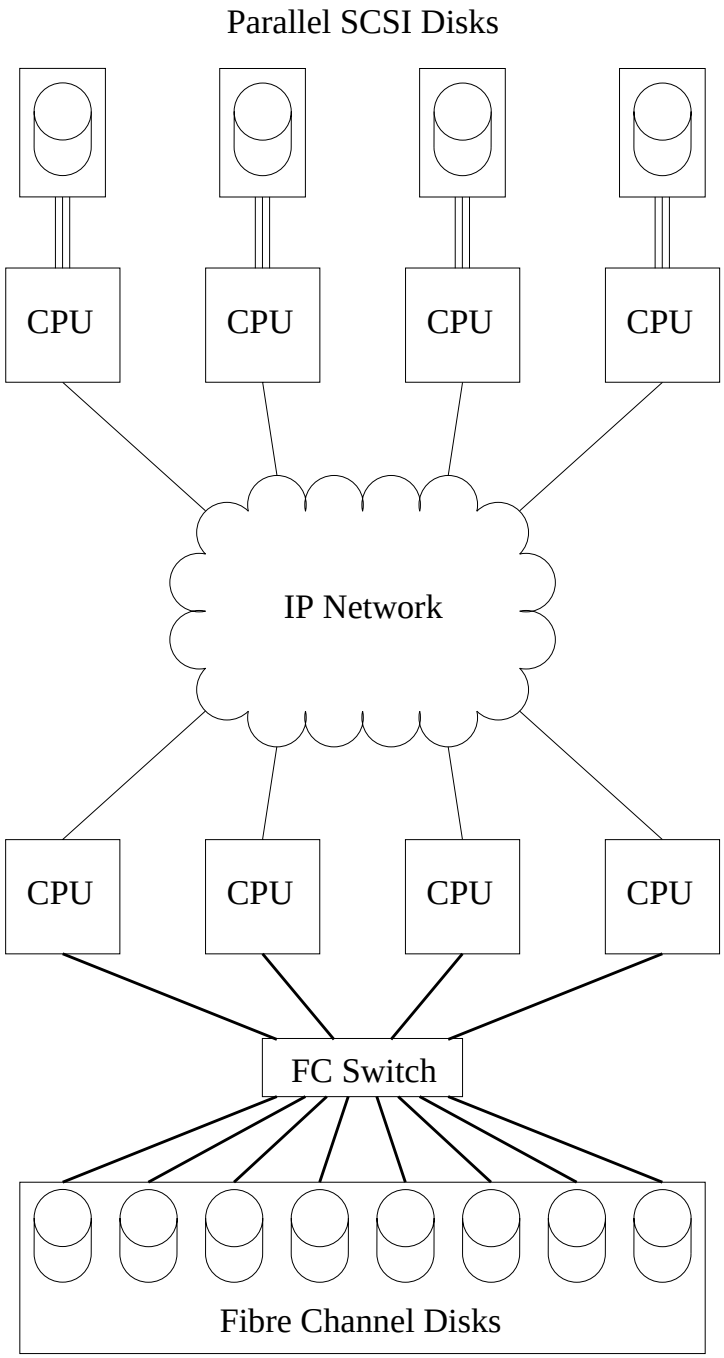

Figure 12: A Storage Area InterNetwork – Because all the CPUs are exporting their disks with SCSI over IP, all machines can access all disks.

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.

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

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

[7] 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.

[8] G. Pfister. *In Search Of Clusters*. Prentice-Hall, Upper Saddle River, NJ, 1995.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, second edition, 1998.

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

[15] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 238–247, June 1986.

[16] Stephen Tweedie. PATCH: Raw device IO for 2.1.131. http:// www.linuxhq.com/ lnxlists/ linux-kernel/ lk_9812_02/ msg00686.html, December 1998.

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

[18] Chris Loveland and The InterOperability Lab of the University of New Hampshire. Linux driver for the Qlogic ISP2100. http:// www.iol.unh.edu/ consortiums/ fc/ fc_linux.html, August 1998.

[19] 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.

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

[21] Douglas Comer. The ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.

[22] Aaron Sawdey. *The Sawdey Hash*. Email Message, April 1998.

[23] Maurice Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[24] Grant M. Erickson. The design and implementation of the global file system in silicon graphics' irix. Master's thesis, University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, March 1998.

[25] R. V. Meter, G. F. Finn, and S. Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[26] X3T10 SCSI committee. Draft proposed X3 technical report – Small Computer System Interface - 3 Generic Packetized Protocol (SCSI-GPP). http:// ftp.symbios.com/ ftp/ pub/ standards/ io/ x3t10/ drafts/ gpp/ gpp-r09.pdf, January 1995.

[27] 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://gfs.lcse.umn.edu`.