The Design and Performance of a Shared Disk File System for IRIX

Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Tom Ruwart
Department of Electrical and Computer Engineering
and
Laboratory for Computational Science and Engineering
University of Minnesota
Minneapolis, MN 55455
okeefe@ece.umn.edu
+1 612 625-6306

Abstract: In this paper we present a new storage architecture for clusters that creates a shared memory of disk storage that is uniformly accessible to all cluster clients, scales to large capacity, and provides very high performance and connectivity. The cluster structure resembles a symmetric multiprocessor (SMP) in that clients (processors) can access disk data (memory) across a local area network like Fibre Channel (a bus or other interconnection network). All clients can see and access the same disk data with perfect consistency. Our approach avoids buffer copy overheads and server bottlenecks found in traditional file systems while scaling to potentially large numbers of clients and large capacity disk systems.

We provide a description of the basic file system design, our current implementation on SGI IRIX operating system, and detailed benchmarking and performance analysis results. Good speedup and throughput is achieved for large files across 3 clients and 4 high performance disk arrays and on other client-array configurations[i] .

1. Introduction

Computer architects have for many years struggled with the problem of fast and efficient transfer of data between main memory and external storage devices, primarily disks. Until recently, it had been convenient (and certainly accurate) to point out that disk drives were three or four orders of magnitude slower than main memory and that their rate of capacity increase (25% per year), access time decrease (1/3 over 10 years), and bandwidth increase (20% per year) were below the corresponding rates for both IC logic for processors and DRAM for main memory trends. However, significant innovation in nearly all aspects of disk technology have accelerated these disk technology improvement curves so that since about 1992:

- capacity increases for disks at 60%/year are now roughly equal to those found in main memory
- bandwidth off the media is now increasing on average at about 40%/year
- access times are greatly reduced for some accesses that exploit on-board disk cache.

As was pointed out in the original RAID (Redundant Arrays of Inexpensive Disks) paper [6], a commodity market in SCSI (Small Computer Systems Interface) disk drives encouraged single-chip integration of SCSI controllers. Coupled with an on-board microprocessor and recent advances in high speed serial interfaces, disks now can

communicate with clients over intelligent, fast, and highly functional interfaces such as Fibre Channel. This interface technology combines both network and storage features and provides an industry-standard, high-bandwidth, switched interconnection network between clients and drives. These dramatic developments have encouraged us to propose a revolutionary rather than evolutionary approach to designing future storage architectures, one that assumes disks are highly-capable peer devices available directly on a network.

Traditional client/server distributed computing is limited in that it simply provides a mechanism for a client machine to transparently access data on a remote server through the client's local file system. Though useful, this approach limits the potential storage efficiency and speed that can be realized by distributed systems. The server is a potential bottleneck and single point of failure[ii]. To avoid both problems requires expensive redundancy (e.g., a Tandem system) or specialized hardware (for example, hardware built by Auspex or Maximum Strategies) tuned for the network file system protocols such as NFS. An alternative and increasing popular approach is *clustering*, which physically integrates stand-alone computers using fast networks, shared disk storage, and a single system image to create scalable compute and data servers [11], [7], [1].

An important component in several cluster designs is a *shared file system* that allows cluster machines to directly access shared disk devices across a network [7], [9], [4], [3], [11], [8] instead of through a server, increasing cluster performance and availability. In addition, we will show that such a shared file system also makes each node in a *storage area network* (SAN) more effective. A SAN consists of a local area network that allows storage devices to be directly attached to the network.

Disk drive and LAN speeds have increased gradually over the last ten years. Hard disk drives have had transfer rates from 1 to 3 Megabytes/second directly from media, though recently these rates have increased to 5 and in the next year will likely be over 20 Megabytes/second. During that period, Ethernet bandwidth has been limited by its shared physical media to less than 1 MB/sec. Hence, though direct disk attachment to networks is now possible [5], the network bandwidth has generally been too low to make it possible to exploit all the aggregate disk bandwidth actually available on the network. This imbalance between disk drive and Ethernet speed has become even more pronounced with RAID devices and multi-level RAID hierarchies which have aggregate bandwidths of several hundred Megabytes/second or more [12].

Fortunately, recent advances in switching technology, fiber optics and the convergence of network and channel interfaces [13], [10] are allowing order-of-magnitude improvements in network latency and bandwidth through new technologies like Fibre Channel and Gigabit Ethernet. Open standards and high-volume markets, combined with the constant increase in functionality and decrease in cost for microelectronic devices, will drive down network costs. The previous speed imbalance between disk drives and networks will be reversed: parallel drive designs will be needed to exploit switched network bandwidth and meet the requirements of tomorrow's demanding applications.

For example, the new Fibre Channel standard integrates both storage and networking capabilities into a single interface that currently has a speed of 100 Megabytes/second (and a growth path to 400 Megabytes/second), allows both low-cost loop connections (much like FDDI rings) with up to 126 devices at distances beyond 100s of meters and is scalable to 100s or 1000s of devices with Fibre Channel switches. Yet Fibre Channel will achieve very widespread use with disk drives and adapters priced about the same as parallel SCSI technology [2]. In contrast, today's parallel SCSI technology supports only about 8 devices per bus with each bus extending at most 25 meters making the technology effectively unscalable.

These new network technologies will certainly improve the performance of today's client-server networks. However, the advances in network-attached storage interfaces, network bandwidth and scalability, disk bandwidths, capacities and access times along with demanding new applications requiring high bandwidth and high availability challenge the basic client-server architecture upon which distributed systems have been constructed. Distributed systems in the future will increasingly rely on clustering for high availability and will require richly interconnected storage networks to support data-intensive computing.

Given these fast, low-cost, switched networks, a serious review of the division of responsibilities between clients, servers, and storage devices has lead us to an alternative storage architecture, based upon our Global File System (GFS) [17] design that is *serverless* and consists only of *clients* and *networked storage devices.* This proposal motivates the design of the GFS, outlines its basic structure, describes the current software and related performance results and how we intend to extend and test the GFS within the context of an innovative hardware infrastructure here at the University of Minnesota.

A key GFS goal is to remove the master-slave structure found in current distributed computing client-server environments. Instead, given the low latency and high bandwidth of new network technologies, we can design a *symmetric multi-client* system where multiple clients access storage devices across a fast switched network such as Fibre Channel. This allows a cluster to behave much like a symmetric multiprocessor: processors (clients) are equal in the eyes of the kernel (there is no master) and each has equal access to the main memory (disk drives or other storage devices) via a fast bus, multistage network, or crossbar switch (e.g., a fast switched network such as Fibre Channel). This structure has many advantages:

· The GFS provides a storage architecture that allows the storage system designer and administrator *to pool disk drives into a shared disk memory* equally accessible to all clients in the system.

· There is *no single point-of-failure* for a storage device since it is not attached to a single client, thus allowing for fail-over redundancy [20]. Low-level RAID striping provides redundancy at in the disk drives, much like error-correcting DRAM main memories.

· The GFS architecture can *exploit the bandwidth capabilities* both within and across next-generation PCs, desktop workstations, high-end servers and supercomputers.

· *Local client bandwidth need not be wasted* in making transfers from a local storage device to another client as in client-server architectures.

· The *GFS architecture is inherently more reliable than other distributed file systems* since it is easier to build redundancy into a disk array than it is to insure that a complicated server (including hardware, software, and network connections) does not fail.

· The size of the file system and consequently the *size of a single file is not limited* by the size of the storage subsystem on any given client.

· The file system may *span multiple storage devices*.

· Each client connected to the peripheral network *views the devices as locally attached*.

These GFS advantages are relevant both in tightly-integrated cluster architectures but also in more loosely-coupled storage area networks, enabling more efficient data transfer and sharing.

2. Global File System—Architecture and Design

The Global File System is a distributed file system based on shared network-attached storage. Clients service only local file system requests and act as file managers for their own requests; storage devices serve data directly to clients. No direct communication is necessary between clients to enable basic GFS operation so that client failures or bottlenecks do not in general affect other clients.

How GFS Views Network Storage

As shown in Figure 1, in a GFS storage system the network-attached storage devices on the peripheral network form a global pool that we call the *Network Storage Pool* (NSP) that can be carved up into many *subpools*. This partitioning into subpools allows the system manager to configure separate subpools, each with different characteristics, including:

- number of disks (or disk arrays) in a subpool
- stripe unit size
- access attributes (such as client affinity for a particular subpool, contiguous blocks, etc.)
- performance attributes (such as meeting a strict bandwidth or latency limit).

GFS provides transparent parallel access to storage devices while maintaining standard UNIX file system semantics: user applications still see only a single logical device via the standard *open*, *close*, *read*, *write* and *fcntl*. This transparency is important for ease-of-use and portability. However, GFS will allow some user control of file placement on physical storage devices based on the appropriate attributes required such as bandwidth, capacity, or redundancy.

A Brief Description of File Systems

File systems maintain persistent user and system data on storage devices such as disk drives. They maintain files by keeping pointers to file data blocks which are fixed size (typically 1-4Kbytes) and an integer multiple of the storage device block size. Some file systems such as UFS [20] have semantics that allow applications to access data in units smaller than the file system block size which generally requires buffering in main memory.

A file is an operating system abstraction that hides from the user the details of how the data is mapped to physical storage devices. Typically, an application reads and writes data to and from a file as if the data were a linear sequence of randomly-accessible bytes (or blocks or possibly records) — but the data may, and often is, scattered throughout the blocks of the physical device.

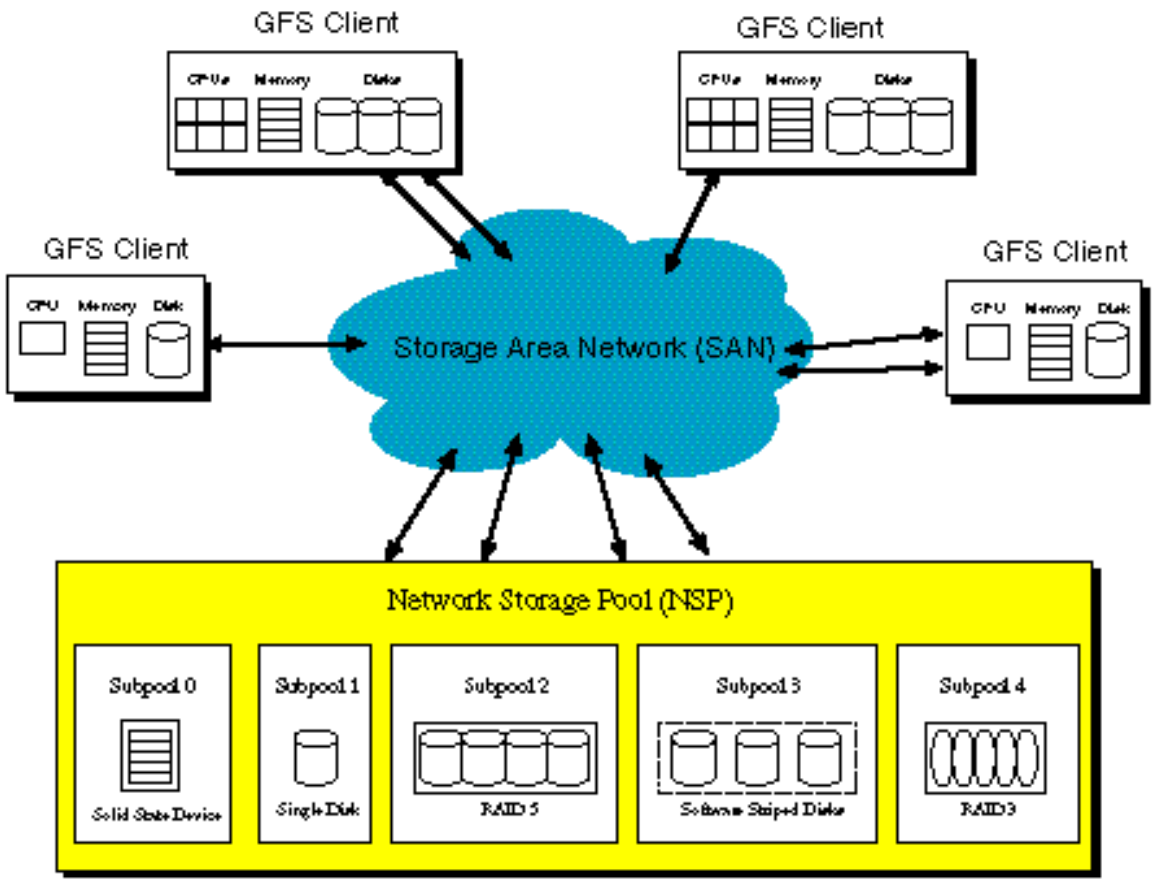

Figure 1: *Global File System Distributed Environment*

A directory is a type of file which contains groups of other files and directories. Directories are hierarchical, yielding a tree-structured name space containing all files and directories for a given file system. Associated with each file is a unique number or handle called an *inode number* in UNIX file systems. Each inode has a corresponding *dinode* located on the physical storage device which maintains information about the file owner, permissions, number of links, access times, size and *pointers* to the location of the file's data blocks on the physical storage devices. An *inode* is the in-memory data structure corresponding to the dinode.

The file system stores the dinodes, known as *metadata*, along with the actual file data. In addition to dinodes, the file system maintains *free lists* of data blocks not allocated to files. In modern UNIX file systems, free lists are implemented as *bitmap tables* where each bit represents a file system block; a bit that is set signifies that the corresponding block is already allocated. A file system maintains a single *superblock* which contains the layout of the file system, maintains counts of free dinodes and free data blocks, and stores mount information such as mount device and access privileges.

GFS Implementation: Metadata and Data

The GFS structure and internal algorithms differ from traditional file systems, emphasizing sharing and connectivity rather than caching. Unlike local file systems, GFS distributes file system resources across the entire storage subsystem, allowing simultaneous access from multiple machines. GFS also attempts to place specific data types, either metadata or data, on subpools with suitable performance characteristics.

The network storage pool (NSP) shown in Figure 1 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 inherent the physical attributes of the underlying devices and network connections.

GFS, unlike typical file systems, distributes its metadata throughout the network storage pool rather than concentrating it all into a single superblock. As shown in Figure 2, 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 can be thought of as partitioning the file system into distinct sets of files and available data and metadata blocks.

Resource groups are similar to the *allocation groups* (AGs) found in SGI's XFS file system [19]. Like resource groups, allocation 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 as shown in Figure 3. 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.

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 16 KB and assume the dinode header information requires 128 bytes. Without stuffing, a 1-byte file requires a total of 32 KB and at least 2 disk transfers to read the dinode and data block. With stuffing, a 1-byte file only requires 16 KB and one read request. The file can grow to 16 KB minus 128 bytes, or 16,266 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.

Using a flat pointer tree structure as shown in Figure 4, the maximum file size for GFS assuming 8K file system blocks and 8 byte pointer addresses is about a factor of 1000 greater than those attainable with UFS. (However, the UFS dinode pointer tree requires fewer indirections for small files.) Other alternatives include extent-based allocation such as SGI's EFS file system or the B-tree approach of SGI's XFS file system [19]. The current structure of the GFS metadata is an implementation choice and these alternatives are worth exploration in future research.

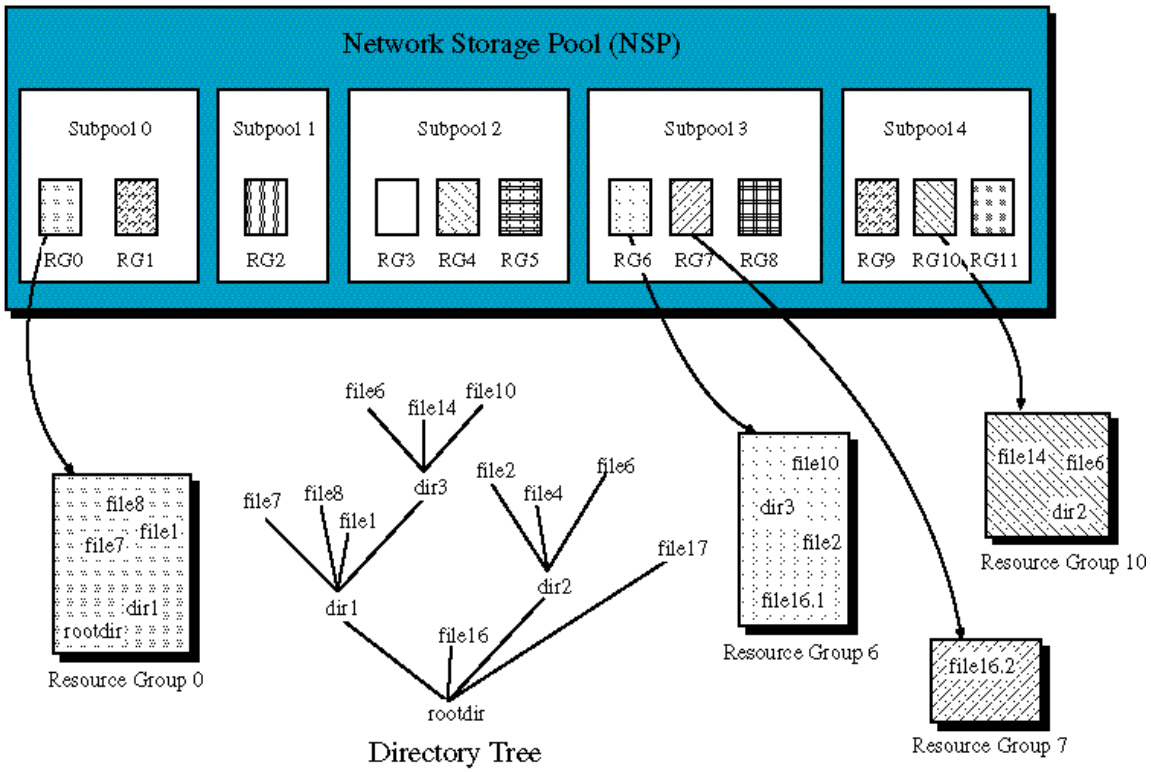

Figure 2: *Files Mapped onto Resource Groups and Subpools.*

GFS Implementation: Consistency and Caching

As shown previously, GFS exploits network-attached storage in a *Network Storage Pool* to create a shared disk memory that behaves much like the shared main memory found in multiprocessors today: each processor sees the same file system name space and has equal access to the shared disk memory. And like a multiprocessor system which must permit controlled, synchronized access to shared memory [14], a key design issue is the ability of each client to access the file system that may span many shared storage devices without destructively interfering with other clients accessing the same file system.

Multiple client accesses to shared devices must be synchronized: the three primary alternatives are disk-based [18], device-based [9] or client-based [7], [8], [3] synchronization. In the disk-based approach used by GFS, locks resident on the drive are used by multiple clients to safely manipulate file system metadata. A device-based approach is similar except that the locks are found on a shared device independent of the disk drives. Finally, client-based synchronization distributes the locking function between the clients: messages are exchanged to lock a particular file or resource. Some form of lock table (either centralized or distributed) is used to maintain the current state of shared, potentially locked resources.

GFS uses atomic read-modify-write operations on disk-resident metadata to maintain file system consistency. These operations guarantee that at a fixed point in time data exists in at most three places: the disk media, the disk on-board cache, and in client memory. The disk maintains consistency between its media and cache while the GFS file system uses disk-based locks to implement atomic operations on metadata to maintain consistency between multiple clients and disk devices. Atomic operations on shared data are performed by acquiring exclusive access to the data via a lock, reading the data from memory or storage,

modifying the data, writing the data back, and releasing the exclusive access by giving up the lock.

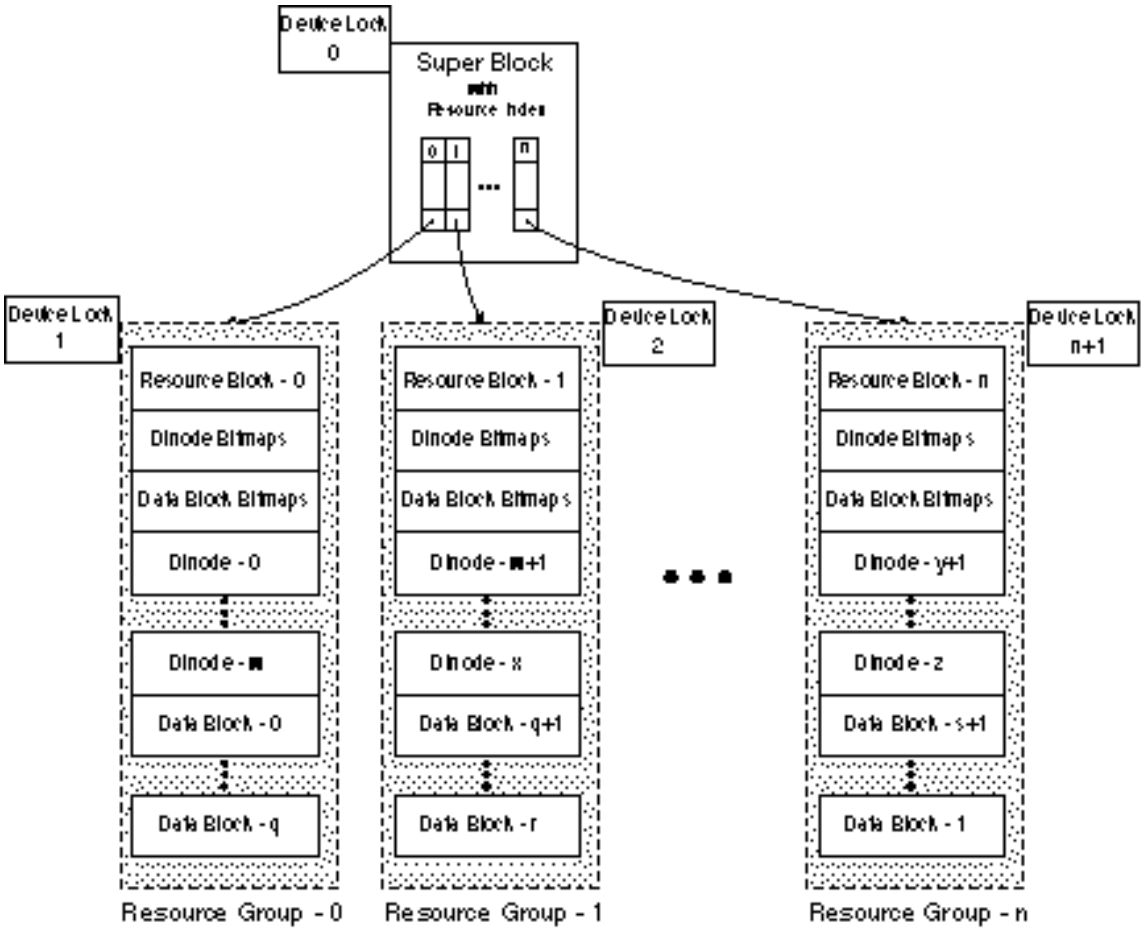

Figure 3: *GFS Metadata Structure*

Our initial GFS implementation [15] used the basic SCSI command for locking devices known as RESERVE/RELEASE [2]. However, this command works at the granularity of an entire disk which makes it impossible to have more than one lock active for a single device, severely restricting parallelism and hence scalability. The SCSI standard does allow reservations on "extents" (contiguous logical blocks) but since this command is not mandatory very few drives actually support it. In addition, in discussions with our industrial partners Seagate and Ciprico we found that this particular command would add significant overhead to other commands, making it less appealing to implement. In response, we worked jointly with Seagate and Ciprico to develop a new SCSI command called DLOCK [15] which provides a fast, efficient locking primitive — a *device lock* — ideal for our GFS implementation.

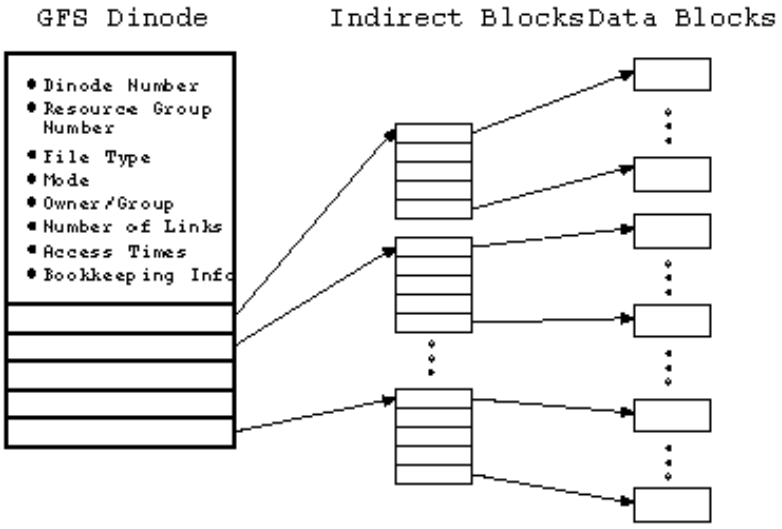

Figure 4: *GFS Dinode Internal Structure*

Device locks (*DLOCKs*) are implemented as an array of state bytes in volatile storage on each device. Each lock is referenced by number in the SCSI command: the state of each lock is described by one bit. If the bit is set to 1, the lock has been acquired and is owned by an initiator (client). If the bit is 0, the lock is available to be acquired by any initiator. The DLOCK command action *test and set* first determines if the lock value is one. If the value is 1, then the command returns with status indicating the lock has already been acquired. If the value is 0, DLOCK sets the lock to 1 and returns GOOD status to the initiator. The DLOCK command *clear* simply sets the lock bit to 0. A *test* operation is provided to read (but not set) the state of the lock.

It is important to realize that the device does not itself understand how the lock relates to the data on its own media or on other devices: that is the task of the file system or database. Hence, the granularity at which the lock is applied is up to the system software. In Figure 3, we can see that the current GFS implementation associated device locks with resource groups, dinodes, and the super block. DLOCKs are general enough to support mutual exclusion on just about any resource on the network. These locks are only held temporarily during metadata updates so it is important that the DLOCK commands be executed quickly.

The GFS metadata strategy is to cache metadata both on the disk drive caches and client memory. If exploited properly, the solid-state buffer memories on the disk drives provide a convenient cache structure that can be shared by multiple GFS clients. GFS clients can also directly cache some file system metadata that is read-only. This approach uses only a small amount of client memory for file caching and removes the burden other distributed file systems like AFS and DFS [20] have in supporting large, sophisticated file caches directly on clients. This plays directly to the new capabilities of disk drives: faster interfaces and fast access to local disk caches[iii].

Reliability and availability are important to the Global File System since it is designed to support large numbers of disks and clients operating in parallel. Given high disk drive failure rates some form of redundancy is necessary at the drive level. This can be done with the appropriate RAID level applied to groups of drives to provide high availability. Client failures can be quite common: in traditional client-server systems a client failure implies loss of access to data on drives attached to the client. This problem is avoided in the GFS because drives are not attached to a single client but are instead accessed across a peripheral

network. This network should in general be more reliable than either clients or devices and provides multiple, redundant paths between them[iv].

But important issues such as lock contention, redundancy, fairness, and error recovery must be considered in the design. Our proposed DLOCK SCSI command provides for additional lock state known as the *logical clock* that lets clients determine when another client with a lock has failed, leaving the lock in a zombie-like state [15]. We have proposed a distributed recovery mechanism using these clocks in the event of client failure. This information can also be used to help prevent a slow client from being starved for access to a particular lock and in measuring device workloads to provide information to a file system load-balancing utility. Finally, this state information allows GFS clients to determine whether metadata cached on the client has been modified back on the device and is therefore stale. Hence, DLOCK provides a simple yet elegant command that integrates consistency, caching, and recovery in the GFS architecture[v].

3. The Current Global File System Implementation and its Performance

We have a GFS implementation developed on Silicon Graphic's IRIX operating system using the VFS/VNODE interface [20]. Our implementation is based upon the architecture described in section 2 and includes the GFS VNODE and VFS operations, a network storage pool driver, test scripts, performance measurement GUIs, and file system utilities (like *mkfs*) that together comprise nearly 40,000 lines of code. It is basically complete and further testing and device integration are proceeding as GFS becomes available to users in our laboratory and elsewhere so that we may study its performance under real application workloads. We are also adding more industrial partners to our efforts to more widely disseminate this new technology.

Though much of our initial testing and development have exploited parallel SCSI disk drives we are most interested in GFS performance and scalability on a true network-attached storage interface like Fibre Channel. A detailed report on our initial performance results can be found in [16]. Here we summarize these results; later we discuss the implications of this work relative to other research in this area and our proposed research objectives.

The test configuration we used is show in Figure 5. We were interested in how GFS performance would scale as both clients and disk arrays were added to this four-client, four-array configuration. We tested for several parameters including file transfer time and bandwidth using a range of file sizes. Some tests measured the aggregate transfer rate across all clients working together while others measure the throughput of transferring files of varying sizes across the whole system. The tests included measurements from eight Seagate Barracuda 9 Fibre Channel disk drives and 4 Ciprico 7000 series RAID-3 Fibre Channel disk arrays connected via a 16-port Brocade Silkworm Fibre Channel switch to 4 SGI Challenge computers.

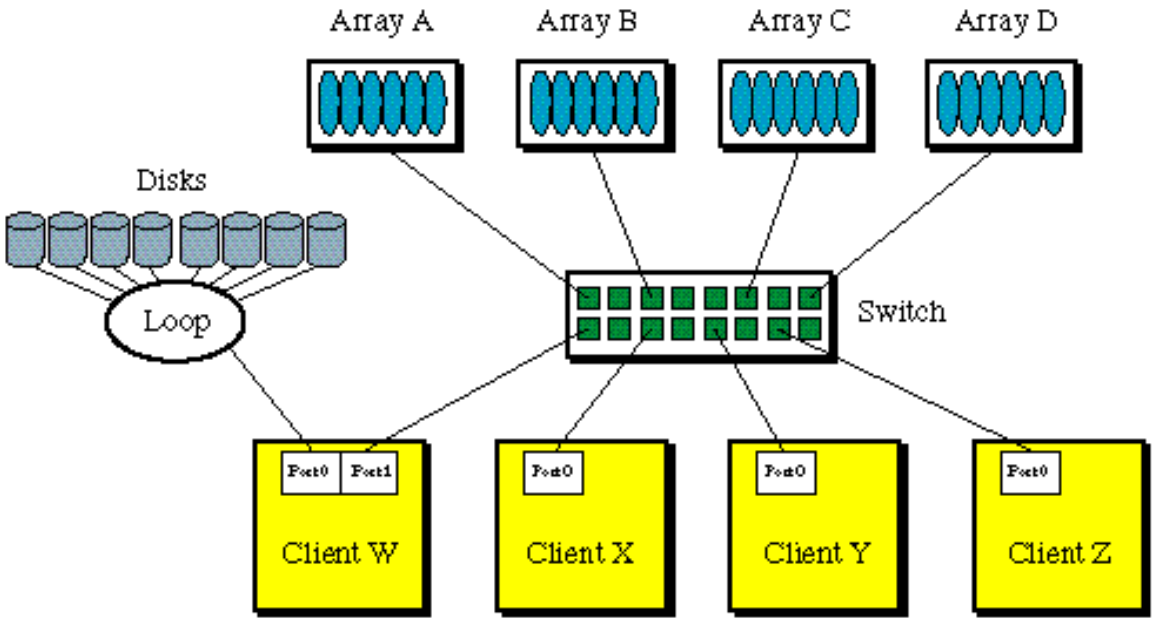

Figure 5: *GFS Hardware Configuration for Tests Performed May 1997.*

Separate measurements were performed on device lock performance which showed that on both the Seagate and Ciprico products locking required about 1.3 milliseconds (ms) while unlocking required 1.0 ms. GFS read performance relative to raw device performance is given in Table 1. It can be seen that GFS performance relative to the underlying hardware is quite good for larger files but lags for smaller files. Small file performance lags in GFS because the high metadata access overheads are not amortized by the long transfer times required by larger files. In addition, current disk device caches are not tuned well to cache metadata; these caches are focused mainly on read-ahead and write-behind of sequential data and are used to essentially cache several tracks worth of data. Hence, further joint work with our industrial partners will be necessary to develop and implement appropriate caching strategies on devices and integrate into GFS the necessary cache control features to exploit these device caches.

Sizes		Single Disk			8-Wide Disks			Disk Array		
File (MB)	Request (MB)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)
4	2	10.5[1]	8.22	78.3	60.9[1]	24.8	40.7	74.8	33.5	44.8
4	4	10.4[1]	8.38	80.6	58.5[1]	25.6	43.6	76.9[2]	33.8	44.0
16	8	10.5	9.81	93.1	59.1	43.2	73.1	78.2	61.1	78.1
16	16	10.1	9.90	96.2	58.5	45.2	77.6	79.0	61.0	77.2
128	8				61.8[2]	19.9	77.0	81.4[2]	68.9	84.1
128	16				61.9[2]	54.0	88.2	82.0[2]	73.1	89.1
256	8							81.4	70.1	86.1
256	16							82.0	74.8	91.2

1 Estimated from 16 MB sequential performance tests
2 Estimated from 256 MB sequential performance tests

Table 1. *Raw Disk Performance versus GFS Read Performance.*

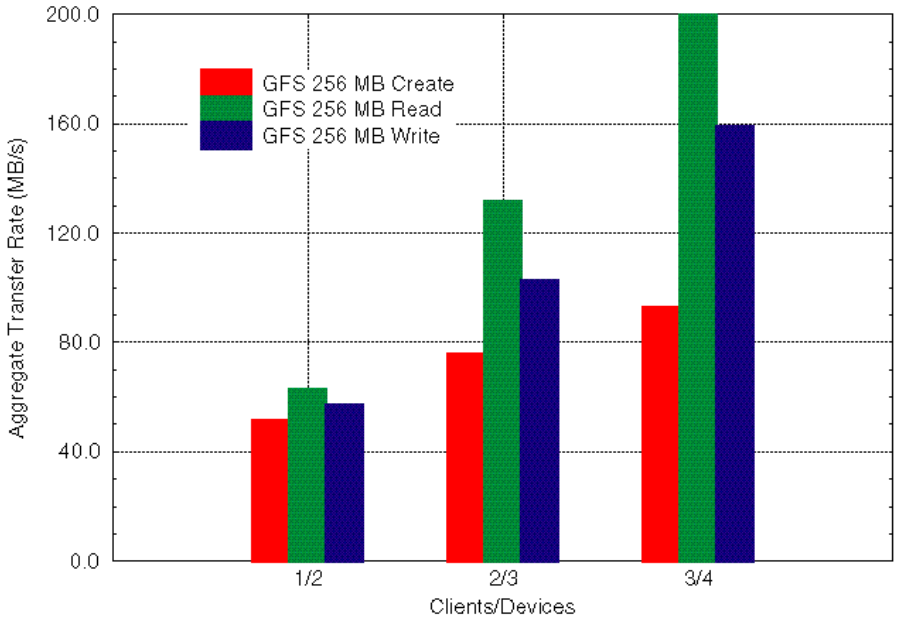

Figure 6: *Aggregate Transfer Rate of 256 Mbyte Files with a Root Directory Device.*

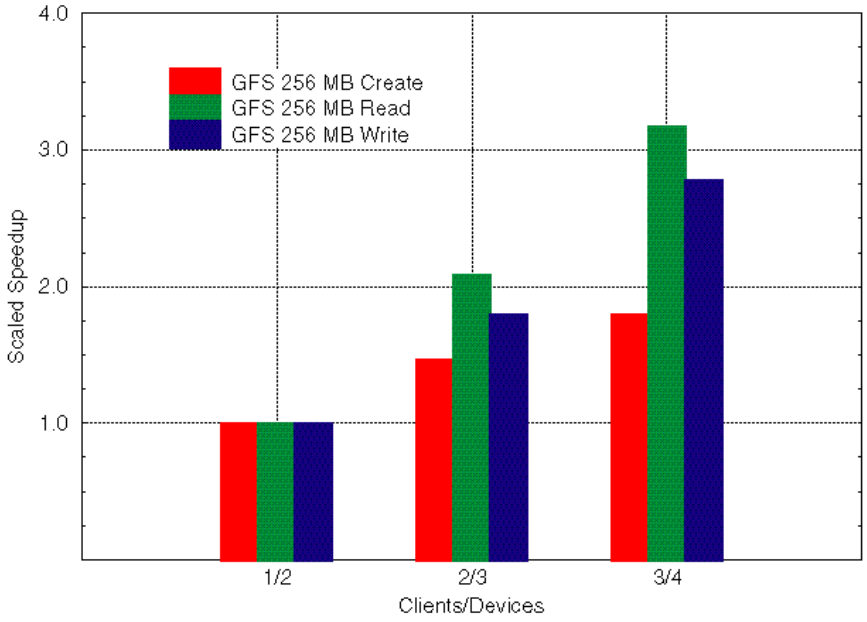

Figure 7: *Scaled Speedup of 256 Mbyte Files with a Root Directory Device.*

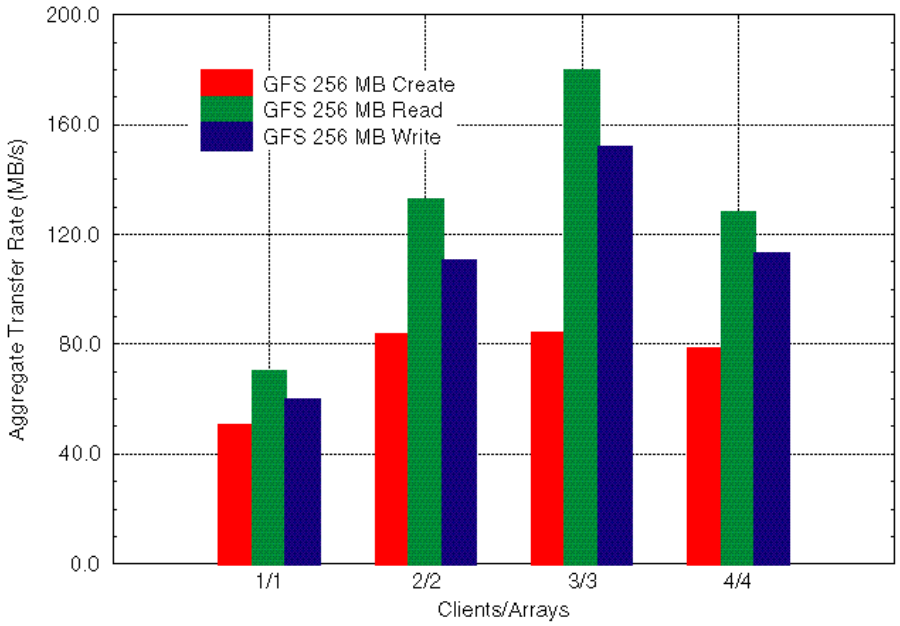

Figure 8: *Aggregate Transfer Rate of 256 Mbyte Files.*

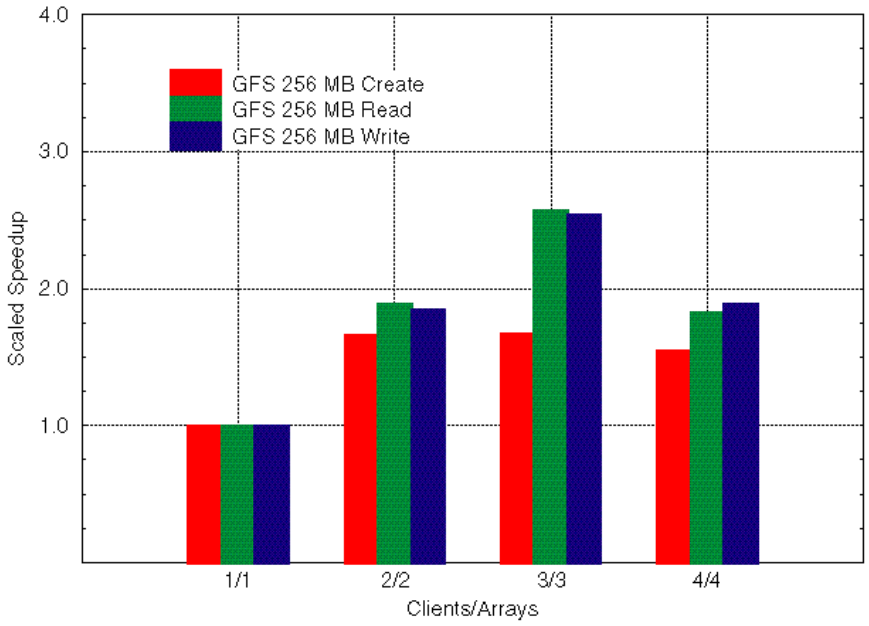

Figure 9: *Scaled Speedup for 256 Mbyte File.*

The scaling observed for aggregate transfer rates across the four machines and arrays are given in two sets of figures. In Figures 6 and 7 aggregate bandwidth and scaled speedup

across from 1 to 3 disk arrays and machines is given for the case where a dedicated device is used for holding device locks and the root directory metadata. Good scaling is seen for this case with large files sizes and minimal contention between device locks, directory accesses, and access to actual file data. This contrasts with Figures 8 and 9 where these same metrics were considered for the case where a dedicated root directory and lock device was not used so that contention between accesses for file data, directories and device locks occurred on the same array. It can be seen that performance in fact decreases with the addition of the fourth array and machine due to this contention.

Fortunately this poor scaling is more the result of implementation decisions rather than the fundamental GFS architecture. In fact in this test case the resource group layout was not optimized for this configuration resulting in more contention than otherwise would have been necessary.

4. Conclusions

We believe that our results show that a new approach to cluster file system design is appropriate in the context of network attached storage. Our performance results are unique compared to earlier cluster file system studies in that our results are derived from a file system designed from scratch to exploit the data and device sharing potential of network attached storage. Future work will include reducing contention for shared resources such as the root directory, performance evaluation of GFS on larger configurations with up to 16 clients and 8 to 12 disk arrays, and more aggressive use of device caching for shared file system metadata. In addition, we are implementing a scheme for distributed recovery for device and client failures which integrates metadata consistency checking with file lock removals for failed nodes.

References

[1] D. Seachrist, R. Kay, and A. Gallant, "Wolfpack Howls Its Arrival," BYTE Magazine, pp. 126-130, vol. 22, no. 8, August 1997.

[2] D. Deming. The SCSI Tutor . Saratoga, CA: ENDL Publishing, 1994.

[3] M. Devarakonda, A. Mohindra, J. Simoneaux, W. Tetzlaff, "Evaluation of Design Alternatives for a Cluster File System," 1995 USENIX Technical Conference, January 1995.

[4] G. Gibson et al., "File Serving Scaling with Network-Attached Secure Disks," Proceedings of the ACM Int. Conf. on Measurements and Modeling of Computer Systems (SIGMETRICs '97), Seattle, WA, June 15-18, 1997.

[5] S. Koegler, "SPANStor Adds on Network Storage with Ease and Convenience," Network Computing, November 1, 1995.

[6] R. Katz, G. Gibson, and D. Patterson, "Disk System Architectures for High Performance Computing," Proceedings of the IEEE, vol. 77, pp.1842-1858, 1989.

[7] N. Kronenberg, H. Levy, W. Strecker, "VAXClusters: A Closely-coupled Distributed System," ACM Transactions on Computer Systems, vol. 4, no. 3, pp. 130-146, May 1986.

[8] I. Lloyd, "The Oracle Parallel Server Architecture," Proceedings of Supercomputing-Europe 92, pp. 5-7, 1992.

[9] K. Matthews, "Implementing a Shared File System on a HiPPI Disk Array," Fourtheenth IEEE Symposium on Mass Storage Systems, pp. 77-88, September 1995.

[10] R. Meter, "A Brief Survey on Current Work on Network Attached Peripherals," ACM Operating Systems Review, pp. 63-70, January 1996.

[11] G. Pfister, In Search of Clusters . Upper Saddle River,NJ: Prentice-Hall, 1995

.

[12] T. Ruwart and M. O'Keefe, "A 500 Megabyte/Second Disk Array," Fourth Nasa/Goddard Conference on Mass Storage Systems and Technologies, College Park, Maryland, March 1995.
[13] M. Sachs, A. Leff, and D. Sevigny, "LAN and I/O Convergence: A Survey of the Issues," IEEE Computer, vol. 27, no. 12, pp. 24-33, December 1994.
[14] C. Schimmel, UNIX Systems for Modern Architectures. Addison-Wesley: Reading, MA, 1995.
[15] S. Soltis, The Design and Implementation of a Distributed File System Based on Shared Network Storage. Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, August 1997.
[16] S. Soltis, G. Erickson, K. Preslan, T. Ruwart, M. O'Keefe, The Global File System: A File System for Shared Disk Storage, submitted to the IEEE Transactions on Parallel and Distributed Systems, October 1997.
[17] S. Soltis, T. Ruwart, and M. O'Keefe, "The Global File System," Fifth NASA Goddard Conference on Mass Storage Systems and Technologies, College Park, MD, September 1996.
[18] S. Soltis, M. O'Keefe, T. Ruwart, and B. Gribstad, SCSI Device Locks, technical report, Department of Electrical Engineering, University of Minnesota, April 1996.
[19] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, "Scalability in the XFS File System," 1996 USENIX Technical Conference, January 1996.
[20] U. Vahalia, UNIX Internals: The New Frontiers. Prentice-Hall, Upper Saddle River, NJ, 1996.

---

[i] This work was supported by the Office of Naval Research under grant no. N00014-94-1-0846, by the National Science Foundation under grant no. CDA-9414015 and no. ASC-9523480, by NASA through grant no. NAG2-1151 and by equipment grants from Seagate Technology, Brocade Communications, Silicon Graphics Inc. and Ciprico. Contact the authors at `okeefe@lcse.umn.edu`.

[ii] Leslie Lamport, a well-known researcher, is quoted as saying a "distributed system is one where the failure of some computer I've never heard of can keep me from getting my work done."

[iii] Solid-state disk caches were not widely used until about 1990. Most operating systems and file systems were developed prior to this time, when no solid-state disk caches were available.

[iv] Much like earlier IBM mainframe channel architectures, future Fibre Channel drives have multiple ports which could be used to improve availability.

[v] DLOCK has been implemented by Seagate Technology and Ciprico in their disk products and we report our initial performance results in [SoE97]. We are in the process of filing a patent application for the DLOCK invention.