USENIX Association

Proceedings of the FAST 2002 Conference on File and Storage Technologies

Monterey, California, USA
January 28-30, 2002

GPFS: A Shared-Disk File System for Large Computing Clusters

Frank Schmuck and Roger Haskin
IBM Almaden Research Center
San Jose, CA

Abstract

GPFS is IBM's parallel, shared-disk file system for cluster computers, available on the RS/6000 SP parallel supercomputer and on Linux clusters. GPFS is used on many of the largest supercomputers in the world. GPFS was built on many of the ideas that were developed in the academic community over the last several years, particularly distributed locking and recovery technology. To date it has been a matter of conjecture how well these ideas scale. We have had the opportunity to test those limits in the context of a product that runs on the largest systems in existence. While in many cases existing ideas scaled well, new approaches were necessary in many key areas. This paper describes GPFS, and discusses how distributed locking and recovery techniques were extended to scale to large clusters.

1 Introduction

Since the beginning of computing, there have always been problems too big for the largest machines of the day. This situation persists even with today's powerful CPUs and shared-memory multiprocessors. Advances in communication technology have allowed numbers of machines to be aggregated into computing *clusters* of effectively unbounded processing power and storage capacity that can be used to solve much larger problems than could a single machine. Because clusters are composed of independent and effectively redundant computers, they have a potential for fault-tolerance. This makes them suitable for other classes of problems in which reliability is paramount. As a result, there has been great interest in clustering technology in the past several years.

One fundamental drawback of clusters is that programs must be partitioned to run on multiple machines, and it is difficult for these partitioned programs to cooperate or share resources. Perhaps the most important such resource is the file system. In the absence of a cluster file system, individual components of a partitioned program must share cluster storage in an ad-hoc manner. This typically complicates programming, limits performance, and compromises reliability.

GPFS is a parallel file system for cluster computers that provides, as closely as possible, the behavior of a general-purpose POSIX file system running on a single machine. GPFS evolved from the Tiger Shark multimedia file system [1]. GPFS scales to the largest clusters that have been built, and is used on six of the ten most powerful supercomputers in the world, including the largest, ASCI White at Lawrence Livermore National Laboratory. GPFS successfully satisfies the needs for throughput, storage capacity, and reliability of the largest and most demanding problems.

Traditional supercomputing applications, when run on a cluster, require parallel access from multiple nodes *within* a file shared across the cluster. Other applications, including scalable file and Web servers and large digital libraries, are characterized by *interfile* parallel access. In the latter class of applications, data in individual files is not necessarily accessed in parallel, but since the files reside in common directories and allocate space on the same disks, file system data structures (metadata) are still accessed in parallel. GPFS supports fully parallel access both to file data and metadata. In truly large systems, even administrative actions such as adding or removing disks from a file system or rebalancing files across disks, involve a great amount of work. GPFS performs its administrative functions in parallel as well.

GPFS achieves its extreme scalability through its *shared-disk* architecture (Figure 1) [2]. A GPFS system consists of the cluster nodes, on which the GPFS file system and the applications that use it run, connected to the disks or disk subsystems over a switching fabric. All nodes in the cluster have equal access to all disks. Files are striped across all disks in the file system – several thousand disks in the largest GPFS installations. In addition to balancing load on the disks, striping achieves the full throughput of which the disk subsystem is capable.

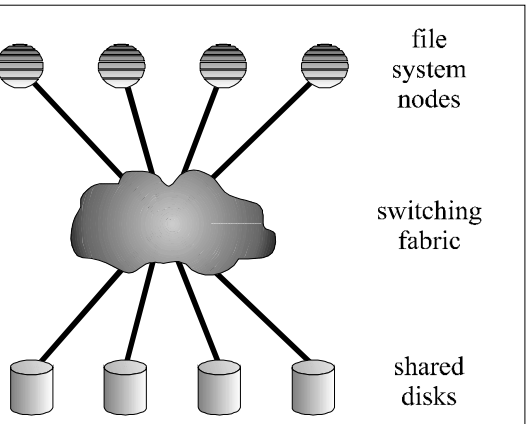


Figure 1: Shared Disk Environment

The switching fabric that connects file system nodes to disks may consist of a storage area network (SAN), e.g. fibre channel or iSCSI. Alternatively, individual disks may be attached to some number of I/O server nodes that allow access from file system nodes through a software layer running over a general-purpose communication network, such as IBM's Virtual Shared Disk (VSD) running over the SP switch. Regardless of how shared disks are implemented, GPFS only assumes a conventional block I/O interface with no particular intelligence at the disks.

Parallel read-write disk accesses from multiple nodes in the cluster must be properly synchronized, or both user data and file system metadata will become corrupted. GPFS uses distributed locking to synchronize access to shared disks. GPFS distributed locking protocols ensure file system consistency is maintained regardless of the number of nodes simultaneously reading from and writing to the file system, while at the same time allowing the parallelism necessary to achieve maximum throughput.

This paper describes the overall architecture of GPFS, details some of the features that contribute to its performance and scalability, describes its approach to achieving parallelism and data consistency in a cluster environment, describes its design for fault-tolerance, and presents data on its performance.

2 General Large File System Issues

The GPFS disk data structures support file systems with up to 4096 disks of up to 1TB in size each, for a total of 4 petabytes per file system. The largest single GPFS file system in production use to date is 75TB (ASCI White [3, 4, 5]). GPFS supports 64-bit file size interfaces, allowing a maximum file size of 2^{63}-1 bytes. While the desire to support large file systems is not unique to clusters, the data structures and algorithms that allow GPFS to do this are worth describing.

2.1 Data Striping and Allocation, Prefetch and Write-behind

Achieving high throughput to a single, large file requires striping the data across multiple disks and multiple disk controllers. Rather than relying on a separate logical volume manager (LVM) layer, GPFS implements striping in the file system. Managing its own striping affords GPFS the control it needs to achieve fault tolerance and to balance load across adapters, storage controllers, and disks. Although some LVMs provide similar function, they may not have adequate knowledge of topology to properly balance load. Furthermore, many LVMs expose logical volumes as logical unit numbers (LUNs), which impose size limits due to addressability limitations, e.g., 32-bit logical block addresses.

Large files in GPFS are divided into equal sized blocks, and consecutive blocks are placed on different disks in a round-robin fashion. To minimize seek overhead, the block size is large (typically 256k, but can be configured between 16k and 1M). Large blocks give the same advantage as extents in file systems such as Veritas [15]: they allow a large amount of data to be retrieved in a single I/O from each disk. GPFS stores small files (and the end of large files) in smaller units called sub-blocks, which are as small as 1/32 of the size of a full block.

To exploit disk parallelism when reading a large file from a single-threaded application GPFS prefetches data into its buffer pool, issuing I/O requests in parallel to as many disks as necessary to achieve the bandwidth of which the switching fabric is capable. Similarly, dirty data buffers that are no longer being accessed are written to disk in parallel. This approach allows reading or writing data from/to a single file at the aggregate data rate supported by the underlying disk subsystem and interconnection fabric. GPFS recognizes sequential, reverse sequential, as well as various forms of strided access patterns. For applications that do not fit one of these patterns, GPFS provides an interface that allows passing prefetch hints to the file system [13].

Striping works best when disks have equal size and performance. A non-uniform disk configuration requires a trade-off between throughput and space utilization: maximizing space utilization means placing more data on larger disks, but this reduces total throughput, because larger disks will then receive a proportionally larger fraction of I/O requests, leaving

the smaller disks under-utilized. GPFS allows the administrator to make this trade-off by specifying whether to balance data placement for throughput or space utilization.

2.2 Large Directory Support

To support efficient file name lookup in very large directories (millions of files), GPFS uses *extensible hashing* [6] to organize directory entries within a directory. For directories that occupy more than one disk block, the block containing the directory entry for a particular name can be found by applying a hash function to the name and using the *n* low-order bits of the hash value as the block number, where *n* depends on the size of the directory.

As a directory grows, extensible hashing adds new directory blocks one at a time. When a create operation finds no more room in the directory block designated by the hash value of the new name, it splits the block in two. The logical block number of the new directory block is derived from the old block number by adding a '1' in the $n+1^{st}$ bit position, and directory entries with a '1' in the $n+1^{st}$ bit of their hash value are moved to the new block. Other directory blocks remain unchanged. A large directory is therefore, in general, represented as a sparse file, with holes in the file representing directory blocks that have not yet been split. By checking for sparse regions in the directory file, GPFS can determine how often a directory block has been split, and thus how many bits of the hash value to use to locate the directory block containing a given name. Hence a lookup always requires only a single directory block access, regardless of the size and structure of the directory file [7].

2.3 Logging and Recovery

In a large file system it is not feasible to run a file system check (fsck) to verify/restore file system consistency each time the file system is mounted or every time that one of the nodes in a cluster goes down. Instead, GPFS records all metadata updates that affect file system consistency in a journal or write-ahead log [8]. User data are not logged.

Each node has a separate log for each file system it mounts, stored in that file system. Because this log can be read by all other nodes, any node can perform recovery on behalf of a failed node – it is not necessary to wait for the failed node to come back to life. After a failure, file system consistency is restored quickly by simply re-applying all updates recorded in the failed node's log.

For example, creating a new file requires updating a directory block as well as the inode of the new file. After acquiring locks on the directory block and the inode, both are updated in the buffer cache, and log records are spooled that describe both updates. Before the modified inode or directory block are allowed to be written back to disk, the corresponding log records must be forced to disk. Thus, for example, if the node fails after writing the directory block but before the inode is written to disk, the node's log is guaranteed to contain the log record that is necessary to redo the missing inode update.

Once the updates described by a log record have been written back to disk, the log record is no longer needed and can be discarded. Thus, logs can be fixed size, because space in the log can be freed up at any time by flushing dirty metadata back to disk in the background.

3 Managing Parallelism and Consistency in a Cluster

3.1 Distributed Locking vs. Centralized Management

A cluster file system allows scaling I/O throughput beyond what a single node can achieve. To exploit this capability requires reading and writing in parallel from all nodes in the cluster. On the other hand, preserving file system consistency and POSIX semantics requires synchronizing access to data and metadata from multiple nodes, which potentially limits parallelism. GPFS guarantees single-node equivalent POSIX semantics for file system operations across the cluster. For example if two processes on different nodes access the same file, a read on one node will see either all or none of the data written by a concurrent write operation on the other node (read/write atomicity). The only exceptions are access time updates, which are not immediately visible on all nodes.[1]

There are two approaches to achieving the necessary synchronization:

1. *Distributed Locking*: every file system operation acquires an appropriate read or write lock to synchronize with conflicting operations on other nodes before reading or updating any file system data or metadata.

[1] Since read-read sharing is very common, synchronizing atime across multiple nodes would be prohibitively expensive. Since there are few, if any, applications that require accurate atime, we chose to propagate atime updates only periodically.

2. *Centralized Management*: all conflicting operations are forwarded to a designated node, which performs the requested read or update.

The GPFS architecture is fundamentally based on distributed locking. Distributed locking allows greater parallelism than centralized management as long as different nodes operate on different pieces of data/metadata. On the other hand, data or metadata that is frequently accessed and updated from different nodes may be better managed by a more centralized approach: when lock conflicts are frequent, the overhead for distributed locking may exceed the cost of forwarding requests to a central node. Lock granularity also impacts performance: a smaller granularity means more overhead due to more frequent lock requests, whereas a larger granularity may cause more frequent lock conflicts.

To efficiently support a wide range of applications no single approach is sufficient. Access characteristics vary with workload and are different for different types of data, such as user data vs. file metadata (e.g., modified time) vs. file system metadata (e.g., allocation maps). Consequently, GPFS employs a variety of techniques to manage different kinds of data: byte-range locking for updates to user data, dynamically elected "metanodes" for centralized management of file metadata, distributed locking with centralized hints for disk space allocation, and a central coordinator for managing configuration changes.

In the following sections we first describe the GPFS distributed lock manager and then discuss how each of the techniques listed above improve scalability by optimizing – or in some cases avoiding – the use of distributed locking.

3.2 The GPFS Distributed Lock Manager

The GPFS distributed lock manager, like many others [2, 9], uses a centralized global lock manager running on one of the nodes in the cluster, in conjunction with local lock managers in each file system node. The global lock manager coordinates locks between local lock managers by handing out *lock tokens* [9], which convey the right to grant distributed locks without the need for a separate message exchange each time a lock is acquired or released. Repeated accesses to the same disk object from the same node only require a single message to obtain the right to acquire a lock on the object (the lock token). Once a node has obtained the token from the global lock manager (also referred as the *token manager* or *token server*), subsequent operations issued on the same node can acquire a lock on the same object without requiring additional messages. Only when an operation on another node requires a conflicting lock on the same object are additional messages necessary to revoke the lock token from the first node so it can be granted to the other node.

Lock tokens also play a role in maintaining cache consistency between nodes. A token allows a node to cache data it has read from disk, because the data cannot be modified elsewhere without revoking the token first.

3.3 Parallel Data Access

Certain classes of supercomputer applications require writing to the same file from multiple nodes. GPFS uses byte-range locking to synchronize reads and writes to file data. This approach allows parallel applications to write concurrently to different parts of the same file, while maintaining POSIX read/write atomicity semantics. However, were byte-range locks implemented in a naive manner, acquiring a token for a byte range for the duration of the read/write call and releasing it afterwards, locking overhead would be unacceptable. Therefore, GPFS uses a more sophisticated byte-range locking protocol that radically reduces lock traffic for many common access patterns.

Byte-range tokens are negotiated as follows. The first node to write to a file will acquire a byte-range token for the whole file (zero to infinity). As long as no other nodes access the same file, all read and write operations are processed locally without further interactions between nodes. When a second node begins writing to the same file it will need to revoke at least part of the byte-range token held by the first node. When the first node receives the revoke request, it checks whether the file is still in use. If the file has since been closed, the first node will give up the whole token, and the second node will then be able to acquire a token covering the whole file. Thus, in the absence of concurrent write sharing, byte-range locking in GPFS behaves just like whole-file locking and is just as efficient, because a single token exchange is sufficient to access the whole file.

On the other hand, if the second node starts writing to a file *before* the first node closes the file, the first node will relinquish only part of its byte-range token. If the first node is writing sequentially at offset o_1 and the second node at offset o_2, the first node will relinquish its token from o_2 to infinity (if $o_2 > o_1$) or from zero to o_1 (if $o_2 < o_1$). This will allow both nodes to continue writing forward from their current write offsets without further token conflicts. In general, when multiple nodes

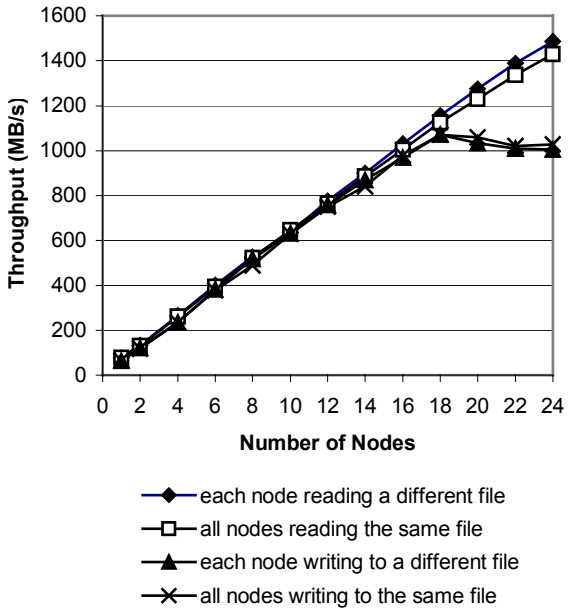


Figure 2: Read/Write Scaling

are writing sequentially to non-overlapping sections of the same file, each node will be able to acquire the necessary token with a single token exchange as part of its first write operation.

Information about write offsets is communicated during this token negotiation by specifying a *required range*, which corresponds to the offset and length of the write() system call currently being processed, and a *desired range*, which includes likely future accesses. For sequential access, the desired range will be from the current write offset to infinity. The token protocol will revoke byte ranges only from nodes that conflict with the required range; the token server will then grant as large a sub-range of the desired range as is possible without conflicting with ranges still held by other nodes.

The measurements shown in Figure 2 demonstrate how I/O throughput in GPFS scales when adding more file system nodes and more disks to the system. The measurements were obtained on a 32-node IBM RS/6000 SP system with 480 disks configured as 96 4+P RAID-5 devices attached to the SP switch through two I/O server nodes. The figure compares reading and writing a single large file from multiple nodes in parallel against each node reading or writing a different file. In the single-file test, the file was partitioned into *n* large, contiguous sections, one per node, and each node was reading or writing sequentially to one of the sections. The writes were updates in place to an existing file. The graph starts with a single file system node using four RAIDs on the left, adding four more RAIDs for each node added to the test, ending with 24 file system nodes using all 96 RAIDs on the right. It shows nearly linear scaling in all tested configurations for reads. In the test system, the data throughput was not limited by the disks, but by the RAID controller. The read throughput achieved by GPFS matched the throughput of raw disk reads through the I/O subsystem. The write throughput showed similar scalability. At 18 nodes the write throughput leveled off due to a problem in the switch adapter microcode.[2] The other point to note in this figure is that writing to a single file from multiple nodes in GPFS was just as fast as each node writing to a different file, demonstrating the effectiveness of the byte-range token protocol described above.

As long as the access pattern allows predicting the region of the file being accessed by a particular node in the near future, the token negotiation protocol will be able to minimize conflicts by carving up byte-range tokens among nodes accordingly. This applies not only to simple sequential access, but also to reverse sequential and forward or backward strided access patterns, provided each node operates in different, relatively large regions of the file (coarse-grain sharing).

As sharing becomes finer grain (each node writing to multiple, smaller regions), the token state and corresponding message traffic will grow. Note that byte-range tokens not only guarantee POSIX semantics but also synchronize I/O to the data blocks of the file. Since the smallest unit of I/O is one sector, the byte-range token granularity can be no smaller than one sector; otherwise, two nodes could write to the same sector at the same time, causing lost updates. In fact, GPFS uses byte-range tokens to synchronize data block allocation as well (see next section), and therefore rounds byte-range tokens to block boundaries. Hence multiple nodes writing into the same data block will cause token conflicts even if individual write operations do not overlap ("false sharing").

To optimize fine-grain sharing for applications that do not require POSIX semantics, GPFS allows disabling normal byte-range locking by switching to *data shipping* mode. File access switches to a method best

[2] The test machine had pre-release versions of the RS/6000 SP Switch2 adapters. In this early version of the adapter microcode, sending data from many nodes to a single I/O server node was not as efficient as sending from an I/O server node to many nodes.

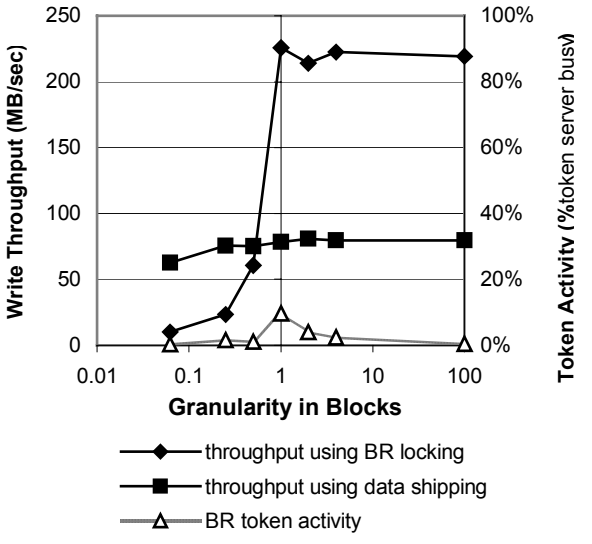


Figure 3: Effect of Sharing Granularity on Write Throughput

described as partitioned, centralized management. File blocks are assigned to nodes in a round-robin fashion, so that each data block will be read or written only by one particular node. GPFS forwards read and write operations originating from other nodes to the node responsible for a particular data block. For fine-grain sharing this is more efficient than distributed locking, because it requires fewer messages than a token exchange, and it avoids the overhead of flushing dirty data to disk when revoking a token. The eventual flushing of data blocks to disk is still done in parallel, since the data blocks of the file are partitioned among many nodes.

Figure 3 shows the effect of sharing granularity on write throughput, using data shipping and using normal byte-range locking. These measurements were done on a smaller SP system with eight file system nodes and two I/O servers, each with eight disks. Total throughput to each I/O server was limited by the switch.[3] We measured throughput with eight nodes updating fixed size records within the same file. The test used a strided access pattern, where the first node wrote records 0, 8, 16, …, the second node wrote records 1, 9, 17, …, and so on. The larger record sizes (right half of the figure) were multiples of the file system block size. Figure 3 shows that byte-range locking achieved nearly the full I/O throughput for these sizes, because they matched the granularity of the byte-range tokens. Updates with a record size smaller than one block (left half of the figure) required twice as much I/O, because of the required read-modify-write. The throughput for byte-range locking, however, dropped off far below the expected factor of one half due to token conflicts when multiple nodes wrote into the same data block. The resulting token revokes caused each data block to be read and written multiple times. The line labeled "BR token activity" plots token activity measured at the token server during the test of I/O throughput using byte-range locking. It shows that the drop in throughput was in fact due to the additional I/O activity and not an overload of the token server. The throughput curve for data shipping in Figure 3 shows that data shipping also incurred the read-modify-write penalty plus some additional overhead for sending data between nodes, but it dramatically outperformed byte-range locking for small record sizes that correspond to fine grain sharing. The data shipping implementation was intended to support fine-grain access, and as such does not try to avoid the read-modify-write for record sizes larger than a block. This fact explains why data shipping throughput stayed flat for larger record sizes.

Data shipping is primarily used by the MPI/IO library. MPI/IO does not require POSIX semantics, and provides a natural mechanism to define the collective that assigns blocks to nodes. The programming interfaces used by MPI/IO to control data shipping, however, are also available to other applications that desire this type of file access [13].

3.4 Synchronizing Access to File Metadata

Like other file systems, GPFS uses inodes and indirect blocks to store file attributes and data block addresses. Multiple nodes writing to the same file will result in concurrent updates to the inode and indirect blocks of the file to change file size and modification time (mtime) and to store the addresses of newly allocated data blocks. Synchronizing updates to the metadata on disk via exclusive write locks on the inode would result in a lock conflict on every write operation.

Instead, write operations in GPFS use a *shared write lock* on the inode that allows concurrent writers on multiple nodes. This shared write lock only conflicts with operations that require exact file size and/or mtime (a stat() system call or a read operation that attempts to read past end-of-file). One of the nodes accessing the file is designated as the *metanode* for the file; only the metanode reads or writes the inode from or to disk. Each writer updates a locally cached copy of the inode

[3] Here, the older RS/6000 SP Switch, with nominal 150 MB/sec throughput. Software overhead reduces this to approximately 125 MB/sec.

and forwards its inode updates to the metanode periodically or when the shared write token is revoked by a stat() or read() operation on another node. The metanode merges inode updates from multiple nodes by retaining the largest file size and latest mtime values it receives. Operations that update file size or mtime non-monotonically (trunc() or utimes()) require an exclusive inode lock.

Updates to indirect blocks are synchronized in a similar fashion. When writing a new file, each node independently allocates disk space for the data blocks it writes. The synchronization provided by byte-range tokens ensures that only one node will allocate storage for any particular data block. This is the reason that GPFS rounds byte-range tokens to block boundaries. Periodically or on revocation of a byte-range token, the new data block addresses are sent to the metanode, which then updates the cached indirect blocks accordingly.

Thus, GPFS uses distributed locking to guarantee POSIX semantics (e.g., a stat() system call sees the file size and mtime of the most recently completed write operation), but I/O to the inode and indirect blocks on disk is synchronized using a centralized approach (forwarding inode updates to the metanode). This allows multiple nodes to write to the same file without lock conflicts on metadata updates and without requiring messages to the metanode on every write operation.

The metanode for a particular file is elected dynamically with the help of the token server. When a node first accesses a file, it tries to acquire the *metanode token* for the file. The token is granted to the first node to do so; other nodes instead learn the identity of the metanode. Thus, in traditional workloads without concurrent file sharing, each node becomes metanode for the files it uses and handles all metadata updates locally.

When a file is no longer being accessed on the metanode and ages out of the cache on that node, the node relinquishes its metanode token and stops acting as metanode. When it subsequently receives a metadata request from another node, it sends a negative reply; the other node will then attempt to take over as metanode by acquiring the metanode token. Thus, the metanode for a file tends to stay within the set of nodes actively accessing that file.

3.5 Allocation Maps

The allocation map records the allocation status (free or in-use) of all disk blocks in the file system. Since each disk block can be divided into up to 32 subblocks to store data for small files, the allocation map contains 32 bits per disk block as well as linked lists for finding a free disk block or a subblock of a particular size efficiently.

Allocating disk space requires updates to the allocation map, which must be synchronized between nodes. For proper striping, a write operation must allocate space for a particular data block on a particular disk, but given the large block size used by GPFS, it is not as important where on that disk the data block is written. This fact allows organizing the allocation map in a way that minimizes conflicts between nodes by interleaving free space information about different disks in the allocation map as follows. The map is divided into a large, fixed number n of separately lockable *regions*, and each region contains the allocation status of $1/n^{th}$ of the disk blocks on every disk in the file system. This map layout allows GPFS to allocate disk space properly striped across all disks by accessing only a single allocation region at a time. This approach minimizes lock conflicts, because different nodes can allocate space from different regions. The total number of regions is determined at file system creation time based on the expected number of nodes in the cluster.

For each GPFS file system, one of the nodes in the cluster is responsible for maintaining free space statistics about all allocation regions. This *allocation manager* node initializes free space statistics by reading the allocation map when the file system is mounted. The statistics are kept loosely up-to-date via periodic messages in which each node reports the net amount of disk space allocated or freed during the last period. Instead of all nodes individually searching for regions that still contain free space, nodes ask the allocation manager for a region to try whenever a node runs out of disk space in the region it is currently using. To the extent possible, the allocation manager prevents lock conflicts between nodes by directing different nodes to different regions.

Deleting a file also updates the allocation map. A file created by a parallel program running on several hundred nodes might have allocated blocks in several hundred regions. Deleting the file requires locking and updating each of these regions, perhaps stealing them from the nodes currently allocating out of them, which could have a disastrous impact on performance.

Therefore, instead of processing all allocation map updates at the node on which the file was deleted, those that update regions known to be in use by other nodes are sent to those nodes for execution. The allocation manager periodically distributes hints about which

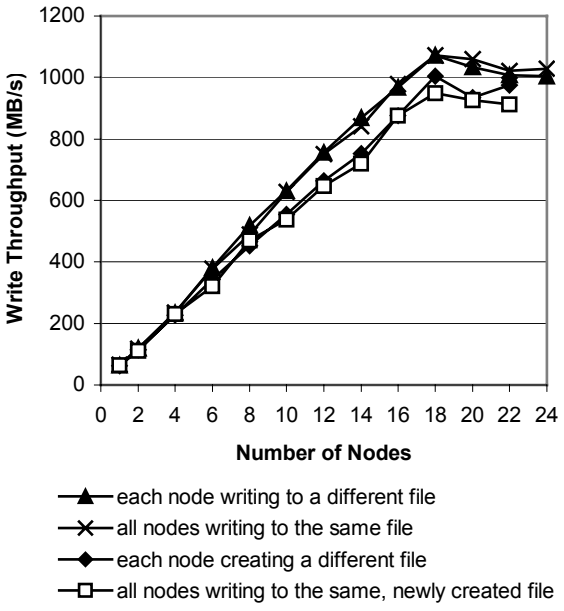


Figure 4: File Create Scaling

regions are in use by which nodes to facilitate shipping deallocation requests.

To demonstrate the effectiveness of the allocation manager hints as well as the metanode algorithms described in the previous section we measured write throughput for updates in place to an existing file against creation of a new file (Figure 4). As in Figure 2, we measured all nodes writing to a single file (using the same access pattern as in Figure 2), as well as each node writing to a different file. The measurements were done on the same hardware as Figure 2, and the data points for the write throughput are in fact the same points shown in the earlier figure. Due to the extra work required to allocate disk storage, throughput for file creation was slightly lower than for update in place. Figure 4 shows, however, that create throughput still scaled nearly linearly with the number of nodes, and that creating a single file from multiple nodes was just as fast as each node creating a different file.

3.6 Other File System Metadata

A GPFS file system contains other global metadata, including file system configuration data, space usage quotas, access control lists, and extended attributes. Space does not permit a detailed description of how each of these types of metadata is managed, but a brief mention is in order. As in the cases described in previous sections, GPFS uses distributed locking to protect the consistency of the metadata on disk, but in most cases uses more centralized management to coordinate or collect metadata updates from different nodes. For example, a quota manager hands out relatively large increments of disk space to the individual nodes writing a file, so quota checking is done locally, with only occasional interaction with the quota manager.

3.7 Token Manager Scaling

The token manager keeps track of all lock tokens granted to all nodes in the cluster. Acquiring, relinquishing, upgrading, or downgrading a token requires a message to the token manager. One might reasonably expect that the token manager could become a bottleneck in a large cluster, or that the size of the token state might exceed the token manager's memory capacity.

One way to address these issues would be to partition the token space and distribute the token state among several nodes in the cluster. We found, however, that this is not the best way — or at least not the most important way — to address token manager scaling issues, for the following reasons.

A straightforward way to distribute token state among nodes might be to hash on the file inode number. Unfortunately, this does not address the scaling issues arising from parallel access to a single file. In the worst case, concurrent updates to a file from multiple nodes generate a byte-range token for each data block of a file. Because the size of a file is effectively unbounded, the size of the byte-range token state for a single file is also unbounded. One could conceivably partition byte-range token management for a single file among multiple nodes, but this would make the frequent case of a single node acquiring a token for a whole file prohibitively expensive. Instead, the token manager prevents unbounded growth of its token state by monitoring its memory usage and, if necessary, revoking tokens to reduce the size of the token state[4].

The most likely reason for a high load on the token manager node is lock conflicts that cause token revocation. When a node downgrades or relinquishes a token, dirty data or metadata covered by that token must be flushed to disk and/or discarded from the cache. As explained earlier (see Figure 3), the cost of disk I/O caused by token conflicts dominates the cost of token manager messages. Therefore, a much more

[4] Applications that do not require POSIX semantics can, of course, use data shipping to bypass byte-range locking and avoid token state issues.

effective way to reduce token manager load and improve overall performance is to avoid lock conflicts in the first place. The allocation manager hints described in Section 3.5 are an example of avoiding lock conflicts.

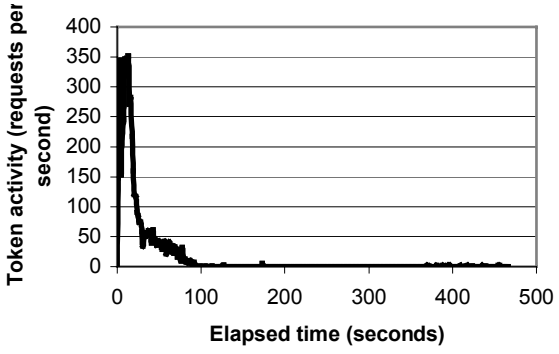
Finally, GPFS uses a number of optimizations in the token protocol that significantly reduce the cost of token management and improve response time as well. When it is necessary to revoke a token, it is the responsibility of the revoking node to send revoke messages to all nodes that are holding the token in a conflicting mode, to collect replies from these nodes, and to forward these as a single message to the token manager. Acquiring a token will never require more than two messages to the token manager, regardless of how many nodes may be holding the token in a conflicting mode.

The protocol also supports token prefetch and token request batching, which allow acquiring multiple tokens in a single message to the token manager. For example, when a file is accessed for the first time, the necessary inode token, metanode token, and byte-range token to read or write the file are acquired with a single token manager request.

When a file is deleted on a node, the node does not immediately relinquish the tokens associated with that file. The next file created by the node can then re-use the old inode and will not need to acquire any new tokens. A workload where users on different nodes create and delete files under their respective home directories will generate little or no token traffic.

Figure 5 demonstrates the effectiveness of this optimization. It shows token activity while running a multi-user file server workload on multiple nodes. The workload was generated by the dbench program [10], which simulates the file system activity of the NetBench [11] file server benchmark. It ran on eight file system nodes, with each node running a workload for 10 NetBench clients. Each client ran in a different subdirectory. The figure shows an initial spike of token activity as the benchmark started up on all of the nodes and each node acquired tokens for the files accessed by its clients. Even though the benchmark created and deleted many files throughout the run, each node reused a limited number of inodes. Once all nodes had obtained a sufficient number of inodes, token activity quickly dropped to near zero. Measurements of the CPU load on the token server indicated that it is capable of supporting between 4000 and 5000 token requests per second, so the peak request rate shown in Figure 5 consumed only a small fraction of the token server capacity. Even the height of the peak was only an artifact of starting the benchmark at the same time on all of the nodes, which would not be likely to happen under a real multi-user workload.

Figure 5: Token Activity During Dbench Run



4 Fault Tolerance

As a cluster is scaled up to large numbers of nodes and disks it becomes increasingly unlikely that all components are working correctly at all times. This implies the need to handle component failures gracefully and continue operating in the presence of failures.

4.1 Node Failures

When a node fails, GPFS must restore metadata being updated by the failed node to a consistent state, must release resources held by the failed node (lock tokens), and it must appoint replacements for any special roles played by the failed node (e.g., metanode, allocation manager, or token manager).

Since GPFS stores recovery logs on shared disks, metadata inconsistencies due to a node failure are quickly repaired by running log recovery from the failed node's log on one of the surviving nodes. After log recovery is complete, the token manager releases tokens held by the failed node. The distributed locking protocol ensures that the failed node must have held tokens for all metadata it had updated in its cache but had not yet written back to disk at the time of the failure. Since these tokens are only released after log recovery is complete, metadata modified by the failed node will not be accessible to other nodes until it is known to be in a consistent state. This observation is true even in cases where GPFS uses a more centralized approach to synchronizing metadata updates, for example, the file size and mtime updates that are collected by the metanode. Even though the write operations causing such updates are not synchronized

via distributed locking, the updates to the metadata on disk are still protected by the distributed locking protocol — in this case, through the metanode token.

After log recovery completes, other nodes can acquire any metanode tokens that had been held by the failed node and thus take over the role of metanode. If another node had sent metadata updates to the old metanode but, at the time of the failure, had not yet received an acknowledgement that the updates were committed to disk, it re-sends the updates to the new metanode. These updates are idempotent, so the new metanode can simply re-apply them.

Should the token manager fail, another node will take over this responsibility and reconstruct the token manager state by querying all surviving nodes about the tokens they currently hold. Since the new token manager does not know what tokens were held by failed nodes, it will not grant any new tokens until log recovery is complete. Tokens currently held by the surviving nodes are not affected by this.

Similarly, other special functions carried out by a failed node (e.g., allocation manager) are assigned to another node, which rebuilds the necessary state by reading information from disk and/or querying other nodes.

4.2 Communication Failures

To detect node failures GPFS relies on a group services layer that monitors nodes and communication links via periodic heartbeat messages and implements a process group membership protocol [12]. When a node fails, the group services layer informs the remaining nodes of a group membership change. This triggers the recovery actions described in the previous section.

A communication failure such as a bad network adapter or a loose cable may cause a node to become isolated from the others, or a failure in the switching fabric may cause a network partition. Such a partition is indistinguishable from a failure of the unreachable nodes. Nodes in different partitions may still have access to the shared disks and would corrupt the file system if they were allowed to continue operating independently of each other. For this reason, GPFS allows accessing a file system only by the group containing a majority of the nodes in the cluster; the nodes in the minority group will stop accessing any GPFS disk until they can re-join a majority group.

Unfortunately, the membership protocol cannot guarantee how long it will take for each node to receive and process a failure notification. When a network partition occurs, it is not known when the nodes that are no longer members of the majority will be notified and stop accessing shared disks. Therefore, before starting log recovery in the majority group, GPFS *fences* nodes that are no longer members of the group from accessing the shared disks, i.e., it invokes primitives available in the disk subsystem to stop accepting I/O requests from the other nodes.

To allow fault-tolerant two-node configurations, a communication partition in such a configuration is resolved exclusively through disk fencing instead of using a majority rule: upon notification of the failure of the other node, each node will attempt to fence all disks from the other node in a predetermined order. In case of a network partition, only one of the two nodes will be successful and can continue accessing GPFS file systems.

4.3 Disk Failures

Since GPFS stripes data and metadata across all disks that belong to a file system, the loss of a single disk will affect a disproportionately large fraction of the files. Therefore, typical GPFS configurations use dual-attached RAID controllers, which are able to mask the failure of a physical disk or the loss of an access path to a disk. Large GPFS file systems are striped across multiple RAID devices. In such configurations, it is important to match the file system block size and alignment with RAID stripes so that data block writes do not incur a write penalty for the parity update.

As an alternative or a supplement to RAID, GPFS supports replication, which is implemented in the file system. When enabled, GPFS allocates space for two copies of each data or metadata block on two different disks and writes them to both locations. When a disk becomes unavailable, GPFS keeps track of which files had updates to a block with a replica on the unavailable disk. If and when the disk becomes available again, GPFS brings stale data on the disk up-to-date by copying the data from another replica. If a disk fails permanently, GPFS can instead allocate a new replica for all affected blocks on other disks.

Replication can be enabled separately for data and metadata. In cases where part of a disk becomes unreadable (bad blocks), metadata replication in the file system ensures that only a few data blocks will be affected, rather than rendering a whole set of files inaccessible.

5 Scalable Online System Utilities

Scalability is important not only for normal file system operations, but also for file system utilities. These

utilities manipulate significant fractions of the data and metadata in the file system, and therefore benefit from parallelism as much as parallel applications.

For example, GPFS allows growing, shrinking, or reorganizing a file system by adding, deleting, or replacing disks in an existing file system. After adding new disks, GPFS allows rebalancing the file system by moving some of the existing data to the new disks. To remove or replace one or more disks in a file system, GPFS must move all data and metadata off the affected disks. All of these operations require reading all inodes and indirect blocks to find the data that must be moved. Other utilities that need to read all inodes and indirect blocks include defragmentation, quota-check, and fsck[5] [13]. Also, when replication is enabled and a group of disks that were down becomes available again, GPFS must perform a metadata scan to find files with missing updates that need to be applied to these disks.

Finishing such operations in any reasonable amount of time requires exploiting the parallelism available in the system. To this end, GPFS appoints one of the nodes as a *file system manager* for each file system, which is responsible for coordinating such administrative activity. The file system manager hands out a small range of inode numbers to each node in the cluster. Each node processes the files in its assigned range and then sends a message to the file system manager requesting more work. Thus, all nodes work on different subsets of files in parallel, until all files have been processed. During this process, additional messages may be exchanged between nodes to compile global file system state. While running fsck, for example, each node collects block references for a different section of the allocation map. This allows detecting inter-file inconsistencies, such as a single block being assigned to two different files.

Even with maximum parallelism, these operations can take a significant amount of time on a large file system. For example, a complete rebalancing of a multi-terabyte file system may take several hours. Since it is unacceptable for a file system to be unavailable for such a long time, GPFS allows all of its file system utilities to run on-line, i.e., while the file system is mounted and accessible to applications. The only exception is a full file system check for diagnostic purposes (fsck), which requires the file system to be unmounted. File system utilities use normal distributed locking to synchronize with other file activity. Special synchronization is required only while reorganizing higher-level metadata (allocation maps and the inode file). For example, when it is necessary to move a block of inodes from one disk to another, the node doing so acquires a special range lock on the inode file, which is more efficient and less disruptive than acquiring individual locks on all inodes within the block.

6 Experiences

GPFS is installed at several hundred customer sites, on clusters ranging from a few nodes with less than a terabyte of disk, up to the 512-node ASCI White system with its 140 terabytes of disk space in two file systems. Much has been learned that has affected the design of GPFS as it has evolved. These lessons would make a paper in themselves, but a few are of sufficient interest to warrant relating here.

Several of our experiences pointed out the importance of intra-node as well as inter-node parallelism and for properly balancing load across nodes in a cluster. In the initial design of the system management commands, we assumed that distributing work by starting a thread on each node in the cluster would be sufficient to exploit all available disk bandwidth. When rebalancing a file system, for example, the strategy of handing out ranges of inodes to one thread per node, as described in Section 5, was able to generate enough I/O requests to keep all disks busy. We found, however, that we had greatly underestimated the amount of skew this strategy would encounter. Frequently, a node would be handed an inode range containing significantly more files or larger files than other ranges. Long after other nodes had finished their work, this node was still at work, running single-threaded, issuing one I/O at a time.

The obvious lesson is that the granules of work handed out must be sufficiently small and of approximately equal size (ranges of blocks in a file rather than entire files). The other lesson, less obvious, is that even on a large cluster, intra-node parallelism is often a more efficient road to performance than inter-node parallelism. On modern systems, which are often high-degree SMPs with high I/O bandwidth, relatively few nodes can saturate the disk system[6]. Exploiting the available bandwidth by running multiple threads per node, for example, greatly reduces the effect of workload skew.

Another important lesson is that even though the small amount of CPU consumed by GPFS centralized

[5] For diagnostic purposes to verify file system consistency, not part of a normal mount.

[6] Sixteen nodes drive the entire 12 GB/s I/O bandwidth of ASCI White.

management functions (e.g., the token manager) normally does not affect application performance, it can have a significant impact on highly parallel applications. Such applications often run in phases with barrier synchronization points. As described in Section 5, centralized services are provided by the file system manager, which is dynamically chosen from the nodes in the cluster. If this node also runs part of the parallel application, the management overhead will cause it to take longer to reach its barrier, leaving the other nodes idle. If the overhead slows down the application *by even one percent*, the idle time incurred by other nodes will be equivalent to leaving five nodes unused on the 512-node ASCI White system! To avoid this problem, GPFS allows restricting management functions to a designated set of administrative nodes. A large cluster can dedicate one or two administrative nodes, avoid running load sensitive, parallel applications on them, and actually increase the available computing resource.

Early versions of GPFS had serious performance problems with programs like "ls –l" and incremental backup, which call stat() for each file in a directory. The stat() call reads the file's inode, which requires a read token. If another node holds this token, releasing it may require dirty data to be written back on that node, so obtaining the token can be expensive. We solved this problem by exploiting parallelism. When GPFS detects multiple accesses to inodes in the same directory, it uses multiple threads to prefetch inodes for other files in the same directory. Inode prefetch speeds up directory scans by almost a factor of ten.

Another lesson we learned on large systems is that even the rarest failures, such as data loss in a RAID, will occur. One particularly large GPFS system experienced a microcode failure in a RAID controller caused by an intermittent problem during replacement of a disk. The failure rendered three sectors of an allocation map unusable. Unfortunately, an attempt to allocate out of one of these sectors generated an I/O error, which caused the file system to take itself off line. Running log recovery repeated the attempt to write the sector and the I/O error. Luckily no user data was lost, and the customer had enough free space in other file systems to allow the broken 14 TB file system to be mounted read-only and copied elsewhere. Nevertheless, many large file systems now use GPFS metadata replication *in addition* to RAID to provide an extra measure of security against dual failures.

Even more insidious than rare, random failures are systematic ones. One customer was unfortunate enough to receive several hundred disk drives from a bad batch with an unexpectedly high failure rate. The customer wanted to replace the bad drives without taking the system down. One might think this could be done by successively replacing each drive and letting the RAID rebuild, but this would have greatly increased the possibility of a dual failure (i.e., a second drive failure in a rebuilding RAID parity group) and a consequent catastrophic loss of the file system. The customer chose as a solution to delete a small number of disks (here, RAID parity groups) from the file system, during which GPFS rebalances data from the deleted disks onto the remaining disks. Then new disks (parity groups) were created from the new drives and added back to the file system (again rebalancing). This tedious process was repeated until all disks were replaced, without taking the file system down and without compromising its reliability. Lessons include not assuming independent failures in a system design, and the importance of online system management and parallel rebalancing.

7 Related Work

One class of file systems extends the traditional file server architecture to a storage area network (SAN) environment by allowing the file server clients to access data directly from disk through the SAN. Examples of such SAN file systems are IBM/Tivoli SANergy [14], and Veritas SANPoint Direct [15]. These file systems can provide efficient data access for large files, but, unlike GPFS, all metadata updates are still handled by a centralized metadata server, which makes this type of architecture inherently less scalable. SAN file systems typically do not support concurrent write sharing, or sacrifice POSIX semantics to do so. For example, SANergy allows multiple clients to read and write to the same file through a SAN, but provides no consistent view of the data unless explicit fcntl locking calls are added to the application program.

SGI's XFS file system [16] is designed for similar, large-scale, high throughput applications that GPFS excels at. It stores file data in large, variable length extents and relies on an underlying logical volume manager to stripe the data across multiple disks. Unlike GPFS however, XFS is not a cluster file system; it runs on large SMPs. CXFS [17] is a cluster version of XFS that allows multiple nodes to access data on shared disks in an XFS file system. However, only one of the nodes handles all metadata updates, like other SAN file systems mentioned above.

Frangipani [18] is a shared-disk cluster file system that is similar in principle to GPFS. It is based on the same, symmetric architecture, and uses similar logging, locking and recovery algorithms based on write-ahead logging with separate logs for each node stored on

shared disk. Like GPFS, it uses a token-based distributed lock manager. A Frangipani file system resides on a single, large (2^{64} byte) virtual disk provided by Petal [19], which redirects I/O requests to a set of Petal servers and handles physical storage allocation and striping. This layered architecture simplifies metadata management in the file system to some extent. The granularity of disk space allocation (64kB) in Petal, however, is too large and its virtual address space is too small to simply reserve a fixed, contiguous virtual disk area (e.g., 1TB) for each file in a Frangipani file system. Therefore, Frangipani still needs its own allocation maps to manage the virtual disk space provided by Petal. Unlike GPFS, Frangipani is mainly "targeted for environments with program development and engineering workloads". It implements whole-file locking only and therefore does not allow concurrent writes to the same file from multiple nodes.

Another example of a shared-disk cluster file system is the Global File System (GFS) [20], which originated as an open source file system for Linux. The newest version (GFS-4) implements journaling, and uses logging, locking, and recovery algorithms similar to those of GPFS and Frangipani. Locking in GFS is closely tied to physical storage. Earlier versions of GFS [21] required locking to be implemented at the disk device via extensions to the SCSI protocol. Newer versions allow the use of an external distributed lock manager, but still lock individual disk blocks of 4kB or 8kB size. Therefore, accessing large files in GFS entails significantly more locking overhead than the byte-range locks used in GPFS. Similar to Frangipani/Petal, striping in GFS is handled in a "Network Storage Pool" layer; once created, however, the stripe width cannot be changed (it is possible to add a new "sub-pools", but striping is confined to a sub-pool, i.e., GFS will not stripe across sub-pools). Like Frangipani, GFS is geared more towards applications with little or no intra-file sharing.

8 Summary and Conclusions

GPFS was built on many of the ideas that were developed in the academic community over the last several years, particularly distributed locking and recovery technology. To date it has been a matter of conjecture how well these ideas scale. We have had the opportunity to test those limits in the context of a product that runs on the largest systems in existence.

One might question whether distributed locking scales, in particular, whether lock contention for access to shared metadata might become a bottleneck that limits parallelism and scalability. Somewhat to our surprise, we found that distributed locking scales quite well. Nevertheless, several significant changes to conventional file system data structures and locking algorithms yielded big gains in performance, both for parallel access to a single large file and for parallel access to large numbers of small files. We describe a number of techniques that make distributed locking work in a large cluster: byte-range token optimizations, dynamic selection of meta nodes for managing file metadata, segmented allocation maps, and allocation hints.

One might similarly question whether conventional availability technology scales. Obviously there are more components to fail in a large system. Compounding the problem, large clusters are so expensive that their owners demand high availability. Add to this the fact that file systems of tens of terabytes are simply too large to back up and restore. Again, we found the basic technology to be sound. The surprises came in the measures that were necessary to provide data integrity and availability. GPFS replication was implemented because at the time RAID was more expensive than replicated conventional disk. RAID has taken over as its price has come down, but even its high level of integrity is not sufficient to guard against the loss of a hundred terabyte file system.

Existing GPFS installations show that our design is able to scale up to the largest super computers in the world and to provide the necessary fault tolerance and system management functions to manage such large systems. Nevertheless, we expect the continued evolution of technology to demand ever more scalability. The recent interest in Linux clusters with inexpensive PC nodes drives the number of components up still further. The price of storage has decreased to the point that customers are seriously interested in petabyte file systems. This trend makes file system scalability an area of interest for research that will continue for the foreseeable future.

9 Acknowledgements

A large number of people at several IBM locations have contributed to the design and implementation of GPFS over the years. Although space does not allow naming all of them here, the following people have significantly contributed to the work reported here: Jim Wyllie, Dan McNabb, Tom Engelsiepen, Marc Eshel, Carol Hartman, Mike Roberts, Wayne Sawdon, Dave Craft, Brian Dixon, Eugene Johnson, Scott Porter, Bob Curran, Radha Kandadai, Lyle Gayne, Mike Schouten, Dave Shapiro, Kuei-Yu Wang Knop, Irit Loy, Benny Mandler, John Marberg, Itai Nahshon, Sybille Schaller, Boaz Shmueli, Roman Talyanski, and Zvi Yehudai.

References

[1] Roger L. Haskin: *Tiger Shark - a scalable file system for multimedia*, IBM Journal of Research and Development, Volume 42, Number 2, March 1998, pp. 185-197.

[2] C. Mohan, Inderpal Narang: Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. VLDB 1991: 193-207.

[3] *IBM builds world's fastest supercomputer to simulate nuclear testing for U.S. Energy Department*. IBM Press Release, Poughkeepsie, N.Y., June 29, 2000. http://www.ibm.com/servers/eserver/pseries/news/pressreleases/2000/jun/asci_white.html

[4] ASCI White. http://www.rs6000.ibm.com/hardware/largescale/supercomputers/asciwhite/

[5] *ASCI White*. http://www.llnl.gov/asci/platforms/white/home/

[6] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, H. Raymond Strong, *Extendible hashing - a fast access method for dynamic files*, ACM Transactions on Database Systems, New York, NY, Volume 4 Number 3, 1979, pages 315-344.

[7] Frank B. Schmuck, James Christopher Wyllie, and Thomas E. Engelsiepen. *Parallel file system and method with extensible hashing*. US Patent No. 05893086.

[8] J. N. Gray, *Notes on database operating systems*, in *Operating systems, an advanced course*, edited by R. Bayer et. al., Springer-Verlag, Berlin, Germany, 1979, pages 393-400.

[9] Ajay Mohindra and Murthy Devarakonda, *Distributed token management in the Calypso file system*, Proceedings of the IEEE Symposium on Parallel and Distributed Processing, New York, 1994.

[10] Dbench benchmark. Available from ftp://samba.org/pub/tridge/dbench/

[11] NetBench benchmark. http://etestinglabs.com/benchmarks/netbench/netbench.asp

[12] *Group Services Programming Guide and Reference*, RS/6000 Cluster Technology, Document Number SA22-7355-01, Second Edition (April 2000), International Business Machines Corporation, 2455 South Road, Poughkeepsie, NY 12601-5400, USA. Also available from http://www.rs6000.ibm.com/resource/aix_resource/sp_books/pssp/index.html

[13] *IBM General Parallel File System for AIX: Administration and Programming Reference*. Document Number SA22-7452-02, Second Edition (December 2000), International Business Machines Corporation, 2455 South Road, Poughkeepsie, NY 12601-5400, USA. Also available from http://www.rs6000.ibm.com/resource/aix_resource/sp_books/gpfs/index.html

[14] Charlotte Brooks, Ron Henkhaus, Udo Rauch, Daniel Thompson. *A Practical Guide to Tivoli SANergy*. IBM Redbook SG246146, June 2001, available from http://www.redbooks.ibm.com/

[15] *VERITAS SANPoint Direct File Access*. Whitepaper, August 2000. VERITAS Software Corporation, Corporate Headquarters, 1600 Plymouth Street, Mountain View, CA 94043.

[16] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. *Scalability in the XFS File System*, Proceedings of the USENIX 1996 Technical Conference, pages 1-14, San Diego, CA, USA, 1996.

[17] *SGI CXS Clustered File System*, Datasheet, Silicon Graphics, Inc., 1600 Amphitheatre Pkwy. Mountain View, CA 94043.

[18] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. *Frangipani: A Scalable Distributed File System*. In Proceedings of the Symposium on Operating Systems Principles, 1997, pages 224-237.

[19] Edward K. Lee and Chandramohan A. Thekkath. *Petal: Distributed Virtual Disks*, In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996, pages 84-92.

[20] Kenneth W. Preslan, Andrew P. Barry, Jonathan Brassow, Russell Cattelan, Adam Manthei, Erling Nygaard, Seth Van Oort, David Teigland, Mike Tilstra, Matthew O'Keefe, Grant Erickson and Manish Agarwal. *Implementing Journaling in a Linux Shared Disk File System.* Seventeenth IEEE Symposium on Mass Storage Systems, March 2000, pages 351-378.

[21] 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. *A 64-bit, Shared Disk File System for Linux.* Sixteenth IEEE Mass Storage Systems Symposium, March 15-18, 1999, San Diego, California, pages 351-378.