The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Scalability in the XFS File System

Adam Sweeney
Silicon Graphics

Scalability in the XFS File System

Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck
Silicon Graphics, Inc.

Abstract

In this paper we describe the architecture and design of a new file system, XFS, for Silicon Graphics' IRIX operating system. It is a general purpose file system for use on both workstations and servers. The focus of the paper is on the mechanisms used by XFS to scale capacity and performance in supporting very large file systems. The large file system support includes mechanisms for managing large files, large numbers of files, large directories, and very high performance I/O.

In discussing the mechanisms used for scalability we include both descriptions of the XFS on-disk data structures and analyses of why they were chosen. We discuss in detail our use of B+ trees in place of many of the more traditional linear file system structures.

XFS has been shipping to customers since December of 1994 in a version of IRIX 5.3, and we are continuing to improve its performance and add features in upcoming releases. We include performance results from running on the latest version of XFS to demonstrate the viability of our design.

1. Introduction

XFS is the next generation local file system for Silicon Graphics' workstations and servers. It is a general purpose Unix file system that runs on workstations with 16 megabytes of memory and a single disk drive and also on large SMP network servers with gigabytes of memory and terabytes of disk capacity. In this paper we describe the XFS file system with a focus on the mechanisms it uses to manage large file systems on large computer systems.

The most notable mechanism used by XFS to increase the scalability of the file system is the pervasive use of B+ trees [Comer79]. B+ trees are used for tracking free extents in the file system rather than bitmaps. B+ trees are used to index directory entries rather than using linear lookup structures. B+ trees are used to manage file extent maps that overflow the number of direct pointers kept in the inodes. Finally, B+ trees are used to keep track of dynamically allocated inodes scattered throughout the file system. In addition, XFS uses an asynchronous write ahead logging scheme for protecting complex metadata updates and allowing fast file system recovery after a crash. We also support very high throughput file I/O using large, parallel I/O requests and DMA to transfer data directly between user buffers and the underlying disk drives. These mechanisms allow us to recover even very large file systems after a crash in typically less than 15 seconds, to manage very large file systems efficiently, and to perform file I/O at hardware speeds that can exceed 300 MB/sec.

XFS has been shipping to customers since December of 1994 in a version of IRIX 5.3, and XFS will be the default file system installed on all SGI systems starting with the release of IRIX 6.2 in early 1996. The file system is stable and is being used on production servers throughout Silicon Graphics and at many of our customers' sites.

In the rest of this paper we describe why we chose to focus on scalability in the design of XFS and the mechanisms that are the result of that focus. We start with an explanation of why we chose to start from scratch rather than enhancing the old IRIX file system. We next describe the overall architecture of XFS, followed by the specific mechanisms of XFS which allow it to scale in both capacity and performance. Finally, we present performance results from running on real systems to demonstrate the success of the XFS design.

2. Why a New File System?

The file system literature began predicting the coming of the "I/O bottleneck" years ago [Ousterhout90], and we experienced it first hand at SGI. The problem was not the I/O performance of our hardware, but the limitations imposed by the old IRIX file system, EFS [SGI92]. EFS is similar to the Berkeley Fast File System [McKusick84] in structure, but it uses extents rather than individual blocks for file space allocation and I/O. EFS could not support file systems greater than 8 gigabytes in size, files greater than 2 gigabytes in size, or give applications access to the full I/O bandwidth of the hardware on which they were running. EFS was not designed with large systems and file systems in mind, and it was faltering under the pressure coming from the needs of new applications and the capabilities of new hardware. While we considered enhancing EFS to meet these new demands, the required changes were so great that we decided it would be better to replace EFS with an entirely new file system designed with these new demands in mind.

One example of a new application that places new demands on the file system is the storage and retrieval of uncompressed video. This requires approximately 30 MB/sec of I/O throughput for a single stream, and just one hour of such video requires 108 gigabytes of disk storage. While most people work with compressed video to make the I/O throughput and capacity requirements easier to manage, many customers, for example those involved in professional video editing, come to SGI looking to work with uncompressed video. Another video storage example that influenced the design of XFS is a video on demand server, such as the one being deployed by Time Warner in Orlando. These servers store thousands of compressed movies. One thousand typical movies take up around 2.7 terabytes of disk space. Playing two hundred high quality 0.5 MB/sec MPEG streams concurrently uses 100 MB/sec of I/O bandwidth. Applications with similar requirements are appearing in database and scientific computing, where file system scalability and performance is sometimes more important than CPU performance. The requirements we derived from these applications were support for terabytes of disk space, huge files, and hundreds of megabytes per second of I/O bandwidth.

We also needed to ensure that the file system could provide access to the full capabilities and capacity of our hardware, and to do so with a minimal amount of overhead. This meant supporting systems with multiple terabytes of disk capacity. With today's 9 gigabyte disk drives it only takes 112 disk drives to surpass 1 terabyte of storage capacity. These requirements also meant providing access to the large amount of disk bandwidth that is available in such high capacity systems. Today, SGI's high end systems have demonstrated sustained disk bandwidths in excess of 500 megabytes per second. We needed to make that bandwidth available to applications using the file system. Finally, these requirements meant doing all of this without using unreasonable portions of the available CPU and memory on the systems.

3. Scalability Problems Addressed by XFS

In designing XFS, we focused in on the specific problems with EFS and other existing file systems that we felt we needed to address. In this section we consider several of the specific scalability problems addressed in the design of XFS and why the mechanisms used in other file systems are not sufficient.

Slow Crash Recovery

A file system with a crash recovery procedure that is dependent on the file system size cannot be practically used on large systems, because the data on the system is unavailable for an unacceptably long period after a crash. EFS and file systems based on the BSD Fast File System [McKusick84] falter in this area due to their dependence on a file system scavenger program to restore the file system to a consistent state after a crash. Running fsck over an 8 gigabyte file system with a few hundred thousand inodes today takes a few minutes. This is already too slow to satisfy modern availability requirements, and the time it takes to recover in this way only gets worse when applied to larger file systems with more files. Most recently designed file systems apply database recovery techniques to their metadata recovery procedure to avoid this pitfall.

Inability to Support Large File Systems

We needed a file system that could manage even petabytes of storage, but all of the file systems we know of are limited to either a few gigabytes or a few terabytes in size. EFS is limited to only 8 gigabytes in size. These limitations stem from the use of data structures that don't scale, for example the bitmap in EFS, and from the use of 32 bit block pointers throughout the on-disk structures of the file system. The 32 bit block pointers can address at most 4 billion blocks, so even with an 8 kilobyte block size the file system is limited to a theoretical maximum of 32 terabytes in size.

Inability to Support Large, Sparse Files

None of the file systems we looked at support full 64 bit, sparse files. EFS did not support sparse files at all. Most others use the block mapping scheme created for FFS. We decided early on that we would manage space in files with variable length extents (which we will describe later), and the FFS style scheme does not work with variable length extents. Entries in the FFS block map point to individual blocks in the file, and up to three levels of indirect blocks can be used to track blocks throughout the file. This scheme requires that all entries in the map point to extents of the same size. This is because it does not store the offset of each entry in the map with the entry, and thus forces each entry to be in a fixed location in the tree so that it can be found. Also, a 64 bit file address space cannot be supported at all without adding more levels of indirection to the FFS block map.

Inability to Support Large, Contiguous Files

Another problem is that the mechanisms in many other file systems for allocating large, contiguous files do not scale well. Most, including EFS, use linear bitmap structures for tracking free and allocated blocks in the file system. Finding large regions of contiguous space in such bitmaps in large file systems is not efficient. For EFS this has become a significant bottleneck in the performance of writing newly allocated files. For other file systems, for example FFS, this has not been a problem up to this point, because they do not try very hard to allocate files contiguously. Not doing so, however, can have bad implications for the I/O performance of accessing files in those file systems [Seltzer95].

Inability to Support Large Directories

Another area which has not been addressed by other Unix file systems is support for directories with more than a few thousand entries. While some, for example Episode [Chutani92] and VxFS [Veritas95], at least speed up searching for entries within a directory block via hashing, most file systems use directory structures which require a linear scan of the directory blocks in searching for a particular file. The lookup and update performance of these unindexed formats degrades linearly with the size of the directory. Others use in-memory hashing schemes layered over simple on-disk structures [Hitz94]. These in memory schemes work well to a point, but in very large directories they require a large amount of memory. This problem has been addressed in some non-Unix file systems, like NTFS [Custer94] and Cedar [Hagmann87], by using B trees to index the entries in the directory.

Inability to Support Large Numbers of Files

While EFS and other file system can theoretically support very large numbers of files in a file system, in practice they do not. The reason is that the number of inodes allocated in these file systems is fixed at the time the file system is created. Choosing a very large number of inodes up front wastes the space allocated to those inodes when they are not actually used. The real number of files that will reside in a file system is rarely known at the time the file system is created. Being forced to choose makes the management of large file systems more difficult than it should be. Episode [Chutani92] and VxFS [Veritas95] both solve this problem by allowing the number of inodes in the file system to be increased dynamically.

In summary, there are several problems with EFS and other file systems that we wanted to address in the design of XFS. While these problems may not have been important in the past, we believe the rules of file system design have changed. The rest of this paper describes XFS and the ways in which it solves the scalability problems described here.

4. XFS Architecture

Figure 1. gives a block diagram of the general structure of the XFS file system.

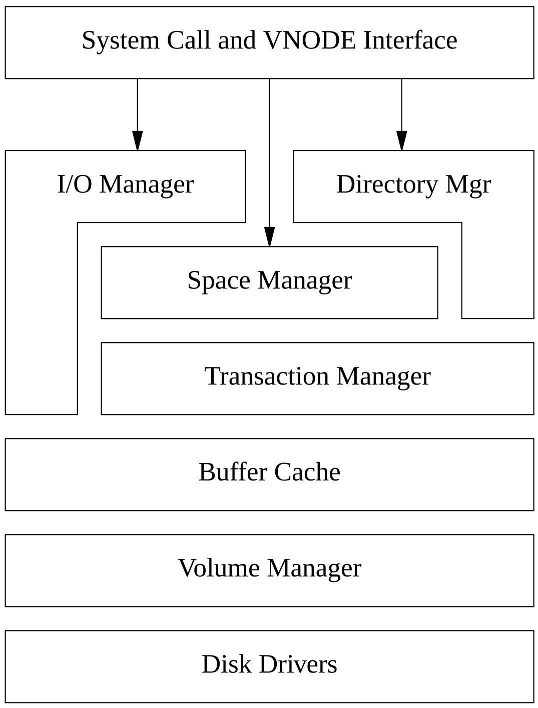

Figure 1. XFS Architecture

The high level structure of XFS is similar to a conventional file system with the addition of a transaction manager and a volume manager. XFS supports all of the standard Unix file interfaces and is entirely POSIX and XPG4 compliant. It sits below the vnode interface [Kleiman86] in the IRIX kernel and takes full advantage of services provided by the kernel, including the buffer/page cache, the directory name lookup cache, and the dynamic vnode cache.

XFS is modularized into several parts, each of which is responsible for a separate piece of the file system's functionality. The central and most important piece of the file system is the space manager. This module manages the file system free space, the allocation of inodes, and the allocation of space within individual files. The I/O manager is responsible for satisfying file I/O requests and depends on the space manager for allocating and keeping track of space for files. The directory manager implements the XFS file system name space. The buffer cache is used by all of these pieces to cache the contents of frequently accessed blocks from the underlying volume in memory. It is an integrated page and file cache shared by all file systems in the kernel. The transaction manager is used by the other pieces of the file system to make all updates to the metadata of the file system atomic. This enables the quick recovery of the file system after a crash. While the XFS implementation is modular, it is also large and complex. The current implementation is over 50,000 lines of C code, while the EFS implementation is approximately 12,000 lines.

The volume manager used by XFS, known as XLV, provides a layer of abstraction between XFS and its underlying disk devices. XLV provides all of the disk striping, concatenation, and mirroring used by XFS. XFS itself knows nothing of the layout of the devices upon which it is stored. This separation of disk management from the file system simplifies the file system implementation, its application interfaces, and the management of the file system.

5. Storage Scalability

XFS goes to great lengths to efficiently support large files, large file systems, large numbers of files, and large directories. This section describes the mechanisms used to achieve such scalability in size.

5.1. Allocation Groups

XFS supports full 64 bit file systems. All of the global counters in the system are 64 bits in length. Block addresses and inode numbers are also 64 bits in length. To avoid requiring all structures in XFS to scale to the 64 bit size of the file system, the file system is partitioned into regions called allocation groups (AGs). These are somewhat similar to the cylinder groups in FFS, but AGs are used for scalability and parallelism rather than disk locality.

Allocation groups keep the size of the XFS data structures in a range where they can operate efficiently without breaking the file system into an unmanageable number of pieces. Allocation groups are typically 0.5 to 4 gigabytes in size. Each AG has its own separate data structures for managing the free space and inodes within its boundaries. Partitioning the file system into AGs limits the size of the individual structures used for tracking free space and inodes. The partitioning also allows the per-AG data structures to use AG relative block and inode pointers. Doing so reduces the size of those pointers from 64 to 32 bits. Like the limitations on the size of the region managed by the AG, this helps to keep the per-AG data structures to an optimal size.

Allocation groups are only occasionally used for disk locality. They are generally far too large to be of much use in this respect. Instead, we establish locality around individual files and directories. Like FFS, each time a new directory is created, we place it in a different AG from its parent. Once we've allocated a directory, we try to cluster the inodes in that directory and the blocks for those inodes around the directory itself. This works well for keeping directories of small files clustered together on disk. For large files, we try to allocate extents initially near the inode and afterwards near the existing block in the file which is closest to the offset in the file for which we are allocating space. That implies blocks will be allocated near the last block in the file for sequential writers and near blocks in the middle of the file for processes writing into holes. Files and directories are not limited to allocating space within a single allocation group, however. While the structures maintained within an AG use AG relative pointers, files and directories are file system global structures that can reference inodes and blocks anywhere in the file system.

The other purpose of allocation groups is to allow for parallelism in the management of free space and inode allocation. Previous file systems, like SGI's EFS, have single threaded block allocation and freeing mechanisms. On a large file system with large numbers of processes running, this can be a significant bottleneck. By making the structures in each AG independent of those in the other AGs, XFS enables free space and inode management operations to proceed in parallel throughout the file system. Thus, processes running concurrently can allocate space in the file system concurrently without interfering with each other.

5.2. Managing Free Space

Space management is key to good file system performance and scalability. Efficiently allocating and freeing space and keeping the file system from becoming fragmented are essential to good file system performance. XFS has replaced the block oriented bitmaps of other file systems with an extent oriented structure consisting of a pair of B+ trees for each allocation group. The entries in the B+ trees are descriptors of the free extents in the AG. Each descriptor consists of an AG relative starting block and a length. One of the B+ trees is indexed by the starting block of the free extents, and the other is indexed by the length of the free extents. This double indexing allows for very flexible and efficient searching for free extents based on the type of allocation being performed.

Searching an extent based tree is more efficient than a linear bitmap scan, especially for large, contiguous allocations. In searching a tree describing only the free extents, no time is wasted scanning bits for allocated blocks or determining the length of a given extent. According to our simulations, the extent based trees are just as efficient and more flexible than hierarchical bitmap schemes such as binary buddy bitmaps. Unfortunately, the results of those simulations have been lost, so we will have to settle here for an analytical explanation. Unlike binary buddy schemes, there are no restrictions on the alignment or size of the extents which can be allocated. This is why we consider the B+ trees more flexible. Finding an extent of a given size with the B+ tree indexed by free extent size, and finding an extent near a given block with the B+ tree indexed by extent starting block are both O(log N) operations. This is why we feel that the B+ trees are just as efficient as binary buddy schemes. The implementation of the allocation B+ trees is certainly more complex than normal or binary buddy bitmap schemes, but we believe that the combination of flexibility and performance we get from the B+ trees is worth the complexity.

5.3. Supporting Large Files

XFS provides a 64 bit, sparse address space for each file. The support for sparse files allows files to have holes in them for which no disk space is allocated. The support for 64 bit files means that there are potentially a very large number of blocks to be indexed for every file. In order to keep the number of entries in the file allocation map small, XFS uses an extent map rather than a block map for each file. Entries in the extent map are ranges of contiguous blocks allocated to the file. Each entry consists of the block offset of the entry in the file, the length of the extent in blocks, and the starting block of the extent in the file system. In addition to saving space over a block map by compressing the allocation map entries for up to two million contiguous blocks into a single extent map entry, using extent descriptors makes the management of contiguous space within a file efficient.

Even with the space compression provided by an extent map, sparse files may still require large numbers of entries in the file allocation map. When the number of extents allocated to a file overflows the number that can fit immediately within an XFS inode, we use a B+ tree rooted within the inode to manage the extent descriptors. The B+ tree is indexed by the block offset field of the extent descriptors, and the data stored within the B+ tree are the extent descriptors. The B+ tree structure allows us to keep track of millions of extent descriptors, and, unlike an FFS style solution, it allows us to do so without forcing all extents to be of the same size. By storing the offset and length of each entry in the extent map in the entry, we gain the benefit of entries in the map which can point to variable length extents in exchange for a more complicated map implementation and less fan out at each level of the mapping tree (since our individual entries are larger we fit fewer of them in each indirect block).

5.4. Supporting Large Numbers of Files

In addition to supporting very large files, XFS supports very large numbers of files. The number of files in a file system is limited only by the amount of space in the file system to hold them. Rather than statically pre-allocating the inodes for all of these files, XFS dynamically allocates inodes as needed. This relieves the system administrator of having to guess the number of files that will be created in a given file system and of having to recreate the file system when that guess is wrong.

With dynamically allocated inodes, it is necessary to use some data structure for keeping track of where the inodes are located. In XFS, each allocation group manages the inodes allocated within its confines. Each AG uses a B+ tree to index the locations of the inodes within it. Inodes are allocated in chunks of sixty-four inodes. The inode allocation B+ tree in each AG keeps track of the locations of these chunks and whether each inode within a chunk is in use. The inodes themselves are not actually contained in the B+ tree. The B+ tree records only indicate where each chunk of inodes is located within the AG.

The inode allocation B+ trees, containing only the offset of each inode chunk along with a bit for each inode in the chunk, can each manage millions of inodes. This flexibility comes at the cost of additional complexity in the implementation of the file system. Deciding where to allocate new chunks of inodes and keeping track of them requires complexity that does not exist in other file systems. File system backup programs that need to traverse the inodes of the file system are also more complex, because they need to be able to traverse the inode B+ trees in order to find all of the inodes. Finally, having a sparse inode numbering space forced us to use 64 bit inode numbers, and this introduces a whole series of system interface issues for returning file identifiers to programs.

5.5. Supporting Large Directories

The millions of files in an XFS file system need to be represented in the file system name space. XFS implements the traditional Unix hierarchical name space. Unlike existing Unix file systems, XFS can efficiently support large numbers of files in a single directory. XFS uses an on-disk B+ tree structure for its directories.

The directory B+ tree is a bit different from the other B+ trees in XFS. The difference is that keys for the entries in the directory B+ tree, the names of the files in the directory, vary in length from 1 to 255 bytes. To hide this fact from the B+ tree index management code, the directory entry names are hashed to four byte values which are used as the keys in the B+ tree for the entries. The directory entries are kept in the leaves of the B+ tree. Each stores the full name of the entry along with the inode number for that entry. Since the hash function used for the directories is not perfect, the directory code must manage entries in the directory with duplicate keys. This is done by keeping entries with duplicate hash values next to each other in the tree. The use of fixed size keys in the interior nodes of the directory B+ tree simplifies the code for managing the B+ tree, but by making the keys non-unique via hashing we add significant complexity. We feel that the increased performance of the directory B+ trees that results from having fixed size keys, described below, is worth the increased complexity of the implementation.

The hashing of potentially large, variable length key values to small, constant size keys increases the breadth of the directory B+ trees. This reduces the height of the tree. The breadth of the B+ tree is increased by using small, constant sized keys in the interior nodes. This is because the interior nodes are fixed in size to a single file system block, and compressing the keys allows more of them to fit into each interior node of the tree. This allows each interior node to have more children, thus increasing the breadth of the tree. In reducing the height of the tree, we reduce the number of levels that must be examined in searching for a given entry in the directory. The B+ tree structure makes lookup, create, and remove operations in directories with millions of entries practical. However, listing the contents of a directory with a million entries is still impractical due to the size of the resulting output.

5.6. Supporting Fast Crash Recovery

File systems of the size and complexity of XFS cannot be practically recovered by a process which examines the file system metadata to reconstruct the file system. In a large file system, examining the large amount of metadata will take too long. In a complex file system, piecing the on-disk data structures back together will take even longer. An example is the recovery of the XFS inode table. Since our inodes are not located in a fixed location, finding all of the inodes in the worst case where the inode B+ trees have been trashed can require scanning the entire disk for inodes. To avoid these problems, XFS uses a write ahead logging scheme that enables atomic updates of the file system. This scheme is very similar to the one described very thoroughly in [Hisgen93].

XFS logs all structural updates to the file system metadata. This includes inodes, directory blocks, free extent tree blocks, inode allocation tree blocks, file extent map blocks, AG header blocks, and the superblock. XFS does not log user data. For example, creating a file requires logging the directory block containing the new entry, the newly allocated inode, the inode allocation tree block describing the allocated inode, the allocation group header block containing the count of free inodes, and the superblock to record the change in its count of free inodes. The entry in the log for each of these items consists of header information describing which block or inode this is and a copy of the new image of the item as it should exist on disk.

Logging new copies of the modified items makes recovering the XFS log independent of both the size and complexity of the file system. Recovering the data structures from the log requires nothing but replaying the block and inode images in the log out to their real locations in the file system. The log recovery does not know that it is recovering a B+ tree. It only knows that it is restoring the latest images of some file system blocks.

Unfortunately, using a transaction log does not entirely obsolete the use of file system scavenger programs. Hardware and software errors which corrupt random blocks in the file system are not generally recoverable with the transaction log, yet these errors can make the contents of the file system inaccessible. We did not provide such a repair program in the initial release of XFS, naively thinking that it would not be necessary, but our customers have convinced us that we were wrong. Without one, the only way to bring a corrupted file system back on line is to re-create it with mkfs and restore it from backups. We will be providing a scavenger program for all versions of XFS in the near future.

6. Performance Scalability

In addition to managing large amounts of disk space, XFS is designed for high performance file and file system access. XFS is designed to run well over large, striped disk arrays where the aggregate bandwidth of the underlying drives ranges in the tens to hundreds of megabytes per second.

The keys to performance in these arrays are I/O request size and I/O request parallelism. Modern disk drives have much higher bandwidth when requests are made in large chunks. With a striped disk array, this need for large requests is increased as individual requests are broken up into smaller requests to the individual drives. Since there are practical limits to individual request sizes, it is important to issue many requests in parallel in order to keep all of the drives in a striped array busy. The aggregate bandwidth of a disk array can only be achieved if all of the drives in the array are constantly busy.

In this section, we describe how XFS makes that full aggregate bandwidth available to applications. We start with how XFS works to allocate large contiguous files. Next we describe how XFS performs I/O to those files. Finally, we describe how XFS manages its metadata for high performance.

6.1. Allocating Files Contiguously

The first step in allowing large I/O requests to a file is to allocate the file as contiguously as possible. This is because the size of a request to the underlying drives is limited by the range of contiguous blocks in the file being read or written. The space manager in XFS goes to great lengths to ensure that files are allocated contiguously.

Delaying Allocation

One of the key features of XFS in allocating files contiguously is delayed file extent allocation. Delayed allocation applies lazy evaluation techniques to file allocation. Rather than allocating specific blocks to a file as it is written in the buffer cache, XFS simply reserves blocks in the file system for the data buffered in memory. A virtual extent is built up in memory for the reserved blocks. Only when the buffered data is flushed to disk are real blocks allocated for the virtual extent. Delaying the decision of which and how many blocks to allocate to a file as it is written provides the allocator with much better knowledge of the eventual size of the file when it makes its decision. When the entire file can be buffered in memory, the entire file can usually be to be allocated in a single extent if the contiguous space to hold it is available. For files that cannot be entirely buffered in memory, delayed allocation allows the files to be allocated in much larger extents than would otherwise be possible.

Delayed allocation fits well in modern file system design in that its effectiveness increases with the size of the memory of the system. As more data is buffered in memory, the allocator is provided with better and better information for making its decisions. Also, with delayed allocation, short lived files which can be buffered in memory are often never allocated any real disk blocks. The files are removed and purged from the file cache before they are pushed to disk. Such short lived files appear to be relatively common in Unix systems [Ousterhout85, Baker91], and delayed allocation reduces both the number of metadata updates caused by such files and the impact of such files on file system fragmentation.

Another benefit of delayed allocation is that files which are written randomly but have no holes can often be allocated contiguously. If all of the dirty data can be buffered in memory, the space for the randomly written data can be allocated contiguously when the dirty data is flushed out to disk. This is especially important for applications writing data with mapped files where random access is the norm rather than the exception.

Supporting Large Extents

To make the management of large amounts of contiguous space in a file efficient, XFS uses very large extent descriptors in the file extent map. Each descriptor can describe up to two million file system blocks, because we use 21 bits in the extent descriptor to store the length of the extent. Describing large numbers of blocks with a single extent descriptor eliminates the

CPU overhead of scanning entries in the extent map to determine whether blocks in the file are contiguous. We can simply read the length of the extent rather than looking at each entry to see if it is contiguous with the previous entry.

The extent descriptors used by XFS are 16 bytes in length. This is actually their compressed size, as the in memory extent descriptor needs 20 bytes (8 for file offset, 8 for the block number, and 4 for the extent length). Having such large extent descriptors forces us to have a smaller number of direct extent pointers in the inode than we would with smaller extent descriptors like those used by EFS (8 bytes total). We feel that this is a reasonable trade-off for XFS because of our focus on contiguous file allocation and the good performance of the indirect extent maps even when we do overflow the direct extents.

Supporting Variable Block Sizes

In addition to the above features for keeping disk space contiguous, XFS allows the file system block size to range from 512 bytes to 64 kilobytes on a per file system basis. The file system block size is the minimum unit of allocation and I/O request size. Thus, setting the block size sets the minimum unit of fragmentation in the file system. Of course, this must be balanced against the large amount of internal fragmentation that is caused by using very large block sizes. File systems with large numbers of small files, for example news servers, typically use smaller block sizes in order to avoid wasting space via internal fragmentation. File systems with large files tend to make the opposite choice and use large block sizes in order to reduce external fragmentation of the file system and their files' extents.

Avoiding File System Fragmentation

The work by Seltzer and Smith [Seltzer95] shows that long term file system fragmentation can degrade the performance of FFS file systems by between 5% and 15%. This fragmentation is the result of creating and removing many files over time. Even if all of the files are allocated contiguously, eventually, the remaining files are scattered about the disk. This fragments the file system's free space. Given the propensity of of XFS for doing large I/O to contiguously allocated files, we could expect the degradation of XFS from its optimum performance to be even worse.

While XFS cannot completely avoid this problem, there are a few reasons why its impact is not as severe as it could be with XFS file systems. The first is the combination of delayed allocation and the allocation B+ trees. In using the two together XFS makes requests for larger allocations to the allocator and is able to efficiently determine one of the best fitting extents in the file system for that allocation. This helps in delaying the onset of the fragmentation problem and reducing its performance impact once it occurs. A second reason is that XFS file systems are typically larger than EFS and FFS file systems. In a large file system, there is typically a larger amount of free space for the allocator to work with. In such a file system it takes much longer for the file system to become fragmented. Another reason is that file systems tend to be used to store either a small number of large files or a large number of small files. In a file system with a smaller number of large files, fragmentation will not be a problem, because allocating and deleting large files still leaves large regions of contiguous free space in the file system. In a file system containing mostly small files, fragmentation is not a big problem, because small files have no need for large regions of contiguous space. However, in the long term we still expect fragmentation to degrade the performance of XFS file systems, so we intend to add an on-line file system defragmentation utility to optimize the file system in the future.

6.2. Performing File I/O

Given a contiguously allocated file, it is the job of the XFS I/O manager to read and write the file in large enough requests to drive the underlying disk drives at full speed. XFS uses a combination of clustering, read ahead, write behind, and request parallelism in order to exploit its underlying disk array. For high performance I/O, XFS allows applications to use direct I/O to move data directly between application memory and the disk array using DMA. Each of these is described in detail below.

Handling Read Requests

To obtain good sequential read performance, XFS uses large read buffers and multiple read ahead buffers. By large read buffers, we mean that for sequential reads we use a large minimum I/O buffer size (typically 64 kilobytes). Of course, for files smaller than the minimum buffer size, we reduce the size of the buffers to match the files. Using a large minimum I/O size ensures that even when applications issue reads in small units the file system feeds the disk array requests that are large enough for good disk I/O performance. For larger application reads, XFS increases the read buffer size to match the application's request. This is very similar to the read clustering scheme in SunOS [McVoy90], but it is more aggressive in using memory to improve I/O performance.

While large read buffers satisfy the need for large request sizes, XFS uses multiple read ahead buffers to increase the parallelism in accessing the underlying disk array. Traditional Unix systems have used only a single read ahead buffer at a time [McVoy90]. For sequential reads, XFS keeps outstanding two to three requests of the same size as the primary I/O buffer. The number varies because we try to keep three read ahead requests outstanding, but we wait until the process catches up a bit with the read ahead before issuing more. The multiple read ahead requests keep the drives in the array busy while the application processes the data being read. The larger number of read ahead buffers allows us to keep a larger number of underlying drives busy at once. Not issuing read ahead blindly, but instead waiting until the application catches up a bit, helps to keep sequential readers from flooding the drive with read ahead requests when the application is not keeping up with the I/O anyway.

Handling Write Requests

To get good write performance, XFS uses aggressive write clustering [McVoy90]. Dirty file data is buffered in memory in chunks of 64 kilobytes, and when a chunk is chosen to be flushed from memory it is clustered with other contiguous chunks to form a larger I/O request. These I/O clusters are written to disk asynchronously, so as data is written into the file cache many such clusters will be sent to the underlying disk array concurrently. This keeps the underlying disk array busy with a stream of large write requests.

The write behind used by XFS is tightly integrated with the delayed allocation mechanism described earlier. The more dirty data we can buffer in memory for a newly written file, the better the allocation for that file will be. This is balanced with the need to keep memory from being flooded with dirty pages and the need to keep I/O requests streaming out to the underlying disk array. This is mostly an issue for the file cache, however, so it will not be discussed in this paper.

Using Direct I/O

With very large disk arrays, it is often the case that the underlying I/O hardware can move data faster than the system's CPUs can copy that data into or out of the buffer cache. On these systems, the CPU is the bottleneck in moving data between a file and an application. For these situations, XFS provides what we call direct I/O. Direct I/O allows a program to read or write a file without first passing the data through the system buffer cache. The data is moved directly between the user program's buffer and the disk array using DMA. This avoids the overhead of copying the data into or out of the buffer cache, and it also allows the program to better control the size of the requests made to the underlying devices. In the initial implementation of XFS, direct I/O was not kept coherent with buffered I/O, but this has been fixed in the latest version. Direct I/O is very similar to traditional Unix raw disk access, but it differs in that the disk addressing is indirected through the file extent map.

Direct I/O provides applications with access to the full bandwidth of the underlying disk array without the complexity of managing raw disk devices. Applications processing files much larger than the system's memory can avoid using the buffer cache since they get no benefit from it. Applications like databases that consider the Unix buffer cache a nuisance can avoid it entirely while still reaping the benefits of working with normal files. Applications with real-time I/O requirements can use direct I/O to gain fine grained control over the I/O they do to files.

The downsides of direct I/O are that it is more restrictive than traditional Unix file I/O and that it requires more sophistication from the application using it. It is more restrictive in that it requires the application to align its requests on block boundaries and to keep the requests a multiple of the block size in length. This often requires more complicated buffering techniques in the application that are normally handled by the Unix file cache. Direct I/O also requires more of the application in that it places the burden of making efficient I/O requests on the application. If the application writes a file using direct I/ O and makes individual 4 kilobyte requests, the application will run much slower than if it made those same requests into the file cache where they could be clustered into larger requests. While direct I/O will never entirely replace traditional Unix file I/O, it is a useful alternative for sophisticated applications that need high performance file I/O.

Using Multiple Processes

Another barrier to high performance file I/O in many Unix file systems is the single threading inode lock used for each file. This lock ensures that only one process at a time may have I/O outstanding for a single file. This lock thwarts applications trying to increase the rate at which they can read or write a file using multiple processes to access the file at once.

XFS uses a more flexible locking scheme that allows multiple processes to read and write a file at once. When using normal, buffered I/O, multiple readers can access the file concurrently, but only a single writer is allowed access to the file at a time. The single writer restriction is due to implementation rather than architectural restrictions and will eventually be removed. When using direct I/O, multiple readers and writers can all access the file simultaneously. Currently, when using direct I/O and multiple writers, we place the burden of serializing writes to the same region of the file on the application. This differs from traditional Unix file I/O where file writes are atomic with respect to other file accesses, and it is one of the main reasons why we do not yet support multiple writers using traditional Unix file I/O.

Allowing parallel access to a file can make a significant difference in the performance of access to the file. When the bottleneck in accessing the file is the speed at which the CPU can move data between the application buffer and the buffer cache, parallel access to the file allows multiple CPUs to be applied to the data movement. When using direct I/O to drive a large disk array, parallel access to the file allows requests to be pipelined to the disk array using multiple processes to issue multiple requests. This feature is especially important for systems like IRIX that implement asynchronous I/O using threads. Without parallel file access, the asynchronous requests would be serialized by the inode lock and would therefore provide almost no performance benefit.

6.3. Accessing and Updating Metadata

The other side of file system performance is that of manipulating the file system metadata. For many applications, the speed at which files and directories can be created, destroyed, and traversed is just as important as file I/O rates. XFS attacks the problem of metadata performance on three fronts. The first is to use a transaction log to make metadata updates fast. The second is to use advanced data structures to change searches and updates from linear to logarithmic in complexity. The third is to allow parallelism in the search and update of different parts of the file system. We have already discussed the XFS data structures in detail, so this section will focus on the XFS transaction log and file system parallelism.

Logging Transactions

A problem that has plagued traditional Unix file systems is their use of ordered, synchronous updates to on-disk data structures in order to make those updates recoverable by a scavenger program like fsck. The synchronous writes slow the performance of the metadata updates down to the performance of disk writes rather than the speed of today's fast CPUs [Rosenblum92].

XFS uses a write ahead transaction log to gather all the writes of an update into a single disk I/O, and it writes the transaction log asynchronously in order to decouple the metadata update rate from the speed of the disks. Other schemes such as log structured file systems [Rosenblum92], shadow paging [Hitz94], and soft updates [Ganger94] have been proposed to solve this problem, but we feel that write ahead logging provides the best trade-off among flexibility, performance, and reliability. This is because it provides us with the fast metadata updates and crash recovery we need without sacrificing our ability to efficiently support synchronous writing workloads, for example that of an NFS server [Sandberg85], and without sacrificing our desire for large, contiguous file support. However, an in depth analysis of write ahead logging or the tradeoffs among these schemes is beyond the scope of this paper.

Logging Transactions Asynchronously

Traditional write ahead logging schemes write the log synchronously to disk before declaring a transaction committed and unlocking its resources. While this provides concrete guarantees about the permanence of an update, it restricts the update rate of the file system to the rate at which it can write the log. While XFS provides a mode for making file system updates synchronous for use when the file system is exported via NFS, the normal mode of operation for XFS is to use an asynchronously written log. We still ensure that the write ahead logging protocol is followed in that modified data cannot be flushed to disk until after the data is committed to the on-disk log. Rather than keeping the modified resources locked until the transaction is committed to disk, however, we instead unlock the resources and pin them in memory until the transaction commit is written to the on-disk log. The resources can be unlocked once the transaction is committed to the in-memory log buffers, because the log itself preserves the order of the updates to the file system.

XFS gains two things by writing the log asynchronously. First, multiple updates can be batched into a single log write. This increases the efficiency of the log writes with respect to the underlying disk array [Hagmann87, Rosenblum92]. Second, the performance of metadata updates is normally made independent of the speed of the underlying drives. This independence is limited by the amount of buffering dedicated to the log, but it is far better than the synchronous updates of older file systems.

Using a Separate Log Device

Under very intense metadata update workloads, the performance of the updates can still become limited by the speed at which the log buffers can be written to disk. This occurs when updates are being written into the buffers faster than the buffers can be written into the log. For these cases, XFS allows the log to be placed on a separate device from the rest of the file system. It can be stored on a dedicated disk or non-volatile memory device. Using non-volatile memory devices for the transaction log has proven very effective in high end OLTP systems [Dimino94]. It can be especially useful with XFS on an NFS server, where updates must be synchronous, in both increasing the throughput and decreasing the latency of metadata update operations.

Exploiting Parallelism

XFS is designed to run well on large scale shared memory multiprocessors. In order to support the parallelism of such a machine, XFS has only one centralized resource: the transaction log. All other resources in the file system are made independent either across allocation groups or across individual inodes. This allows inodes and blocks to be allocated and freed in parallel throughout the file system.

The transaction log is the most contentious resource in XFS. All updates to the XFS metadata pass through the log. However, the job of the log manager is very simple. It provides buffer space into which transactions can copy their updates, it writes those updates out to disk, and it notifies the transactions when the log writes complete. The copying of data into the log is easily parallelized by making the processor performing the transaction do the copy. As long as the log can be written fast enough to keep up with the transaction load, the fact that it is centralized is not a problem. However, under workloads which modify large amount of metadata without pausing to do anything else, like a program constantly linking and unlinking a file in a directory, the metadata update rate will be limited to the speed at which we can write the log to disk.

7. Experience and Performance Results

In this section we present results demonstrating the scalability and performance of the XFS file system. These results are not meant as a rigorous investigation of the performance of XFS, but only as a demonstration of XFS's capabilities. We are continuing to measure and improve the performance of XFS as development of the file system proceeds.

7.1. I/O Throughput Test Results

Figures 2 and 3 contain the results of some I/O throughput tests run on a raw volume, XFS, and EFS. The results come from a test which measures the rate at which we can write a previously empty file (create), read it back (read), and overwrite the existing file (write). The number of drives over which the underlying volume is striped ranges from 3 to 57 in the test. The test system is an 8 CPU Challenge with 512 megabytes of memory. The test is run with three disks per SCSI channel, and each disk is capable of reading data sequentially at approximately 7 MB/sec and writing data sequentially at approximately 5.5 MB/ sec. All tests are run on newly created file systems in order to measure the optimal performance of the file systems. All tests using EFS and XFS are using direct I/O and large I/O requests, and the tests using multiple threads are using the IRIX asynchronous I/O library with the given number of threads. Measurements for multiple, asynchronous threads with EFS are not given, because the performance of EFS with multiple threads is the same or worse as with one thread due to its single threaded (per file) I/O path. The test files are approximately 30 megabytes per disk in the volume in size, and for the raw volume tests we write the same amount of data to the volume itself. The stripe unit for the volumes is 84 kilobytes for the single threaded cases and 256 kilobytes for the multi-threaded cases. We have found these stripe units to provide the best performance for each of the cases in our experimentation.

We can draw several conclusions from this data. One is that XFS is capable of reading a file at nearly the full speed of the underlying volume. We manage to stay within 5-10% of the raw volume performance in all disk configurations when using an equivalent number of asynchronous I/O threads. Another interesting result is the parity of the create and write results for XFS versus the large disparity of the results for EFS. We believe that this demonstrates the efficiency of the XFS space allocator. Finally, the benefits of parallel file access are clearly demonstrated in these results. At the high end this makes a 55 MB/sec difference in the XFS read results. For writing and creating files it makes a 125 MB/sec difference. This is entirely because the parallel cases are capable of pipelining the drives with requests to keep them constantly busy

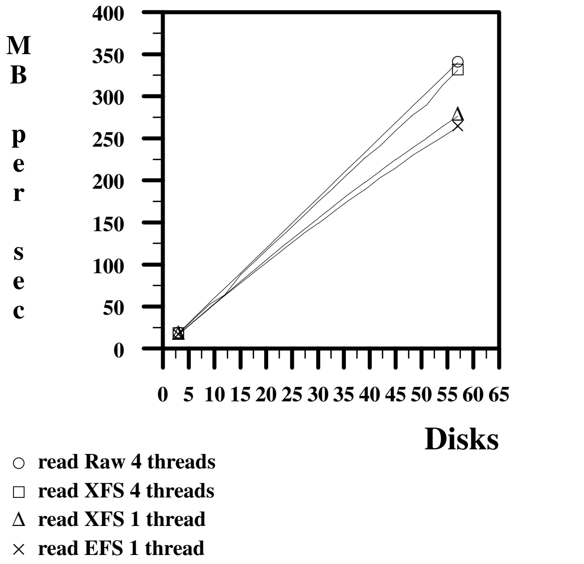

Figure 2. Read Throughput.

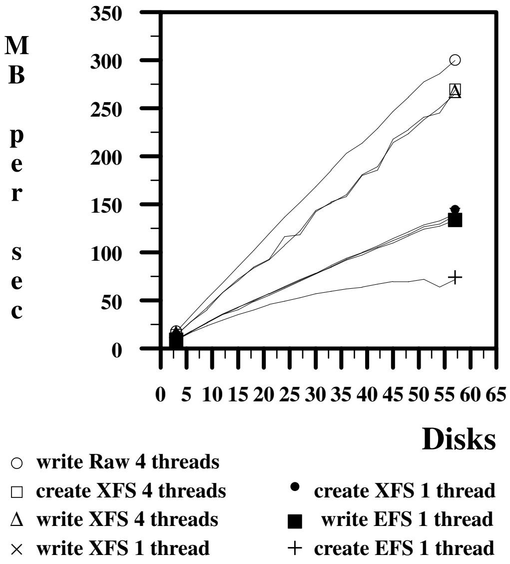

Figure 3. Write/Create Throughput.

whereas the single threaded cases are not.

7.2. Database Sort Benchmark Results

Using XFS, Silicon Graphics recently achieved record breaking performance on the Datamation sort [Anon85] and Indy MinuteSort [Nyberg94] benchmarks. The Datamation sort benchmark measures how fast the system can sort 100 megabytes of 100 byte records. The MinuteSort benchmark measures how much data the system can sort in one minute. This includes start-up, reading the data in from disk, sorting it in memory, and writing the sorted data back out to disk. On a 12 CPU 200 Mhz Challenge system with 2.25 gigabytes of memory and a striped volume of 96 disk drives, we performed the Datamation sort in 3.52 seconds and sorted 1.6 gigabytes of data in 56 seconds for the MinuteSort. The previous records of 7 seconds and 1.08 gigabytes of data were achieved on a DEC Alpha system running VMS.

Achieving this level of results requires high memory bandwidth, high file system and I/O bandwidth, scalable multiprocessing, and a sophisticated multiprocessing sort package. The key contribution of XFS to these results is the ability to create and read files at 170 MB/sec. This actually moved the bottleneck in the system from the file system to the allocation of zeroed pages for the sort processes.

7.3. LADDIS Benchmark Results

The results of the SPEC SFS (a.k.a. LADDIS) benchmark using XFS are encouraging as well. On a 12 CPU 250 Mhz Challenge XL with 1 gigabyte of memory, 4 FDDI networks, 16 scsi channels, and 121 disks, we achieved a maximum throughput of 8806 SPECnfs operations per second. While XFS plays only a part in achieving such outstanding performance, these results exceed our previous results using the EFS file system. On a slightly less powerful machine using EFS, we originally reported a result of 7023 SPECnfs operations per second. We estimate that the difference in hardware accounts for approximately 800 of the operations, leaving XFS approximately 1000 operations per second ahead of EFS. The difference is that EFS achieves 65 operations per second per disk, while XFS achieves 73 operations per disk. While this 12% increase might not seem like much, the LADDIS workload is dominated by small, synchronous write performance. This is often very difficult to improve without better disk hardware. We believe that the improvement with XFS is the result of the high performance directory structures, better file allocations, and synchronous metadata update batching of the transaction log provided by XFS.

7.4. Directory Lookups in Large Directories

Figure 4 contains the results for a test measuring the performance of random lookups in directories of various sizes for EFS and XFS. The results included are the average of several iterations of the test. The machine used for the test is a 4 CPU machine with 128 megabytes of memory. Each file system was created on a single, 2 gigabyte disk with nothing else on it. To make sure that we are measuring the performance of the file system directory structures, the test is run with the directory name lookup cache turned off. Also, the entries in the directories are all links to just a few real files. There are 20,000 links per real file. The test performs lookups using the stat(2) system call, so making most of the entries links to just a few files eliminates the size of the inode cache from the variability of the test.

Directory entries	EFS	XFS
100	5,970	8,972
500	2,833	6,600
1,000	1,596	7,089
10,000	169	6,716
30,000	43	6,522
50,000	27	6,497
70,000	-	5,661
90,000	-	5,497
150,000	-	177
250,000	-	102
350,000	-	90
450,000	-	79
1,000,000	-	66

Figure 4. Lookup Operations Per Second

It is clear from this test that lookups in medium to large directories are much more efficient using XFS. EFS uses a linear directory format similar to that used by BSD FFS. It degrades severely between 1,000 and 10,000 entries, at which point the test is entirely CPU bound scanning the cached file blocks for the entries being looked up. For XFS, the test is entirely CPU bound, but still very fast, until the size of the directory overflows the number of blocks that can be cached in memory. While there is a large amount of memory in the machine, only a limited portion of it can be used to cache directory blocks due to limitations of the IRIX metadata block cache. At the point where we overflow the cache, the interior nodes of the directory B+ tree are still cached, but most leaf nodes in the tree need to be read in from disk when they are accessed. This reduces the performance of the test to the performance of directory block sized I/O operations to the single underlying disk drive. The reason the performance continues to degrade as the directory size increases is most likely that the effectiveness of the leaf block caching continues to decrease with the increase in directory size.

8. Conclusion

The main idea behind the design of XFS is very simple: think big. This idea brings forth the needs for large file systems, large files, large numbers of files, large directories, and large I/O that are addressed in the design and implementation of XFS. We believe that by satisfying these needs, XFS will satisfy the needs of the next generation of applications and systems so that we will not be back to where we are today in just a few years.

The mechanisms in XFS for satisfying the requirements of big systems also make it a high performance general purpose file system. The pervasive use of B+ trees throughout the file system reduces many of the algorithms in the file system from linear to logarithmic. The use of asynchronous transaction logging eliminates many of the metadata update performance problems in previous file systems. Also, the use of delayed allocation improves the performance of all file allocations, especially those of small files. XFS is designed to perform well on both the desktop and the server, and it is this focus on scalability that distinguishes XFS from the rest of the file system crowd.

9. Acknowledgments

We would like to thank John Ousterhout, Bob Gray, and Ray Chen for their help in reviewing and improving this paper; Chuck Bullis, Ray Chen, Tin Le, James Leong, Jim Orosz, Tom Phelan, and Supriya Wickrematillake, the other members of the XFS/XLV team, for helping to make XFS and XLV real, commercial products; and Larry McVoy for his magic troff incantations that made this paper presentable.

10. References

[Anon85] Anonymous, "A Measure of Transaction Processing Power," Datamation, Vol. 31 No. 7, 112-118.

[Baker91] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a

Distributed File System," Proceedings of the 13th Symposium on Operating System Principles, Pacific Grove, CA, October 1991, 192-212.

[Chutani92] Chutani, S., Anderson, O., et. al., "The Episode File System," Proceedings of the 1992 Winter Usenix, San Francisco, CA, 1992, 43-60.

[Comer79] Comer, D., "The Ubiquitous B-Tree," Computing Surveys, Vol. 11, No. 2, June 1979 121-137.

[Dimino94] Dimino, L., Mediouni, R., Rengarajan, T., Rubino, M., Spiro, P., "Performance of DEC Rdb Version 6.0 on AXP Systems," Digital Technical Journal, Vol. 6, No. 1, Winter 1994 23-35.

[Ganger94] Ganger, G., Patt, Y., "Metadata Update Performance in File Systems," Proceedings of the First Usenix Symposium on Operating System Design and Implementation, Monterey, CA, November, 1994, 49-60.

[Hagmann87] Hagmann, R., "Reimplementing the Cedar File System Using Logging and Group Commit," Proceedings of the 10th Symposium on Operating System Principles, November, 1987.

[Hisgen93] Hisgen, A., Birrell, A., Jerian, C., Mann, T., Swart, G., "New-Value Logging in the Echo Replicated File System," Research Report 104, Systems Research Center, Digital Equipment Corporation, 1993.

[Hitz94] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," Proceedings of the 1994 Winter Usenix, San Francisco, CA, 1994, 235-246.

[Kleiman86] Kleiman, S., "Vnodes: an Architecture for Multiple File System types in Sun Unix," Proceedings of the 1986 Summer Usenix, Summer 1986.

[McKusick84] McKusick, M., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX," ACM Transactions on Computer Systems Vol. 2, No. 3, August 1984, 181-197.

[McVoy90] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," Proceedings of the 1991 Winter Usenix, Dallas, Texas, June 1991, 33-43.

[Nyberg94] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D., "AlphaSort: A RISC Machine Sort," Proceedings of the 1994 SIGMOD International Conference on Management of Data, Minneapolis, 1994.

[Ousterhout85] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," Proceedings of the 10th Symposium on Operating System Principles, Orcas Island, WA, December 1985, 15-24.

[Ousterhout90] Ousterhout, J. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" Proceedings of the 1990 Summer Usenix, Anaheim, CA, June, 1990, 247-256.

[Rosenblum92] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems Vol 10, No. 1, February 1992, 26-52.

[Sandberg85] Sandberg, R., et al., "Design and Implementation of the Sun Network File System," Proceedings of the 1985 Summer Usenix, June, 1985, 119-130.

[Seltzer95] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V., "File System Logging Versus Clustering: A Performance Comparison," Proceedings of the 1995 Usenix Technical Conference, January 1995, 249-264.

[SGI92] IRIX Advanced Site and Server Administration Guide, Silicon Graphics, Inc., chapter 8, 241-288

[Veritas95] Veritas Software, http://www.veritas.com

Adam Sweeney, **Doug Doucette**, **Wei Hu**, **Curtis Anderson**, **Michael Nishimoto**, and **Geoff Peck** are all members of the Server Technology group at Silicon Graphics. Adam went to Stanford, Doug to NYU and Berkeley, Wei to MIT, Curtis to Cal Poly, Michael to Berkeley and Stanford, and Geoff to Harvard and Berkeley. None of them holds a Ph.D. All together they have worked at somewhere around 27 companies, on projects including secure operating systems, distributed operating systems, fault tolerant systems, and plain old Unix systems. None of them intends to make a career out of building file systems, but they all enjoyed building one.