White Paper

Scalability and Performance in Modern Filesystems

By Philip Trautman

1. Executive Summary

In the early 1990's SGI undertook the development of a new filesystem, XFS™, to meet the needs of its customers for filesystem scalability and I/O performance. The XFS filesystem has been designed from scratch to scale to previously unheard of levels in terms of filesystem capacity, file size, number of files stored, and directory size. Because of the pervasive use of B+ trees to speed up traditionally linear algorithms, XFS is able to provide tremendous scalability while delivering I/O performance that approaches the maximum throughput of the underlying hardware. At the same time, XFS provides asynchronous metadata logging to ensure rapid crash recovery without creating a bottleneck to I/O performance.

This paper compares the XFS filesystem with three other filesystems in widespread use today:

- the UNIX®Filesystem (UFS), still widely available from UNIX vendors like Sun and HP
- the Veritas Filesystem (VxFS), a commercial filesystem frequently used on UNIX platforms
- the Microsoft® Windows NT® Filesystem (NTFS)

Because of its unique second-generation design, XFS offers superior scalability and performance in comparison to these filesystems. The following table is a summary of key features discussed in the paper:

Feature	XFS	UFS	VxFS	NTFS
Max FS Size	18 million TB	1TB	1TB	2TB
Max File Size	9 million TB	1TB	1TB	2TB
File Space Allocation	Extents	Blocks	Extents	Extents
Max. Extent Size	4GB	NA	64MB	Undoc'd
Free Space Mgmt	Free extents organized by B+ trees	Bitmap per cylinder grp	Bitmap per allocation unit	Single bitmap
Variable Block Size	512 bytes to 64KB	4KB or 8KB	Undoc'd	512 bytes to 64KB (4KB w/ compression)
Sparse File Support?	Yes	Yes	No	NT 5.0
Directory Organization	B+ Tree	Linear	Hashed	B+ tree
Inode Allocation	Dynamic	Static	Dynamic	Dynamic
Crash Recovery	Asynch. Journal	Fsck*	Synch. Journal	Synch. Journal
Maximum Performance	7GB/sec 4GB/sec (single file)	Not Available	1GB/sec	Not Available

Table 1: Feature comparison of XFS with the UNIX Filesystem (UFS), The Veritas Filesystem (VxFS), and the Windows NT Filesystem (NTFS).

While some of the size limits of XFS may appear excessive and those of the competitors more than adequate, it should be noted that some SGI' customers already have XFS filesystems that exceed the limits of the competitors.

XFS far exceeds the scalability of even its closest competitors. For customers who need to maximize performance while scaling to handle huge files and huge data sets, there is no other choice that comes close.

2. Introduction

If someone had told you ten years ago that disk drive capacity would increase to such an extent that in 1998 even the humblest desktop systems would ship with multi-gigabyte disk drives, you scarcely would have believed them. Yet today multi-gigabyte desktop systems are the norm and system administrators struggle to manage millions of files stored on servers with capacities measured in terabytes. Many installations are doubling their total disk storage capacity every year. At the same time, processing power has grown astronomically, so the average system, whether desktop or server, must not only manage and store much more data, it also must access and move that data much more rapidly to meet system and/or network I/O demands.

Perhaps not surprisingly, the filesystem technology that was adequate to manage and access the multi-megabyte systems of the 1980s doesn't scale to meet the demands created when storage capacity and processing power increase so dramatically. Algorithms that were once sufficient often either fail to scale to the level required or perform so slowly and inefficiently that they might as well have failed.

In the early 1990s—when it was already clear that the EFS filesystem was creating I/O bottlenecks for many SGI customers-SGI set out to develop a filesystem that would scale into the next millennium.

From direct customer experience, SGI knew that it needed a next generation filesystem that supported:

- Filesystems of as much as a petabyte. EFS was limited to a maximum filesystem size of 8GB. Many 32-bit filesystems are limited to 2GB.

- Large files. Many filesystems were limited to a maximum file size of 2GB. In addition, File I/O can be dramatically accelerated by allocating disk space contiguously.

- Large directories. Most filesystems use linear searches, going through a directory entry by entry, to locate a particular file. This becomes very inefficient when the number of files in a directory exceeds a few thousand.

- Large numbers of files. The only way to efficiently scale to support large numbers of files (without prior knowledge of the number of files the filesystem would ultimately support) is to dynamically allocate index space for files.

- Rapid crash recovery. Many traditional filesystems require a checking program to check filesystem consistency after a crash. On large, active filesystems this type of checking can take a prohibitively long time to complete. Solving this problem must not degrade I/O performance.

- Unparalleled performance. Performance should not degrade as the size of the filesystem, an individual file, or the total number of files stored grows. The ideal filesystem should provide performance that approaches the maximum performance of the underlying hardware.

From surveying available filesystems, it was clear there was no single existing technology that met all these requirements. In response, SGI developed the XFS filesystem, a filesystem that could manage the huge data sets of supercomputers while at the same time meeting the needs of the desktop workstation.

XFS benefits from tight integration with the IRIX(r) operating system to take full advantage of the underlying hardware. IRIX has been a full 64-bit operating system since 1994. By comparison, HP began shipping its first full 64-bit operating system in 1997 and Sun's 64-bit version of Solaris was released in 1999. A 64-bit version of Windows NT is still far in the future.

XFS also complements the ccNUMA architecture of the SGI (tm)Origin(tm) line of servers. The ccNUMA architecture provides high performance and extreme scalability through its unique interconnect technology, allowing the system to scale from two to 128 processors. This unprecedented and seamless scalability addresses exponential data growth and rapidly evolving business needs, allowing companies to meet the growing demands of changing environments.

This paper examines in detail how XFS satisfies the scalability, performance, and crash recovery requirements of even the most demanding applications, comparing XFS to other widely available filesystem technologies:

UFS: The archetypal UNIX filesystem still widely available from UNIX vendors such as Sun and HP

VxFS: The Veritas Filesystem, a commercially developed filesystem available on a number of UNIX platforms including Sun and HP

NTFS: The filesystem designed by Microsoft for Windows NT

3. Filesystem Scalability

Simply put, a filesystem is the software used to organize and manage the data stored on disk drives. The filesystem ensures the integrity of the data. Anytime data is written to disk, it should be identical when it is read back. In addition to storing the data contained in files, a filesystem also stores and manages important information about the files and about the filesystem itself. This information is commonly referred to as metadata.

File metadata includes date and time stamps, ownership, access permissions, other security information such as access control lists (ACLs) if they exist, the file's size, and the storage location or locations on disk.

In addition, the filesystem must also keep track of free versus allocated space and provide mechanisms for creating and deleting files and allocating and freeing disk space as files grow, shrink, or are deleted.

For this discussion, filesystem scalability is defined as the ability to support very large filesystems, large files, large directories, and large numbers of files while still providing I/O performance. (A more detailed discussion of performance is contained in a later section.)
The scalability of a filesystem depends in part on how it stores information about files. For instance, if file size is stored as a 32-bit number, then no file in the filesystem can usefully exceed 232 bytes (4 GB).

Scalability also depends on the methods used to organize and access data within the filesystem. As an example, if directories are stored as a simple list of file names in no particular order, then to look up a particular file each entry must be searched one by one until the desired entry is found. This works fine for small directories but not so well for large ones.

XFS is a filesystem that was designed to scale to meet the most demanding storage capacity and I/O needs. XFS achieves this through extensive use of B+ trees in place of traditional linear filesystem structures and by ensuring that all data structures are appropriately sized. B+ trees provide an efficient indexing method that is used to rapidly locate free space, to index directory entries, to manage file extents, and to keep track of the locations of file index information within the filesystem. The B+ tree structure takes the form of an inverted tree, in some ways analogous in form to a directory hierarchy. The tree can be efficiently searched by descending from the root, making simple comparisons between the desired value and the values stored in the tree. The B+ tree is particularly well suited to paged files because it supports random or sequential access to data stored within the tree.

By comparison, the UFS filesystem was designed at UC Berkeley in the early 1980s when the scalability requirements were much different than they are today. Filesystems at that time were designed as much to conserve the limited available disk space as to maximize performance. This filesystem is also frequently referred to as FFS or the "fast" filesystem. While numerous enhancements have been made to UFS over the years to overcome limitations that appeared as technology marched on, the fundamental design still limits its scalability in many areas.

The Veritas filesystem was designed in the mid-1980s and draws heavily from the UFS design but with substantial changes to improve scalability over UFS. However, as you will see, VxFS lacks many of the key scalability enhancements of XFS, and in many respects represents an intermediate point between first generation filesystems like UFS and a true second generation filesystem like XFS.

The design of NTFS began in the late 1980s as the Microsoft Windows NT operating system was first being developed, and design work is still ongoing. NTFS was designed primarily for the desktop PC with some thought for the PC server. At the time development began, the typical desktop PC would have had disk capacities in the tens to hundreds of megabytes and a server would have been no more than 1GB. While some thought was given to scaling beyond 32-bit limits, no mechanisms are apparent to manage data effectively in large disk volumes. NTFS seems to have ignored some of the important scalability lessons that could have been learned from first-generation UNIX filesystems.

The terminology that Microsoft uses to describe NTFS (and filesystems in general) is almost completely different, and in many cases the same term is used with a different meaning. This terminology will be explained as the discussion progresses. (A glossary of terms is included at the end of this document.)

This section examines the scalability of these four filesystems based on the way they organize data and the algorithms they use.

3.1 Support for Large Filesystems

Not only have individual disk drives gotten bigger, but most operating systems have the ability to create even bigger volumes by joining partitions from multiple drives together. RAID devices are also commonly available, each RAID array appearing as a single large device. Data storage needs at many sites are doubling every year, making larger filesystems a necessity.
The minimum requirement to allow a filesystem to scale beyond 4GB in size is support for sizes beyond 32-bits. In addition, to be effective, a filesystem must also provide the appropriate algorithms and internal organization to meet the demands created by the much greater amount of I/O that is likely in a large filesystem.

UFS

The UFS filesystem was designed at a time when 32-bit computing was the norm. As such, it originally supported filesystems of up to 231 or 2GB. (The number is 231 bytes rather than 232 bytes because the size was stored as a signed integer in which one bit is needed for the sign.) Because of the practical limitations this imposes, most current implementations have been extended to support larger filesystems. For instance, Sun extended UFS in Solaris 2.6 to support filesystems of up to 1TB in size.

UFS divides its filesystems into cylinder groups. Each cylinder group contains bookkeeping information including inodes (file index information) and bitmaps for managing free space. The major purpose of cylinder groups is to reduce disk head movement by keeping inodes closer to their associated disk blocks.

XFS

XFS includes a fully integrated volume manager, XLV, which is capable of concatenating, plexing (mirroring), or striping across up to 128 volume elements. Each volume element can consist of up to 100 disk partitions or RAID arrays so that single volumes can scale to hundreds of terabytes of capacity. The EFS filesystem was only capable of supporting filesystems up to 8 GB in size, which was inadequate for many purposes. (EFS is the first-generation SGI filesystem that took the place of UFS in early versions of the IRIX operating system.)

XFS is a 64-bit filesystem. All of the global counters in the system are 64 bits in length, as are the addresses used for each disk block and the unique number assigned to each file (the inode number). A single filesystem can theoretically be as large as 18 million terabytes.

To avoid requiring all data structures in the filesystem to be 64 bits in length, the filesystem is partitioned into regions called allocation groups (AGs). Like UFS cylinder groups, each AG manages its own free space and inodes. However, the primary purpose of allocation groups is to provide scalability and parallelism within the filesystem. This partitioning also limits the size of the structures needed to track this information and allows the internal pointers to be 32 bits. AGs typically range in size from 0.5 to 4GB. Files and directories are not limited to allocating space within a single AG.

The free space and inodes within each AG are managed independently and in parallel so multiple processes can allocate free space throughout the filesystem simultaneously. This is in sharp contrast to other filesystems such as UFS, which are single-threaded, requiring space and inode allocation to occur one process at a time, resulting in a big bottleneck in large active filesystems.

VxFS
The maximum filesystem size supported by VxFS depends on the operating system on which it is running. For instance, in HP-UX 10.x the maximum filesystem size is 128 GB. This increases to 1 TB in HP-UX 11. Internally, VxFS volumes are divided into allocation units of about 32MB in size. Like UFS, these allocation units are intended to keep inodes and associated file data in proximity but do not provide greater parallelism within the filesystem as allocation groups do for XFS.

NTFS
NTFS provides a full 64-bit filesystem, theoretically capable of scaling to large sizes. However, other limitations result in a practical limit of 2TB for a single filesystem. NTFS provides no internal organization analogous to XFS allocation groups or even to UFS cylinder groups. In practice, this will severely limit NTFS's ability to efficiently use the underlying disk volume when it is very large.

In summary, XFS, because of its full 64-bit implementation and large allocation groups that operate independently from one another, is most able to take advantage of the throughput of large disk volumes.

3.2 Support for Large Files

Most traditional filesystems support files no larger than 231 or 232 bytes (2GB or 4GB) in length. In other words, they use no more than 32 bits to store the length of the file. At a minimum then, a filesystem must use more bits to represent the length of the file in order to support larger files. In addition, a filesystem must be able to allocate and track the disk space used by the file, even when the file is very large, and do so efficiently.
File I/O performance can often be dramatically increased if the blocks of a file are allocated contiguously. So the method by which disk space is allocated and tracked is critical. There are two general disk allocation methods used by the filesystems in this discussion.

- Block Allocation: Blocks are allocated one at a time and a pointer is kept to each block in the file.

- Extent Allocation: Large numbers of contiguous blocks-called extents-are allocated to the file and tracked as a unit. A pointer need only be maintained to the beginning of the extent. Because a single pointer is used to track a large number of blocks, the bookkeeping for large files is much more efficient.

The method by which free space within the filesystem is tracked and managed becomes important because it directly impacts the ability to quickly locate and allocate free blocks or extents of appropriate size. Most filesystems use linear bitmap structures to map free versus allocated space. Each bit in the bitmap represents a block in the filesystem. However, it is extremely inefficient to search through a bitmap to find large chunks of free space, particularly when the filesystem is large.

It is also advantageous to control the block size used by the filesystem. This is the minimum-sized unit that can be allocated within the filesystem. It is important to distinguish here between the physical block size used by disk hardware (typically fixed at 512 bytes) and the block size used by the filesystem, often called the logical block size.

If a system administrator knows the filesystem is going to be used to store large files it would make sense to use the largest possible logical block size, thereby reducing external fragmentation. (External Fragmentation is the term used to describe the condition when files are spread in small pieces throughout the filesystem. In the worst case in some implementations, disk space may be unallocated but unusable.)

Conversely, if the filesystem is used for small files (such as news) a small block size makes sense, and helps to reduce internal fragmentation. (Internal Fragmentation is the term used to describe disk space that is allocated to a file but unused because the file is smaller than the allocated space).

UFS
UFS, having been designed in an era when 64-bit computing was nonexistent, was designed with file size limited to 231 bytes or 2GB. Disk space is allocated by block with the inode storing 12 direct block pointers. For files larger than 12 blocks, three indirect block pointers are provided. The first indirect block pointer designates a disk block that contains additional pointers to file blocks. The second pointer, called the double indirect block pointer, designates a block that contains pointers to blocks that contains pointers to file blocks. The third pointer, the triple indirect block pointer, simply extends this same concept one more level. In practice, it has rarely been used. You can see that, for very large files, this method becomes extremely inefficient. To support a full 64-bit address space yet another level of indirection would be required.

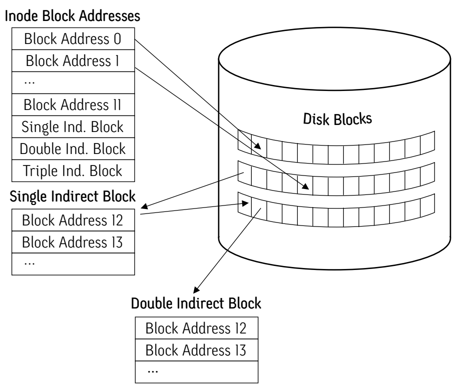

Figure 1. Illustration of UFS direct and indirect block pointers

UFS uses traditional bitmaps to keep track of free space, and in general the block size used is fixed by the implementation. (For example, Sun's UFS implementation allows block sizes of 4 or 8KB only.) Little or no attempt is made to keep file blocks contiguous, so reading and writing large files can have a tremendous amount of overhead.

In summary, traditional UFS is limited to files no larger than 2GB, block allocation is used rather than extent allocation, and its algorithms are inefficient when managing very large files and large amounts of free space. Because these restrictions would be almost unacceptable today, most vendors that use UFS have made some changes. For instance, in the early 1990s Sun implemented a new algorithm called clustering that allows for more extent-like behavior by gathering up to 56KB of data in memory and then trying to allocate disk space contiguously. Sun also extended the maximum file size to 1TB in Solaris 2.6. However, despite these enhancements, the more fundamental inefficiencies in block allocation and free space management remain.

XFS

By comparison, given its scalability goals, it is not surprising that large file support is a major area of innovation within XFS. XFS is a full 64-bit filesystem. All data structures are designed to support files as large as 263 bytes (9 exabytes).

XFS uses variable-length extent allocation to allow files to allocate the largest possible chunks of contiguous space. Each extent is described by its block offset within the file, its length in blocks, and its starting block in the filesystem. A single extent can consist of up to two million contiguous blocks or a maximum of 4GB of disk space. (The limit that applies depends on the logical block size.)

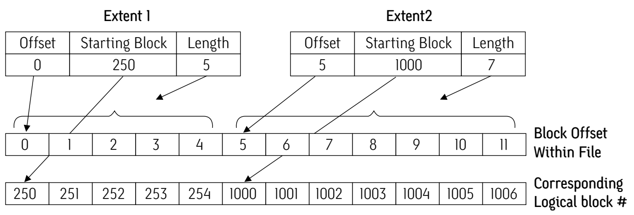

Figure 2. Illustration of XFS extent descriptors.

Despite the ability to allocate very large extents, files in XFS may still consist of a large number of extents of varying sizes, depending on the way a file grows over time. XFS inodes contain 9 or more entries that point to extents. (Inode size is tunable in XFS. The default size of 256 bytes allows for 9 direct entries.) If a given file contains more extents than that, the file's extents are mapped by a B+ tree to enhance the speed with which any given block in the file can be located.

As mentioned previously, free space in XFS is managed on a per allocation group basis. Two B+ trees are maintained for each AG that describe its free extents. One tree is indexed by the starting block of the free extents and the other by the length of the free extents. Depending on the type of allocation, the filesystem can quickly locate either the closest extent to a given location or rapidly find an extent of a given size. XFS also allows the logical block size to range from 512 bytes to 64KB on a per-filesystem basis.

VxFS

The maximum file size supported by VxFS depends on the version. For HP-UX 11 it is 1TB. Like XFS, VxFS uses extent based allocation. The maximum extent size is 64MB for VxFS (versus 4GB for XFS). Each VxFS inode stores 10 direct pointers. Each extent is described by its starting block within the disk volume and length. When a file stored by VxFS requires more than 10 extents, indirect blocks are used in a manner analogous to UFS, except that the indirect blocks store extent descriptors rather than pointers to individual blocks.

Unlike XFS, VxFS does not index the extent maps for large files, and so locating individual blocks in a file requires a linear search through the extents. Free space is managed through linear bitmaps like UFS and the logical block size for the filesystem is not tunable. In summary, VxFS has some enhancements that make it more scalable than UFS, but lacks corollaries to the key enhancements that make XFS so effective at managing extremely large files.

NTFS

NTFS support full 64-bit file sizes. The disk space allocation mechanism in NTFS is extent-based, but Microsoft typically refers to extents as runs in its literature. Despite the use of extent-based allocation, NTFS doesn't appear to have the same concern with contiguous allocation that XFS or VxFS has. Available information suggests that fragmentation rapidly becomes a problem. Each file in NTFS is mapped by an entry in the master file table or MFT.

The MFT entry for a file contains a simple linear list of the extents or runs in the file. Since each MFT entry is only 1024 bytes long and that space is also used for other file attributes, a large file with many extents will exceed the space in a single MFT entry. In that case, a separate MFT entry is created to continue the list. It is also important to note that the MFT itself is a linear data structure, so the operation to locate the MFT entry for any given file requires a linear search.

NTFS manages free space using bitmaps like UFS and VxFS. However, unlike those filesystems, NTFS uses a single bitmap to map the entire volume. Again, this would be grossly inefficient on a large volume, especially as it becomes full, and provides no opportunities for parallelism.

The NTFS literature refers to logical blocks as clusters. Allowable cluster sizes range from 512 bytes to 64KB. (The maximum size drops to 4KB for a filesystem where compression will be allowed.) The default used by NTFS is adjusted based on the size of the underlying volume.

3.2.1 Sparse Files

Some applications create files with large amounts of blank space within them. A significant amount of disk space can be saved if the filesystem can avoid allocating disk space until this blank space is actually filled.

UFS, designed in a time when each disk block was precious, supports sparse files. Mapping sparse files is relatively straightforward for filesystems that use block allocation, although large sparse files still require a large number of block pointers and thus still experience the inefficiencies of multiple layers of indirection.

XFS provides a 64-bit, sparse address space for each file. Support for sparse files allows files to have holes in them for which no disk space is allocated. Support for 64-bit files means that there are potentially a very large number of blocks to be indexed for every file. The methods that XFS uses to allocate and manage extents make this efficient, since XFS stores the block offset within the file as part of the extent descriptor. Therefore, extents can be discontinuous.

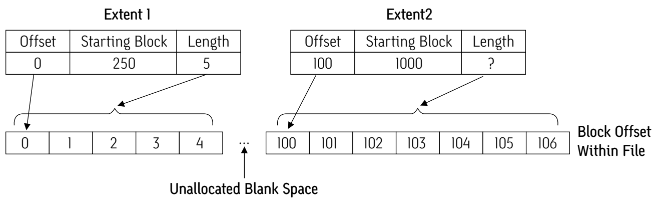

Figure 3. XFS extent descriptors mapping a sparse file.

VxFS does not support sparse files. Since VxFS extent descriptors only include the starting block within the filesystem and the extent length (no offset within the file), there is no easy way to skip holes.
Sparse file support is planned for NTFS in Windows NT 5.0 (Windows 2000), which will soon be released.

3.3 Large Directories

Large software builds and applications such as sendmail, netnews, and digital media creation often result in single directories containing thousands of files. Looking up a file name in such a directory can take a significant amount of time using linear search techniques. UFS directories are unordered, consisting only of pairings of file names and associated inode numbers. Thus, to locate any particular entry, the directory is read from the beginning until the desired entry is found.

XFS uses an on-disk B+ tree structure for its directories. File names in the directory are first converted to 4-byte hash values that are used to index the B+ tree. 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 millions of entries remains impractical due to the size of the resulting output.

In contrast, VxFS uses a hashing mechanism to organize the entries in its directories. This involves performing a mathematical operation on the file name to generate a key. This key is then used to order the file within the directory. However, the keys generated by any hashing mechanism are unlikely to be unique, particularly in a large directory. So for a particular key value, a large number of file names might still need to be searched, and this search process is linear. In practice, the file naming conventions of some programs that generate large numbers of files can render hashing alone ineffective.

NTFS uses B+ trees to index its directories in a manner similar to XFS. A number of popular PC applications create large numbers of files in single directories, so this was a filesystem problem that was recognized by NTFS designers.

3.4 Large Numbers of Files

In order to support a large number of files efficiently, a filesystem should dynamically allocate the inodes that are used to keep track of files. In traditional filesystems like UFS, the number of inodes is fixed at the time the filesystem is created. Choosing a large number up front consumes a significant amount of disk space that may never be put to use. On the other hand, if the number created is too low, the filesystem must be backed up, re-created with a larger number of inodes, and then restored.

With a very large number of files, it is also reasonable to expect that file accesses, creations, and deletions will in many cases be more numerous. Therefore, to handle a large number of files efficiently the filesystem should also allow multiple file operations to proceed in parallel.

In XFS, the number of files in a filesystem is limited only by the amount of space available to hold them. XFS dynamically allocates inodes as needed. Inodes are managed within the confines of each allocation group. Inodes are allocated 64 at a time and a B+ tree in each allocation group keeps track of the location of each group of inodes and records which inodes are in use. XFS allows each allocation group to function in parallel, allowing for a greater number of simultaneous file operations. (This is described in greater detail in section 3.1.)

VxFS allocates inodes dynamically in each allocation unit. The mechanism used to track their location is undocumented.

All files in NTFS are accessed through the master file table (MFT). Because the MFT is a substantially different approach to managing file information than the other filesystems use, it requires a brief introduction. Everything within NTFS, including metadata, is stored as a file accessible in the filesystem namespace and described by entries in the MFT.

When an NTFS volume is created, the MFT is initialized with the first entry in the MFT describing the MFT itself. The next several files in the MFT describe other metadata files such as the bitmap file that maps free versus allocated space and the log file for logging filesystem transactions. When a file or directory is created in NTFS, an MFT record is allocated to store the file and point to its data. Because this changes the MFT itself, the MFT entry must also be updated. The MFT is allocated space throughout the disk volume as needed. As the MFT grows or shrinks, it may become highly fragmented. Note that this may pose a significant problem since the MFT is a linear structure. While the remainder of the filesystem can be defragmented by third party utilities, the MFT itself cannot be defragmented. The MFT can therefore become highly fragmented in a filesystem subject to a large number of file creations and deletions, resulting in inefficient operation. Access to the MFT is single-threaded, so NTFS lacks the inherent parallelism of XFS.

3.4.1 Small Files

Since applications that create large numbers of files frequently create very small files, it seems appropriate to discuss small files at this point. If the smallest unit of disk allocation is a block (as large as 64KB in some filesystem implementations) then disk space is wasted any time a file smaller than 64KB is stored in that block. This is frequently referred to in the literature as internal fragmentation. Various approaches exist to handle small files more efficiently.

UFS, which was developed when disk space was comparatively limited and expensive, provides a mechanism to break blocks into smaller units (fragments or frags) for the storage of small files. This process is used until the file exceeds the size that can be stored in direct block pointers. Beyond that size, only whole blocks are allocated.

XFS was designed with large and relatively inexpensive storage devices in mind. Nevertheless, it allows very small files like symbolic links and directories to be stored directly in the inode to conserve space. This also accelerates access to such files since no further disk accesses are needed once the inode is read. The default inode size is 256 bytes, but this can be made larger when the filesystem is created, allowing for more space to store small files. XFS also allows the logical block size of the filesystem to be set on a per-filesystem basis. The allowable sizes range from 512 bytes to 64KB. Smaller sizes are ideal for filesystems used by applications like news that typically store a large number of small files.

VxFS has a 96-byte “immediate area” in each inode that can be used to store symbolic links and small directories. NTFS also allows small files and directories to be stored directly in the MFT record for the file.

4. Rapid Crash Recovery

The slowest part of the I/O system is the disk drive since its performance depends in part on mechanical rather than electronic components. To reduce the bottleneck created by disk drives to overall I/O performance, filesystems cache important information in memory from the disk image of the filesystem. This information is periodically flushed to disk to synchronize it.

While caching is essential to system performance, an unexpected system disruption can have serious consequences. Since the latest updates to the filesystem may not have been transferred from memory to disk, the filesystem will not be in a consistent state. Traditional filesystems use a checking program to examine the filesystem structures and return them to consistency.

As filesystems have grown larger and servers have grown to support more and more of them, the time taken by traditional filesystems to run these checking programs and recover from a crash has become significant. On the largest servers, the process can take many hours. Filesystems that depend on these checking programs also must keep their internal data structures simple to preserve the efficiency of those programs. But as the previous section demonstrates, simple data structures and algorithms may be inefficient for large filesystems.

Most modern filesystems use journaling techniques borrowed from the database world to improve crash recovery. Disk transactions are written sequentially to an area of disk called the journal or transaction log before being written to their final locations within the filesystem. These writes are generally performed synchronously, but gain some acceleration because they are performed sequentially and written contiguously. If a failure occurs, these transactions are replayed from the journal to ensure the filesystem is up to date. Implementations vary in terms of what data is written to the log. Some implementations log only filesystem metadata changes, while others log all filesystem writes. The journaling filesystems discussed in this paper log only metadata during normal operation. (The Veritas filesystem adds the ability to log small synchronous writes to accelerate database operations.) Depending on the implementation, logging may have significant consequences for I/O performance.

It is also important to note that using a transaction log does not make the use of filesystem checking programs entirely obsolete. Hardware and software errors that corrupt random blocks in the filesystem are not generally recoverable with the transaction log, yet these errors can make the contents of the filesystem inaccessible. This type of event is relatively rare, but still important.

This section examines the implementations used by the various filesystems and discusses the implications of those methods.

UFS
Traditionally, UFS did not provide journaling. In case of a system failure, the program fsck is used by UFS to check the filesystem. This program scans through all the inodes, the directories, and the free block lists to restore consistency. The key point is that everything must be checked whether it has been changed recently or not.

Numerous enhancements have been implemented within UFS over the years to try to overcome this problem. Some implementations have clean flags that are set every time the filesystem is synced and then unset every time it is altered. In practice, this can have a dramatic effect on the reboot time of systems with many filesystems, although recovery time is unpredictable. If the clean flag is set, the filesystem does not have to be checked. A further enhancement along this same line adds clean flags for each cylinder group in the filesystem, further reducing the amount of filesystem data that has to be checked.

Sun implemented a journal for the UFS filesystem several years ago as part of its optional DiskSuite product. This technology was bundled in the Enterprise Server Edition of Solaris 2.7, which was announced in October 1998 and released in 1999. The journal is implemented as a separate log-structured write cache for the filesystem. After logging, blocks of metadata are not maintained in the in-memory cache. Therefore, under load when the log fills and needs to be flushed, blocks actually have to be reread from the log and then written to the filesystem. This filesystem is generally recommended for use in situations where availability, not performance, is the primary concern.

XFS
XFS logs all updates to the filesystem metadata before user data is committed to disk. This includes inodes, directory blocks, free extent tree blocks, inode allocation tree blocks, file extent map blocks, AG header blocks, and the super block. XFS does not write user data to the log. Logging new copies of the modified items makes recovering the XFS log independent of both the size and the complexity of the filesystem. Recovering the data structures from the log requires nothing but moving the block and inode images in the log to the appropriate locations in the filesystem. The log recovery process does not know that it is recovering a B+ tree. It only knows that it is restoring the latest images of some filesystem blocks.

Traditional write ahead logging schemes write the journal 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 filesystem to the rate at which it can write the journal. While XFS provides a mode for making synchronous journal updates for use when the filesystem is exported via NFS(tm), the normal mode of operation for XFS is to use asynchronous logging. XFS still ensures that the write ahead logging protocol is followed in that modified data cannot be flushed to disk until after the metadata is committed to the journal. Rather than keep the modified resources locked until the transaction is committed to disk, the resources are instead unlocked and pinned in memory until the transaction is fully committed. 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 filesystem.

XFS gains two advantages 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. 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 filesystems.

In situations where metadata updates are very intense, the log can be stored on a separate device such as a dedicated disk or a nonvolatile memory device. This is particularly useful when a filesystem is exported via NFS, which requires that all transactions be synchronous.

VxFS
VxFS employs a metadata journaling scheme in many respects similar to that used by XFS. The VxFS journal is updated synchronously, so it lacks the performance benefits of the XFS implementation. In addition to metadata logging, VxFS allows small synchronous writes to be logged as well. This may be advantageous for databases running within the filesystem; however, most database vendors recommend running their products in raw disk volumes rather than filesystems. VxFS does allow the log to be placed on a separate device if desired.

NTFS
NTFS also uses a synchronous metadata journal that is typically a few megabytes in size. The journal is embedded in the MFT, and therefore cannot be allocated on a device separate from the rest of the filesystem.

5. Filesystem Performance

Most of the issues discussed in sections 3 and 4 contribute to performance by alleviating or avoiding potential bottlenecks. However, they are not the whole story when it comes to maximizing the I/O performance of the underlying hardware. This section describes the performance characteristics of the various filesystems in more detail and discusses performance as measured by the various vendors.

5.1 Factors Contributing to I/O Throughput

Modern servers typically use large, striped disk arrays capable of providing an aggregate bandwidth of tens to hundreds of megabytes per second. The keys to optimizing the performance from 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 disk array, this need for large requests is increased as individual requests are broken up into smaller requests to the individual drives. Many requests must be issued in parallel to keep all of the drives in an array busy.

Large I/O requests to a file can only be made if the file is allocated contiguously, because the number of contiguous blocks in the file being read or written limits the size of a request to the underlying drives. UFS uses block allocation and in general does not attempt to allocate files in a contiguous fashion so is therefore limited in this regard.

XFS has the most advanced features for allocating files contiguously. XFS can allocate single extents of up to 2 million blocks and uses B+ trees to manage free space so that appropriately sized extents can be found rapidly (see section 3.2). In addition, XFS uses delayed allocation to ensure that the largest possible extents are allocated.

Rather than allocating specific blocks to a file as it is written in the buffer cache, XFS simply reserves blocks in the filesystem 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 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 often prevents short-lived files from ever having any real disk blocks allocated to contain them. Even files that are written randomly, such as memory mapped files, can often be written contiguously because of delayed allocation.

It is the job of the XFS I/O manager to read and write a file in requests large enough 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 the underlying disk array.

To obtain the best possible sequential read performance, XFS uses large read buffers and multiple read ahead buffers. For sequential reads, a large minimum I/O buffer size (typically 64KB) is used. The size of the buffers is reduced to match the file for files smaller than the minimum buffer size. Using a large minimum I/O size ensures that, even when applications issue reads in small units, the filesystem feeds the disk array requests that are large enough for high I/O performance. For larger application reads, XFS increases the read buffer size to match the application's request.

XFS uses multiple read ahead buffers to increase I/O parallelism. Traditional UNIX systems use only a single read ahead buffer. For sequential reads, XFS keeps outstanding two to three requests of the same size as the primary I/O buffer. Multiple read ahead requests keep the drives in the array busy while the application processes the data being read. A large number of read ahead buffers ensures a large number of underlying drives are kept busy at once.

To improve write performance, XFS uses aggressive write clustering. Modified file data is buffered in memory in chunks of 64KB, 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 data buffered 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 modified data waiting to be written and the need to keep I/O requests streaming out to the underlying disk array.

Various UFS implementations take advantage of read and write clustering in a manner analogous to XFS, but none goes to the extreme degree that XFS does to ensure I/O throughput. Since VxFS lacks the tight integration with the host operating system that XFS has, it is difficult for VxFS to provide these optimizations. While Windows NT and NTFS have the necessary integration, so far Microsoft has not chosen to implement any of the optimizations discussed here to any significant extent, most likely because the need has not yet arisen in typical Windows NT environments.

For very large file I/O-when the file size nears or exceeds the size of physical memory-caching file data in memory can actually impede performance. In such situations, it may be desirable to use Direct I/O which bypasses the buffer cache and reads and writes data directly to and from disk. The XFS implementation of Direct I/O supports preallocation, allowing an application to request that contiguous space be allocated before it is needed. By enhancing contiguous allocation for applications using Direct I/O, preallocation can substantially improve throughput. Direct I/O is discussed further in section 6.

5.2 Measured Performance

Unfortunately, it is impossible to do direct performance comparisons of the filesystems under discussion since they are not all available on the same platform. Further, since available data is necessarily from differing hardware platforms, it is difficult to distinguish the performance characteristics of the filesystem from that of the hardware platform on which it is running. However, some conclusions can be drawn from the available information.

If we look simply at measured system throughput, there is no question that XFS is the hands-down winner. SGI recently announced total throughput of 7.32GB/sec on a single filesystem. The system was an SGI Origin 2000 configured with 32 processors and 897 9GB fibre channel disks. Single file numbers were 4.03GB/s read performance and 4.47GB/sec write performance. These numbers were obtained using Direct I/O, which bypasses the buffer cache.

The next closest measured system throughput number was achieved using VxFS on an 8-processor Sun UltraSPARC 6000 configured with 4+ Terabytes of disk. With this configuration a maximum throughput of 1.049 GB/sec was also achieved using Direct I/O, which bypasses the system's buffer cache.

No similar measurements have been reported with UFS or NTFS. In more modest comparison tests, however, Veritas has found that VxFS generally outperforms UFS in most situations. Because of the differences in typical system configurations, there are no valid comparisons that can be drawn between NTFS throughput tests and the tests described above. Typical NTFS test systems have relatively few disks and throughput is generally in the range of tens rather than hundreds or thousands of megabytes per second.

Another way of examining filesystem performance and efficiency is by comparing the number of operations per drive achieved using the SPECnfs benchmark. This benchmark is the standard benchmark for testing NFS file server performance. The comparisons discussed here are based on tests performed with the SPECnfs_A93 benchmark, which uses NFS version 2 only. Again, there is no way to control for performance differences based on underlying hardware differences. Where possible, results have been chosen that reflect approximately the same level of performance from roughly comparable systems.

In a test performed in April 1997 a single processor Origin 200 system with 256MB of memory and 30 disk drives demonstrated 2,822 SPECnfs operations per second using XFS as the filesystem. This corresponds to 94 SPECnfs operations per disk.

A similar test was performed in April 1996 on a single processor Sun Enterprise 3000 with 1GB of memory and 37 disks. That system achieved a maximum of 2,004 SPECnfs operations per second using UFS or 54 SPECnfs operations per disk.

The above two tests were chosen because of the approximately similar system configurations (although note that the Sun system had four times more memory) and because these are among the smallest system configurations tested. This allows for a limited comparison with tests performed by Veritas using VxFS. The system Veritas used for its testing was a dual-processor SPARCstation 10 with 64 MB of memory and 6 disk drives. This system was tested using VxFS, UFS, and UFS with journaling. VxFS achieved 76 SPECnfs operations per disk while UFS demonstrated 64 SPECnfs operations per disk without journaling and 20 SPECnfs operations per disk with journaling enabled. To date no SPECnfs results have been reported for machines using NTFS.

6. Other Features

In addition to the features already described, each of the filesystems discussed here offers a number of other features that may be important for some applications. This section discusses some of the more unique features briefly.

UFS

Because of its long life span, UFS has been enhanced in many areas. However, none of its incarnations has any single unique feature that all the others lack. This is perhaps not surprising since UFS is in many senses a precursor of XFS and VxFS.

XFS

The most unique feature of XFS is its support for Guaranteed Rate I/O (GRIO), which allows applications to reserve bandwidth to or from the filesystem. XFS calculates the performance available and guarantees that the requested level of performance is met for a specified time. This frees the programmer from having to predict performance, which can be complex and variable. This functionality is required for full rate, high-resolution media delivery systems such as video-on-demand or satellite systems that need to process information at a certain rate. By default, XFS provides four GRIO streams (concurrent uses of GRIO). The number of streams can be increased to 40 or more using the High-Performance Guaranteed-Rate I/O-5-40 option or the Unlimited Streams option.

XFS provides full real-time support through the optional use of a real-time subvolume. This feature is used by GRIO, but a programming interface exists independent of GRIO that allows application developers to provide real-time access for applications that require it.

The XFS inode also provides support for user-mode attributes-the ability to store user-defined data about the file. SGI makes use of this feature to provide Access Control Lists (ACLs) for secure IRIX, and it is also used to support DMAPI, the Data Migration API (DMAPI). Using this API, storage management applications such as Silicon Graphic's Data Migration Facility (DMF) can take advantage of advanced hierarchical storage management. Data can be easily migrated from online disk storage to near-line and off-line storage media. This allows for easy management of data sets much larger than the capacity of online storage. Using this technology, some customers are already managing over 300 Terabytes of data with plans to scale up to a Petabyte or more. With data storage doubling every year, it quickly becomes clear why a full 64-bit filesystem like XFS will soon be essential. VxFS also has DMAPI support, but lacks the extreme scalability of XFS as discussed in section 3.

VxFS

Veritas promotes the online administration features of VxFS and its support for databases. On-line features include the ability to grow or shrink a filesystem, snapshot backup, and defragmentation. In this regard, XFS offers the ability to grow (but not shrink) a filesystem and online consistent backup, but no snapshots. XFS does not currently offer online defragmentation because in practice, given the delayed allocation algorithms used by XFS and the large size of typical XFS filesystems, fragmentation has not been a big problem. Defragmentation will be added in the future.

The database support features in VxFS allow applications to avoid having their data cached in system memory by the operating system. Operating systems typically try to cache recently used or likely to be requested blocks in memory to speed up access in the event that the data is needed again. Each block read or written passes through this buffer cache. However, the algorithms used can actually be detrimental to certain types of applications such as databases and/or other applications that manipulate files larger than system memory. Database vendors frequently choose to use raw I/O to unformatted disk volumes to avoid the penalty created by the buffer cache, but this requires the application to deal with the underlying complexities of managing raw disk devices.

Direct I/O allows an application to specify that its data not be cached in the buffer cache. This allows the application to take advantage of filesystem features and backup programs. Typically, a mechanism is also provided to allow multiple readers and writers to access a file at one time. Both XFS and VxFS support Direct I/O. Sun added support for Direct I/O to its version of UFS in Solaris 2.6. In practice most databases prefer to be installed on raw disk volumes, so it is not clear that this feature provides any great advantage to databases although other large I/O applications might take advantage of it.

NTFS

Perhaps the sole unique feature of NTFS is its support for compression, which allows files to be compressed individually, by folder or by volume. Of course, the utilities to perform similar tasks have existed in the UNIX operating system for a long time. In terms of performance, making compression the task of the operating system is probably counter-productive, and it is likely that future generations of disk drives will provide hardware-based compression. The next version of NTFS will support encryption, but again, the utilities already exist in UNIX for those who need encryption.

7. Conclusion

It is clear that XFS is unrivaled in the management of large filesystems, large files, large directories, large numbers of files and overall filesystem performance. Based on its from-scratch design and use of advanced data structures, XFS is able to scale where other filesystems would simply fail to perform. At the same time, XFS provides enhanced reliability and rapid crash recovery without hampering performance through its use of asynchronous journaling.

Each of the filesystems discussed here has its strengths. UFS has been in use for years, so many system administrators are familiar and comfortable with it. VxFS provides modest improvements over UFS and probably works well in many environments. NTFS is the only advanced filesystem currently available for Windows NT, and therefore has a captive audience. However, if your application requires the manipulation of huge data files with the fastest possible I/O performance, XFS is the only solution. XFS offers unparalleled performance and data security. Because XFS enables SGI computer systems to do more work faster, it offers significant time-to-market advantages to those who deploy it.

8. Glossary

Allocation Group—The subunit into which XFS divides its disk volumes. Each allocation group is responsible for managing its own inodes and free space and can function in parallel with other allocation groups in the filesystem.

B+ tree—The B+ tree is a data structure that is used throughout XFS to index and accelerate access to important filesystem data. The B+ tree structure takes the form of an inverted tree, in some ways analogous in form to a directory hierarchy. The tree can be efficiently searched by descending from the root, making simple comparisons between the desired value and values stored in the tree. The B+ tree is particularly well suited to paged files because it supports random or sequential access to data stored within the tree.

Bitmap—Bitmaps are often used to keep track of allocated versus free space in filesystems. Each block in the filesystem is represented by a bit in the bitmap. In place of bitmaps, XFS maintains two b-trees: one which indexes free extents by size, and another that indexes free extents by starting block.

Block Allocation—A disk space allocation method in which a single block is allocated at a time and a pointer is maintained to each block.

Cylinder Group—The subunit into which UFS divides its disk volumes. Each cylinder group contains inodes, bitmaps of free space and data blocks. Typically about 4MB in size, cylinder groups are used primarily to keep an inode and its associated data blocks close together and thereby to reduce disk head movement and decrease latency.

Cluster (Windows NT)—In Windows NT parlance, the term cluster is defined as a number of physical disk blocks allocated as a unit. This is analogous to the term logical block size that is generally used in UNIX environments.

Clustering (UNIX)—A technique that is used in UFS to make its block allocation mechanism provide more extent-like allocation. Disk writes are gathered in memory until 56KB has accumulated. The allocator then attempts to find contiguous space and if successful performs the write sequentially.

Direct Block Pointer—Inodes in the UNIX Filesystem (UFS) store up to 12 addresses that point directly to file blocks. For files larger than 12 blocks, indirect block pointers are used.

Indirect Block Pointer—Inodes in the UNIX filesystem (UFS) store 12 addresses for direct block pointers and 3 addresses for indirect block pointers. The first indirect block pointer, called the single indirect block pointer, points to a block that stores pointers to file blocks. The second, called the double indirect block pointer, points to a block that stores pointers to blocks that store pointers to file blocks. The third address, the triple indirect block pointer, extends the same concept one more level.

Extent—An extent is a contiguous range of disk blocks allocated to a single file and managed as a unit. The minimum information needed to describe an extent is its starting block and length. An extent descriptor can therefore map a large region of disk space very efficiently.

Extent Allocation—A disk space allocation method in which extents of variable size are allocated to a file. Each extent is tracked as a unit using an extent descriptor that minimally consists of the starting block in the filesystem and the length. This method allows large amounts of disk space to be allocated and tracked efficiently.

External Fragmentation—The condition where files are spread in small pieces throughout the filesystem. In some filesystem implementations, this may result in unallocated disk space becoming unusable.

Filesystem—The software used to organize and manage the data stored on disk drives. In addition to storing the data contained in files, a filesystem also stores and manages important information about the files and about the filesystem itself. This information is commonly referred to as metadata. See also metadata.

Fragment—Also frag. The smallest unit of allocation in the UFS filesystem. UFS can break a logical block into up to 8 frags and allocate the frags individually.

Inode—Index node. In many filesystem implementations an inode is maintained for each file. The inode stores important information about the file such as ownership, size, access permissions, and time-stamps (typically for creation, last modification and last access), and stores the location or locations where the file's blocks may be found. Directory entries typically map file names to inode numbers.

Internal Fragmentation—Disk space that is allocated to a file but not actually used by that file. For instance, if a file is 2KB in size, but the logical block size is 4KB, then the smallest unit of disk space that can be allocated for that file is 4KB and thus 2KB is wasted.

Journal—An area of disk (sometimes a separate device) where filesystem transactions are recorded prior to committing data to disk. The journal ensures that the filesystem can be rapidly recovered in a consistent state should a power failure or other catastrophic failure occur.

Log—See journal.

Logical Block Size—The smallest unit of allocation in a filesystem. Physical disk blocks typically 512 bytes in size. Most filesystems use a logical block size that is a multiple of this number; 4KB and 8KB are common. Using a larger logical block size increases the minimum I/O size and thus improves efficiency. See also cluster (Windows NT).

Master File Table—The table that keeps track of all allocated files in NTFS. The Master File Table (MFT) takes the place of the inodes typically used in UNIX filesystem implementations.

Metadata—Information about the files stored in the filesystem. Metadata typically includes date and time stamps, ownership, access permissions, other security information such as access control lists (ACLs) if they exist, the file's size, and the storage location or locations on disk.

MFT—See Master File Table.

RAID—Redundant Array of Independent Disks. RAID is a means of providing for redundancy in the face of a disk drive failure to ensure higher availability. RAID 0 provides striping for increased disk performance but no redundancy.

Run—(Windows NT) The Windows NT literature generally refers to chunks of disk space allocated and tracked as a single unit as Runs. These are more commonly referred to in the general literature as extents. See also extent.

Volume—A span of disk space on which a filesystem can be built, consisting of one or more partitions on one or more disks. See also Volume Manager.

Volume Manager—Software that creates and manages disk volumes. Typical volume managers provide the ability to join individual disks or disk partitions in various ways such as concatenation, striping, and mirroring. Concatenation simply joins multiple partitions into a single large logical partition. Striping is similar to concatenation, but data is striped across each disk. Blocks are numbered such that a small number are used from each disk or partition in succession. This spreads I/O more evenly across multiple disks. Mirroring provides for two identical copies of all data written to the filesystem to be maintained and thereby protects against disk failure.

sgi™

Corporate Office
1600 Amphitheatre Pkwy.
Mountain View, CA 94043
(650) 960-1980
www.sgi.com

North America 1(800) 800-7441
Latin America 1(650) 933-4637
Europe (44) 118.925.75.00
Japan (81) 3.5488.1811
Asia Pacific (65) 771.0290