Highly Scalable, High Performance Perforce Server Environments

Shiv Sikand and Roger March

Matrix Semiconductor, Inc.

Perforce User Conference, May 2003, Las Vegas, USA

Introduction

The performance and scalability of large Perforce installations is heavily dependent on the file system provided by the operating system. We will investigate the design and implementation of select file systems and perform some simple benchmarks to determine their relative advantages. For this discussion, file system scalability is defined as the ability to support very large numbers of files and directories while providing good I/O performance

Definitions

The file system is one of the most important parts of an operating system. It stores and manages data on disk drives and ensures that what is read from the storage is identical to what was originally written. In addition to storing files, the file system also creates and manages information about files and about itself. In this paper, we will look at first generation file systems such as UFS in Solaris, second generation UFS derivatives such as WAFL (Write Anywhere File System) from Network Appliance and new implementations of second generation file systems, namely XFS on Linux and NTFS on Windows 2000.

File system architecture

The way file systems manage their metadata is the key to their performance and scalability. A file system stores data on your hard drive by determining where each file's blocks content should be stored and by maintaining a table of the locations of those blocks. It 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. The data structure specifying the location of a block (or a set of blocks) is called an inode. The particular methods for allocating and retrieving file blocks determine the overall performance of reads and writes to disk and the reliability of the file system itself.

Most first generation UNIX file systems are based on the Berkeley Fast File System commonly known as UFS or FFS. The first generation Microsoft file systems, FAT16 and FAT32 will not be discussed here due to their primitive nature. In the FFS design, the file system maintains a map of inodes, which point to data and directory blocks; each inode has a number that uniquely identifies it. Directory blocks contain a table associating a list of inode numbers with the names of the files and other directories stored in that directory. A file's inode contains information describing the file. This information includes the files inode number, metadata such as ownership and permission information, size of the file, date the file was last accessed or modified, and a list of each of the file's data blocks. For large files, this list can also contain an indirect block, which are blocks that themselves contain lists of blocks allocated to the file. In turn, indirect blocks can

contain lists of double-indirect blocks and double-indirect blocks can point to triple indirect blocks. For very large files, this method becomes extremely inefficient. UFS uses traditional bitmaps to keep track of free space and in general the block size used is fixed by the implementation. Its file data is allocated randomly based on the availability of free space. UFS divides its file systems into cylinder groups. Each cylinder group contains bookkeeping information including inodes 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. The inode metadata is clustered in cylinder groups to help improve its locality. This method of allocation at creation also tends to make read performance a function of the initial layout.

The UFS file system was designed at a time when 32-bit computing was the norm. Originally, it only supported 2^31 bytes, but current implementations have been extended to support up to 2TB.

The techniques developed to allow UFS to survive the inevitable crash has had a major impact on its performance characteristics. To minimize the possibility of damage it would be desirable to synchronously write all data. The performance penalty for this is so large that UFS has resorted to asynchronous writes for file data and synchronous writes for the metadata.

The next generation

Second generation file systems have been designed to add support for large file systems, large files, sparse files, large directories, large numbers of files, rapid crash recovery and high I/O throughput and RAID support.

RAID primer

RAID is an acronym for Redundant Array of Inexpensive Disks. A RAID array is a collection of drives which collectively act as a single storage system that can tolerate the failure of a drive without losing data and which can operate independently of each other. There are a few different RAID levels, but we will only consider the most important for performance and scalability.

RAID Level 0 is not redundant, but data is split across drives, resulting in higher data throughput. Since no redundant data is stored, performance is very good, but the failure of any disk in the array results in data loss. This level is commonly referred to as striping.

RAID Level 1 provides redundancy by duplicating all data from one drive on another drive. The performance of a level 1 array is only slightly better than a single drive but the cost per megabyte is doubled. This level is commonly referred to as mirroring.

RAID Level 4 stripes data at a block level across several drives, with parity stored on one drive. The parity information allows recovery from the failure of any single drive. The

performance of a level 4 array is very good for reads. Writes, however, require that parity data be updated each time. This slows small random writes, in particular, though large writes or sequential writes are fairly fast. Because only one drive in the array stores redundant data, the cost per megabyte of a level 4 array can be fairly low.

RAID Level 10 is implemented as a striped array whose segments are RAID Level 1 arrays. High I/O rates are achieved but it is expensive since it requires a minimum of 4 drives to implement.

RAID Level 0+1 is implement as a mirrored array whose segments are RAID Level 0 arrays. High I/O rates are achieved thanks to multiple stripe segments but a single drive failure will cause the array to become, in essence, a RAID Level 0 array.

Development histories

XFS was designed by SGI in the 1990's from scratch to meet the needs of its customers for file system scalability and performance. It was ported to Linux from IRIX in 2000. The design of NTFS began in the late 1980's. While NTFS could be considered a second generation file system based on the fact that it was developed from scratch, many of the scalability lesson that could have been learned from first generation UNIX file systems seem to have been ignored.

Network Appliance designed a file system designed to work specifically in an NFS appliance, but it can trace its root to UFS. The primary focus of the design was to implement a scheme for Snapshots, which are read-only clones of the active file system. In addition, since they decided to use a RAID Level 4 architecture, WAFL contains enhancements to mitigate the write penalty for parity computation as well as extended crash recovery features.

Crash recovery

It is common knowledge that 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, file systems cache important information from the disk image of the file system in memory. This information is periodically flushed to disk, to synchronize it.

This caching process, so essential to system performance, has serious consequences in the event of an unexpected system crash or power outage. Since the latest updates to the file system may not have been transferred from memory to disk, the file system will not be in a consistent state. Traditional file systems use a checking program to go through and examine the file system structures and return them to consistency.

As file systems have grown larger and servers have grown to support more and more of them, the time taken by traditional file systems to run these checking programs and recover from a crash has become significant. On the largest servers, the process can literally take many hours as the program sweeps through all data. File systems that depend on these checking programs also must keep their internal data structures simple to preserve any efficiency. Having critical data offline for long periods of time during the repair process is considered an unacceptable cost in many environments.

Most modern file systems 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 log before being written to their final locations within the file system. These writes are generally performed synchronously, but gain some acceleration because they are contiguous. If a failure occurs, these transactions can then be replayed from the journal to ensure the file system is up to date. Implementations vary in terms of what data is written to the log. Some implementations write only the file system metadata, while others log all writes to the journal. The journaling file systems discussed in this paper log only metadata during normal operation. Depending on the implementation, journaling may have significant consequences for I/O performance.

It is also important to note that using a transaction log does not entirely obsolete the use of file system checking programs. Hardware and software errors that 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. This type of event is relatively rare, but still important.

Filesystem Feature Comparisons

Feature	XFS	UFS	NTFS
Max FS Size	18 million TB	1 TB	2 TB
Max File Size	9 million TB	1 TB	2 TB
File Space Allocation	Extents	Blocks	Extents
Max Extent Size	4 GB	NA	Undocumented
Free Space Management	Free extents	Bitmap per cylinder group	Single bitmap
Variable Block Size	512 bytes to 64 KB	4KB or 8KB	512 bytes to 64KB
Sparse file support	Yes	Yes	NTFS 5.0
Directory organization	B+ Tree	Linear	B+ Tree
Inode allocation	Dynamic	Static	Dynamic
Crash recovery	Asynchronous Journal	Fsck	Synchronous Journal

File and free space allocation

One of the goals of modern file systems is to reduce fragmentation. External fragmentation is the condition where files are spread in small pieces throughout the file system. In some file system implementations, this may result in unallocated disk space becoming unusable. Internal Fragmentation is 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.

XFS and NTFS both use 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.

In XFS, the number of files in a file system is limited only by the space available to hold them. XFS dynamically allocates inodes as needed. Each allocation group manages the inodes within its confines. Inodes are allocated sixty-four 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.

All files in NTFS are accessed through the master file table or MFT. Because the MFT is a substantially different approach to managing file information than the other file systems use, it requires a brief introduction. Everything within NTFS, including metadata, is stored as a file accessible in the file system 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 file system 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 file system can be de-fragmented by third party utilities, the MFT itself cannot be de-fragmented. The MFT appears to be strictly single-threaded.

Free space in XFS is managed on a per Allocation Group basis. Each allocation group maintains two B+ trees 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 file system 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 64 kilobytes on a per file system basis.

NTFS manages free space using bitmaps like UFS. However, unlike those file systems, 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.

Performance measurements

We conducted a series of tests on different hardware configurations to determine the file-system performance of different machines running Perforce 2002.1.

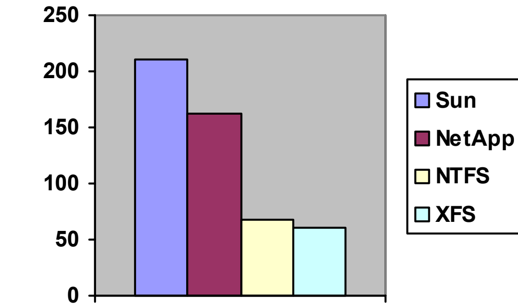
Config 1: Sun 280R Server, Solaris 2.8, UFS, no RAID
Config 2: Sun 280R Server, Solaris 2.8, combined with Network Appliance F820
Config 3: Dell PowerEdge 1600 SC, SCSI HW RAID1, Windows 2000 AS, NTFS
Config 4: Dell PowerEdge 1600 SC, SCSI HW RAID1, Linux Mandrake 9.0, XFS

Dataset information

Number of files	186,177
Number of directories	114,902
Smallest file	4 K
Largest file	57 M
Most common	94,725 @ 4 K 20,148 @ 8 K 16,219 @ 12 K
Number of different file sizes	168
Number of files > 1M	53
Total data size	1.9 G

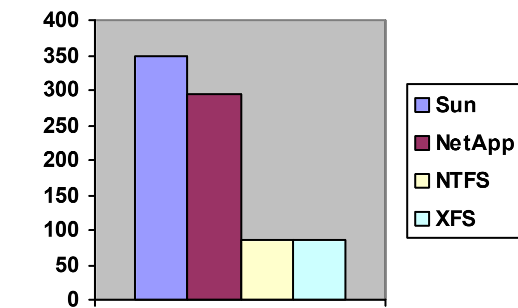
Test 1

Import from our production depot using **p4 integ**. The hierarchy of the data is shallow and broad, representing a typical branch of our 64MB 3D-Memory hardware. This test demonstrates file system write performance.



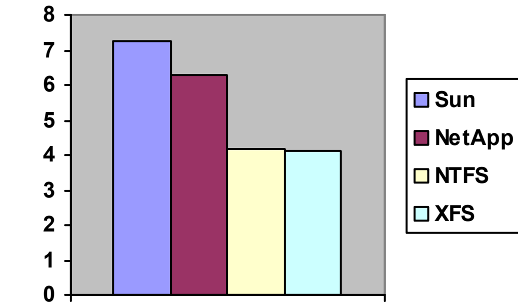
Test 2

p4 submit of the above data. This test demonstrates simultaneous read-write performance of a volume.



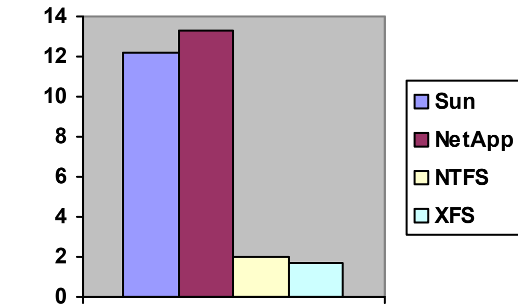
Test 3

Branch integrate time of imported data using **p4 integ -v**



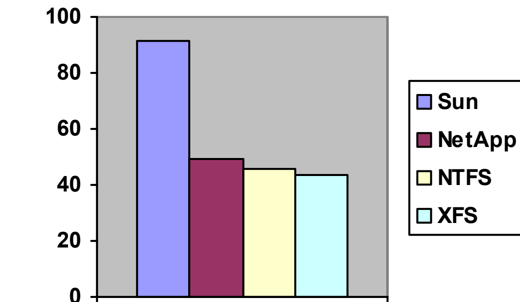
Test 4

Branch submit time



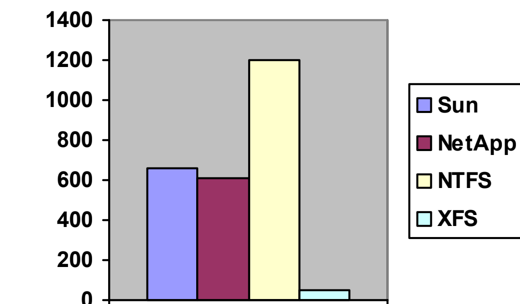
Test 5

Two client simultaneous sync (average time)



Test 6

Delete performance



Analysis

The import test clearly shows the advantage to be gained by second generation file systems in the area of write performance. The data shows the SUN clearly struggling under the weight of synchronously updating its metadata during file creation. The NetApp uses write logging with impressive disk hardware and manages only a modest improvement over the SUN. It is likely that the parity drive in its RAID4 configuration presents a bottleneck to better performance. In the case of both the NTFS and XFS results it appears that the network was the limiting factor.

The submit test stresses the simultaneous read/write performance of the file systems. Here the SUN reaps what it has sown. Its read performance possibly degraded by its poor allocation policy and the write bandwidth suffering from many synchronous writes. The NetApp result is a bit surprising, it has closed with the SUN rather than distancing itself. The NetApp pays an extra metadata penalty relative to the SUN to support features like

snapshots, but it seems unlikely this could account for their closeness. The closeness of NTFS and XFS seems to indicate another hardware/software limit may have been reached. They runs were done on the same hardware using similar software. It appears both file systems perform so well they are not the bottleneck.

The integ –v and branch submit test shows the large difference between the PC solutions and the others. It's not clear why the NetApp configuration submit time was slower than the pure UFS case.

The test involving the two simultaneous syncs were performed by having two separate machines pull their data off the server. The results show the SUN again crippled by its file system. The NetApp finally gets a chance to show off its excellent read performance and both NTFS and XFS make fine showings. There is a suspicion that the 100 MB/s network link was a limiting factor for NTFS and XFS. They served a 200MB/s demand through a 100 MB/s interface. This argument raises questions about the NetApp, it had a 1000 MB/s interface to service the same load and yet did not perform better than NTFS or XFS.

The delete test was done primarily to illustrate some differences in the NTFS and XFS file systems. They have both shown uniformly good results for Perforce operations but under the hood they have some difference. Both file systems de-emphasis deletes over other file operations. The architecture of the MFT in NTFS causes it to pay an enormous penalty over than incurred by XFS. When properly used, this behavior should not present a barrier to using NTFS, but it is something to be aware of.

Conclusion

In the hardware design community, the poor performance of the UFS file system has been known for some time. It is typically resolved by throwing expensive hardware at the problem. Either by acquiring a larger, faster SUN server equipped with RAID or using a NetApp to hold the databases. Our experience in applying Perforce to hardware design environments show that, unsurprisingly, its performance is strongly influenced by the underlying file system.

Based on our results it appears that neither the SUN nor NetApp solution is an optimal choice for hosting a Perforce server. In most cases, raw performance is significantly inferior to either NTFS or XFS. From the cost standpoint the non-PC solutions are especially hard to justify, our NetApp being quite a bit more than order of magnitude more expensive than the PC. There are still legitimate reasons to use a SUN or NetApp solution, total available file system capacity being one of them. The lack of a high performance, reliable and scalable file system on the PC is no longer a reason.

Though NTFS performed comparably to XFS there appear to be some additional constraints for NTFS. One observation is that NTFS seems to have a significantly larger memory requirement. The belief is that Windows achieves its performance by keeping nearly the entire MFT in memory. There are concerns that its memory usage may degrade

performance in those cases where Perforce's memory demands are high. Fragmentation is also a concern with NTFS and it seems prudent that the depot be kept on its own partition to minimize the effect.

Information sources

Raid Levels, Advanced Computer and Network Corporation
File System Design for an NFS File Server Appliance, Dave Hitz et al., NetApp TR3002
Linux Magazine, October 2002 Journalling File Systems, Steve Best, Moshe Bar
Scalability and Performance in Modern File Systems, Philip Trautman and Jim Mostek