Breaking Through the NFS Performance Barrier

Joseph Moran
Russel Sandberg
Don Coleman
Jonathan Kepecs
Bob Lyon
jkepecs@legato.com

Legato Systems, Inc.
260 Sheridan Ave.
Palo Alto, CA. 94306
U.S.A.

ABSTRACT

The Sun Network File System (NFS) has become popular because it allows users to transparently access files across heterogeneous networks. NFS supports a spectrum of network topologies, from small, simple and homogeneous, to large, complex, multi-vendor networks. Given the diversity and complexity of NFS environments, isolating performance problems can be difficult. This paper identifies common NFS performance problems, and recommends specific practical solutions. In particular, we evaluate the impact of reliable write caching on NFS performance.

Introduction

NFS [San85] performance problems can be broken down into three basic areas: client, network, and server bottlenecks. We will explore each of these areas and show why the server, and in particular, the server's I/O subsystem, is usually the primary cause of poor NFS performance. We demonstrate that reliable write caching can have a major impact on NFS performance improvement by eliminating slow, synchronous disk I/O.

Network Bottlenecks

Analysis of the network used to communicate between the client and server may turn up bottlenecks in addition to the inherent latency of the network. Three important factors to evaluate include network load, network topology, and packet loss.

Network Load

If the network is over-utilized, clients will experience longer delays waiting for a free slot to send requests in. However, it is a mistake to assume that an Ethernet can only handle a small fixed load because of increasing collision rates. Boggs et. al. [Bog88] have explored this issue in detail and demonstrate that in many cases, a heavily loaded Ethernet can deliver its full bandwidth. Even with 24 hosts continuously sending maximum-length packets, only about 3% of the delay in sending is due to collisions. The remainder of the delay results from the Ethernet's fairness policies which effectively cause hosts to "wait their turn" to send a request. Although the authors point out that average transmission delay increases linearly with the number of hosts and does not increase dramatically at a particular load threshold, it is still significant. With three hosts generating continuous maximum-length packets, average transmission delays of about 3-4ms are experienced. Since the average NFS response time under a low to medium load is around 30 milliseconds, it is apparent that a 10% (i.e., noticeable) delay would be experienced if three NFS clients or servers were continuously sending read or write data. Normal NFS clients do not put this much load on the network. To saturate an Ethernet, a client would have to continuously send around 800 1536-byte packets/sec. We have observed that average NFS request rates tend to be around 1.5 NFS ops/sec/client, with less than 4 packets/op. This is an average rate of less than 6 packets/second. In bursts, however, a single client can effectively saturate the Ethernet (e.g., by issuing a stream of read requests that are satisfied from server memory). In our experience, when the Ethernet is continuously loaded at a level of more than 40%, or when there are sustained bursts in excess of a 60% load, noticeable delays are seen. Thus, if you observe such Ethernet loads (Sun's *traffic* can help to measure Ethernet load), it would be useful to try to identify those clients generating the heaviest NFS load. *nfsstat* can be used to measure NFS load. Since NFS read operations can account for 7 packets per request (as opposed to 2 for most other NFS operations), it is important to take the type of load into account. If a small number of clients are found to be responsible for generating large read loads, buying these particular clients a disk may be a cost-effective solution if the problem of distributing files to these disks appropriately can be solved.

Network Topology

Another factor contributing to excessive delay is network topology. If clients are located many hops away from the servers that they use heavily, their requests will experience long delays. Restructuring the network topology to better distribute load can solve the problem.

Retransmissions

Excessive retransmissions can cause poor performance because the client must wait a long time for the server to respond before it can retransmit a request. Causes of excessive retransmissions include: overloaded servers which drop packets due to insufficient buffering; inadequate Ethernet transceivers which cause packets to be dropped under bursty conditions; physical network errors (such as noisy coaxial cable); and software bugs (such as broadcast storms caused by incorrect broadcast addresses). The *nfsstat* utility can be used to measure the NFS retransmission rate. We can calculate when the rate of retransmissions becomes unacceptable. Average NFS response time to an 8 KB client read request under a low to medium load is around 30 milliseconds. Most clients retransmit after 0.7 seconds. If we assume that the user can feel a 10% reduction in performance then a 3 millisecond reduction is the tolerable limit. This gives an acceptable NFS retransmission rate of 0.42%:

$$\frac{.003\ sec/call}{0.7\ sec/retransmission} = 0.0042\ retransmissions/call$$

Since an 8 KB NFS read request requires 7 packets (1 request, 6 fragmented replies), the error rate of the network must be less than 0.06%:

$$\frac{0.42\%}{7} = 0.06\%$$

The acceptable retransmission rate will vary with request size and frequency, but it is clear that even a small retransmission rate can have a big impact on performance.

The causes of a high retransmission rate are often easy to fix. The first place to look is to see if an individual machine is experiencing more than a few network errors. The *netstat* utility can be used to measure the network error rate. If the problem seems pervasive, analysis of the cabling technology being used may isolate the difficulty. The operating system itself may be incorrectly tuned for optimum network performance. With SunOS 4.0, we found high retransmission rates due to dropped packets. The cause turned out to be that too few Ethernet buffers were allocated to the Ethernet driver. This in turn resulted in a high percentage of Ethernet input errors when packets arrived with no place to go. To fix this problem, we wrote programs to monitor the Ethernet driver for the problem, and to patch the kernel to increase the number of buffers (see Appendix).

Client Bottlenecks

Adding disk or memory to the client can improve performance in two ways: by improving access time and by reducing overall load on the server and network. A client can avoid NFS performance problems for files that are not shared (e.g. swap and temporary files) by using local disks for these files. As disks become smaller, quieter and cheaper, this is becoming an acceptable alternative. Once these disks are used to hold valuable data however, the problems of administering them (e.g., backup and restore) and sharing the data they contain, become more of a problem. In addition, the cost of configuring in the UNIX file system (ufs) code on a client adds about 120 KB to a Sun-3 kernel, which reduces the amount of memory available for other purposes. On diskless clients, more memory can make a big improvement in swapping performance by allowing the client to swap less often. By adding local resources, the demands on the server and the network may be reduced, yielding better system throughput.

While it is easy to improve client performance by adding memory or disk, there are no simple rules of thumb to guide you when determining how cost-effective these improvements are. However, it is easy to see that resources added to the server are more cost effective (because a single instance of a resource can be shared by many) and are easier to administer. Thus, the best place to start when trying to improve performance is with the server.

Server Bottlenecks

The server Ethernet interface may be implemented with old technology (such as 16-bit data paths) which causes decreased throughput. The server CPU could be a bottleneck, although in an environment with a reasonably powerful server (>= 2 MIP machines), this is probably not the case. Processors of this class or above are usually able to keep up with the rate of NFS requests that the Ethernet can deliver. Slower processors (< 1 MIP machines) are not able to keep up with the Ethernet, and even moderate network loads on a slow server could swamp its CPU. Many users who experience NFS performance problems with Sun-3's are surprised that upgrading to a Sun-4 gives no better NFS performance. This is because CPU performance was not the problem. Sun's *perfmeter* utility may be used to measure CPU utilization.

On most NFS servers the limiting factor is the speed of the disk. Most high-speed disks today can sustain from 30 to 40 disk operations per second. Most of the time spent waiting for a disk operation is in seeking or rotational delay. Using a faster disk or disk controller, and spreading the load over multiple disks may help, but this usually only provides an incremental improvement. The best way to improve disk performance is to reduce the number of disk operations.

NFS Server Performance in UNIX

UNIX uses a buffer cache to avoid disk operations whenever possible. The buffer cache is very effective in reducing the time that clients wait for slow disk I/O. It also makes disk I/O more efficient by allowing staging and scheduling of disk operations. A performance gain is made by allowing the disk device driver to schedule several requests at a time to take advantage of the position of the disk arm. Also, the total amount of disk I/O is reduced, since repeat requests may be found in the cache.

The buffer cache on NFS servers is usually very efficient for reads. Hit rates greater than 80% are normal, and hit rates as high as 98% are not uncommon. If the buffer cache hit rate on your server is low, adding more memory to increase the size of the cache should improve NFS read performance. With most current UNIX implementations, buffer cache sizes can be increased by changing system parameters. In SunOS 4.0, adding memory directly increases the buffer cache size since all of memory is effectively a cache.

NFS Bottlenecks

The nature of NFS itself causes performance bottlenecks at the server. To see how these bottlenecks occur, we must first discuss how NFS works.

NFS uses a simple stateless protocol. This protocol requires that each client request be complete and self-contained, and that the server completely process each request before sending an acknowledgement back to the client. If the server crashes, or an acknowledgement is lost, the client will retransmit its request to the server. As a consequence of this policy, the server cannot acknowledge the client's request until data is safely written to non-volatile storage. In this way, the client machine knows exactly how much modified data has been safely stored by the server. This means that the server cannot store modified data in volatile storage (storage that can be lost when the server crashes) to avoid disk writes, because if it did store and acknowledge it and subsequently crash, the acknowledgement would be a lie. The client would assume that its data was safely stored, and would be in for a rude surprise when it is eventually discovered that its data was not available. In fact, the client may not discover that anything went wrong until long after the crash occurs (e.g., if a page that was swapped out is swapped in with different contents). Since a single server stores data for many clients, this surprise could make life painful for an entire user community. By always writing modifications directly through to the disk, NFS ensures that server crashes do not cause data to be lost and crash recovery is trivial.

In addition to writes, other NFS operations cannot be cached to avoid disk writes. Operations that modify the file system in any way: file creation, file removal, attribute modification, etc. add significantly to the amount of filesystem data that the server must write synchronously to disk before responding. For example, when the client creates a new file, the server must update the data and inode blocks for the directory, and the inode block for the file.

This very desirable property of crash-survivability causes performance problems because of the fact that many NFS operations must be synchronously committed to disk. First, these operations take place at disk speeds, not the memory speeds available to cachable operations. Second, since these operations are processed serially, there is no opportunity to optimize the scheduling of the disk arm or the rotation of the disk. Finally, since modifications to the UNIX cache are written through synchronously to disk, there is no opportunity to decrease write disk traffic with cache hits.

If you have already added memory to your server to increase the size of the buffer cache and the server is still too slow, one possible solution is to purchase another server and split the load between the two servers. Not only does this have a large direct cost, there is a significant administrative cost associated with supporting this additional server. We will next examine an alternative solution which gives analogous performance without requiring a new server and without added administrative overhead.

Write Caching to Boost NFS Performance

Unless the server is only supplying read-only access to files (i.e., all modified files, including client swap files, are maintained locally), it is required to synchronously commit some NFS operations to its disks. Since disks are several orders of magnitude slower than memory, this is a tremendous burden. If disk writes could be safely cached in non-volatile memory, significant performance gains could be made.

There are four basic reasons why reliable write caching can boost NFS server performance.

1. Faster Transfer Time

An incoming NFS write request can be secured safely in non-volatile memory and then acknowledged to the client. Because the time to transfer data to memory is far less than the time to transfer data to disk, the turnaround time for NFS update request handling is reduced.

2. Repeat Cache Hits

A single NFS write operation causes two or three writes to the disk. This is because each server write must update not only the data block but the inode and (often) an indirect block as well. Since the same inode is updated for each data block in the file, many disk writes can be saved by rewriting the cached inode. Data blocks may also be found in the cache, although the frequency of these cache hits is significantly less than the hits on the inodes. If data blocks are reclaimed (e.g., as a result of removing a file) before they are flushed, it is possible that a file can be created, written and removed with no disk I/O involved at all. By lessening disk traffic, overall system performance is enhanced.

3. Improved Disk Scheduling

Since the non-volatile memory will continue to retain data until the disk acknowledges that the data has been written, data may be flushed asynchronously from the non-volatile memory. This allows multiple blocks of data to be scheduled together to take advantage of the location of the disk arm. As disk seek times and rotational delays are significant (tens of milliseconds), this represents a major economy.

4. Importance of Update Operations in NFS Traffic

Because UNIX read caching is already very effective, NFS operations that modify file data account for a disproportionately large amount of actual disk traffic. In an environment where NFS operations that modify data comprise about 20% of the operation mix, over 60% of the requests for disk I/O are due to these operations. This is measured by code in the device driver for our write cache which is instrumented to provide statistics on disk access.

As a result of these factors, write caching can greatly increase server performance. In addition to accelerating NFS servers, a write cache can speed up any application requiring synchronous writes to disk. This includes local synchronous operations such as file and directory creation and removal. Many database or transaction systems require local synchronous writes of files and can potentially show significant performance improvements with write caching.

Write Cache Implementation

Prestoserve is Legato's filesystem write cache. It consists of non-volatile memory and a device driver. Non-volatile memory is required since data must not be lost when power fails or the system crashes. We use low-power, highly reliable static RAM, and a triply-redundant set of lithium batteries which can keep data valid for over two years without external power. The device driver intercepts requests for the disk by inserting its own routines into the *bdevsw* and *cdevsw* tables where the original device entries were. The original entries are maintained in a table used by the device driver. In addition, the *reboot*(2) system call is intercepted, so that the driver can flush its buffers when the system is shut down cleanly. Because the write cache device driver looks like a normal disk device driver to the rest of the kernel, all UNIX semantics are maintained. In our current implementation, all disk controllers use the same write cache.

The administrator can select which filesystem will use the write cache (by default, all writable filesystems are selected). When a synchronous write request is issued to an accelerated disk, it is intercepted by the device driver which stores the data in non-volatile memory instead of on disk. Thus, synchronous writes occur at memory speeds. As the cache fills up, it asynchronously flushes cached data to the disk in groups large enough to allow the disk drivers to optimize the order of the writes.

From the point of view of the operating system, the write cache simply makes existing disk drivers faster. Since it is implemented as a device driver and maintains its own data structures in stable storage, the cache is relatively invulnerable to system crashes which are caused by software bugs, administrator error, power failures or hardware failures.

Reliability Considerations

Because it interacts with the server's disk data, it is imperative that all error situations be properly handled by the write cache device driver. The basic axiom followed in the design of the write cache is to preserve data integrity at all times. Some situations that must be handled correctly include:

- orderly shutdown
- system panics
- power failures
- removable disk packs
- disk failures
- attempts to use a cache with dirty data on a different system

To ensure that these situations are properly managed, we take the following precautions. Should the cache ever be disabled or removed for any reason, all data is passed directly through to the disk. Should it prove impossible to flush a block to disk, the block remains in the cache until the disk can be repaired. When the system is shut down normally (e.g., via *shutdown*, *reboot*, or *halt*), all cached data is flushed to disk. This is done so that normal board swapping activities have no potential for causing data to be lost. The write cache should be viewed just like a disk: it is possible to lose data if you really try. By pulling the plug on a running system, removing the board, and turning off the batteries, you can lose data. This is analogous to opening the top of a disk drive and de-magnetizing the surface of the disk. To prevent a board containing valid data from being inadvertently moved into a different machine, the cache maintains the host id of the system it is installed on in non-volatile memory. If a board with valid data is moved to a foreign host, the administrator will be asked upon reboot whether the data in the board should really be discarded or not. When a removable disk pack is removed, it must first be unmounted. As a result, the device close routine is invoked. Our device close routine is interposed between the real device close routine and the rest of the kernel, so we ensure that its invocation causes write cache contents for that device to be flushed. Thus, the contents of the removable disk pack are consistent.

Performance Results

Any filesystem not mounted read-only will benefit from write caching. The implementation required much experimentation to determine the correct cache size and flushing algorithms to employ. The 1MB cache can hold 1-2 seconds of Ethernet traffic at maximum speed, adequate time to flush the entire cache to disk if needed. Empirical results also suggest that the cost/benefit of a larger cache is unwarranted. However, when faster networks (such as FDDI) are employed, a larger cache can be of value. A modified form of Least Recently Used (LRU) replacement is used to efficiently flush the cache and keep enough free buffers available to handle bursty traffic. Care has been taken to ensure that the cache is also used effectively to assist the UNIX buffer cache when appropriate for asynchronous requests. For example, read requests not satisfied by the UNIX buffer cache may be satisfied from the write cache, and blocks repeatedly updated asynchronously (e.g., superblock, cylinder groups) may be written to the write cache instead of to disk.

Our investigations show that users have an acute awareness of poor NFS performance, although some would prefer that response time (the time to process an individual request) be improved while others express a preference for throughput (the number of clients a server can support). To objectively measure response time and throughput, we have devised a benchmark called *nhfsstone* which places an artificial load on an NFS server based on NFS operation mixes obtained from working systems.

The *nfsstat* utility provided with the Sun NFS reference port can be used to determine the mix of NFS operations in a particular environment, and this mix is used to drive the *nhfsstone* benchmark. To chart response time, the turnaround time for handling a given load is measured. Throughput is the inverse: how many NFS operations per second can be handled at a given level of response time. It is important to use a benchmark that simulates a realistic load on the server to avoid getting a distorted view of performance. For example, measuring the time to copy a large file over NFS on a server with write caching shows at least a 300% improvement. This is somewhat misleading, since file copying does not dominate the workload of a typical system. However, it does illustrate that some applications may be accelerated more than others.

Nhfsstone Results

To evaluate performance, we used *nhfsstone* to generate a reproducible load on a server. The mix of NFS operations used (note that 19% of these operations modify data) follows in figure 1. Operations representing less than 0.5% of the mix are not included.

% in Mix	*NFS Operation*
13	getattr
1	setattr
34	lookup
8	readlink
22	read
15	write
2	create
1	remove
3	readdir
1	fsstat

Figure 1: *Typical Diskless NFS Operation Mix*

Results vary depending upon the configuration of client, server and disk being used. Since *nhfsstone* was designed to run on a single client, it is important to use a client powerful enough to generate significant loads. For the results given here we used a Sun-3/470 client machine and a Sun-4/260 server running SunOS 4.0.3. The server had 24MB memory, a Xylogics 7053 controller, and a CDC 9720-850 disk. *nhfsstone* divided its load between two disk partitions to simulate normal use (e.g., a */home* and a */export* partition). The results with this mix at varying loads are shown in figure 2.

We observe that with write caching, response time stays acceptable, even as the load climbs over 100 NFS operations per second. Without write caching the server cannot process loads exceeding 70 NFS operations per second. We have also noticed that with write caching, CPU utilization at high loads begins to approach 100%. Servers without write caching begin thrashing before utilizing 100% of the CPU. Thus, a faster CPU will not make much difference in improving NFS performance until the I/O bottleneck at the disk is reduced. *nhfsstone* currently measures a single client loading a server. With write caching the server can comfortably handle more load than one client can generate, so we are unable to report on server behavior at loads much beyond 120 NFS operations/second.

The numbers we have reported reflect an NFS operation mix in which about 20% of the operations modify data. This corresponds to a fairly typical diskless NFS environment. If a response time of 40 ms/operation is acceptable, then a server without write caching can support about 40 operations per second. The same server with write caching can support about 110 operations per second. This translates into nearly a 200% increase in the number of clients that can be supported at this level of responsiveness. If local disks are used for temporary files and swapping, the percentage of NFS operations that modify data typically drops to about 10%.

Load(ops/sec)	*Non-cached Response* Time (ms/op)	*Cached Response* Time (ms/op)
10	11	9
20	19	10
30	27	11
40	39	15
50	54	17
60	71	19
70	93	22
80	infinite	26
90	infinite	32
100	infinite	35
110	infinite	40
120	infinite	46

Figure 2: *Results at Varying NFS Operation Loads*

In PC NFS, clients typically write directly to a file in 512-byte chunks. This translates into an 8 KB write requiring not 1 8 KB synchronous filesystem write but 16 512-byte synchronous filesystem writes on the server. As each of these writes requires an inode (and possibly an indirect block as well) to be written synchronously, at least 32 and as many as 48 synchronous disk write operations will be required. With write caching, an average of one asynchronous disk write per 8 KB of file will be generated. Thus, PC NFS benefits from write caching can be impressive.

To really understand how an individual site will benefit from write caching, each site will have to measure its own mix of NFS operations. The higher the percentage of operations that modify data, the more beneficial write caching will be. Also, the higher the load on the server, the more noticeable the improvement will be. To get a feel for how different mixes of NFS operations affect the throughput increase from write caching, we present some ranges we have measured in figure 3. We use "dataless" to indicate a client that uses a local disk for temporary files, root partition and swapping, and NFS for all other file access.

Clients	*Modify Mix*	*Throughput Increase with* Write Caching
All dataless	5%	10-40%
Mostly dataless	10%	30-80%
Diskless/dataless	15%	60-120%
All Diskless	20%	80-200%
PC/NFS	10%	200+%

Figure 3: *Effect of Different Mixes on Throughput*

Other Solutions to NFS Performance

There are several other methods that can be employed to achieve NFS performance benefits. Many of these methods have drawbacks or limitations that make them unacceptable for some users.

RAM disks are simply large amounts of memory (which may have battery back-up) with an interface that emulates a disk interface (SCSI for example). Unlike a RAM disk, a write cache serves as a front end to normal disks, so the benefits of fast memory are shared among all the disks on the server. A RAM disk has a finite capacity, and the expense of providing large amounts of storage is prohibitive.

Some vendors change the kernel NFS code to not do synchronous updates for NFS requests that modify filesystem data. However, this *dangerous mode* of NFS operation makes clients susceptible to mysterious losses of large amounts of data in the event of a server crash. It is also a violation of the NFS protocol specification. Some vendors use this method without warning their customers; it is always a good idea to check.

A new and better UNIX filesystem [Bra89] could improve filesystem write performance. Significant improvements from advanced techniques such as disk striping and the use of disk arrays [Kat88] may also accelerate filesystem write performance. These solutions can be expensive, and require major changes to the UNIX file system.

Attention to correct implementation and underlying protocol issues can yield significant NFS performance improvements (see [Now89], and [Jus89]). We expect to see these enhancements make their way into vendor's implementations of NFS in the near future. Other NFS modifications that address cache consistency, such as [Sri89] can reduce the frequency of update traffic (for example, setting the sticky bit on swap files tells the server that inode modifications need not be written synchronously as long as the size of the file remains constant and no new blocks are allocated). However, none of these enhancements completely eliminate the need for synchronous writes and the performance penalty of writing at disk speed. Because a server with a write cache can respond quickly to update requests, clients can issue new requests more quickly, generating more load on the server. The write cache is able to handle this increased load by immediately buffering incoming requests safely and flushing to disk at its leisure.

Summary

While many factors can contribute to poor NFS performance, server disk I/O is often the leading contributor. Write caching can improve NFS server performance by allowing the server to add memory to cache update requests safely, much as existing technology allows the addition of memory to improve the efficiency of read requests. By making caching available to operations that modify files, we can put back the performance NFS took away from UNIX.

Appendix

Legato has made publicly available some unsupported software (in source form) which is intended to help measure and improve NFS performance. To obtain this software, send electronic mail to *request@Legato.COM* or to *{sun,uunet}!legato!request* with a subject line: *send unsupported <product>*. Please include your return address by including it in the body of your message with a *path* command: *path <return address>*. A subject line of *index unsupported* will produce a list of all currently available software which includes:

nhfsstone	– NFS load generating and performance measuring benchmark
etherck	– Ethernet monitoring tool and kernel patch to increase available network buffers
netck	– Heuristic program to locate general network performance problems

References

[Bog88] David R. Boggs, Jeffrey C. Mogul, and Christopher A. Kent, "Measured Capacity of an Ethernet: Myths and Reality," in *SIGCOMM '88 Symposium on Communications Architectures and Protocols*, August, 1988.

[Bra89] Andrew Braunstein, Mark Reiley, and John Wilkes, "Improving the efficiency of UNIX file buffer caches," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December, 1989.

[Jus89] Chet Juszczak, "Improving the Performance and Correctness of an NFS Server," in *1989 Winter USENIX Technical Conference, San Diego*, 1989.

[Kat88] R.H. Katz, G.A. Gibson, and D.A. Patterson, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *ACM SIGMOD 88*, June 1988.

[Now89] Bill Nowicki, "Transport Issues in the Network File System," in *Computer Communication Review, Vol. 19 #2*, April, 1989.

[San85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, "Design and Implementation of the Sun Network Filesystem," in *USENIX Summer Conference Proceedings, Portland Oregon*, 1985.

[Sri89] V. Srinivasan and Jeffery C. Mogul, "Spritely NFS: Experiments with Cache-Consistency Protocols," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December, 1989.