Linux NFS Client Write Performance[1]

Chuck Lever, *Network Appliance, Incorporated*
<cel@netapp.com>

Peter Honeyman, *CITI, University of Michigan*
<honey@citi.umich.edu>

Abstract

We introduce a simple sequential write benchmark and use it to improve Linux NFS client write performance. We reduce the latency of the `write()` system call, improve SMP write performance, and reduce kernel CPU processing during sequential writes. Cached write throughput to NFS files improves by more than a factor of three.

1. Introduction

Network-attached storage (NAS) is an easy way to manage large amounts of data. Applications access data stored on NAS via various standard Internet protocols such as HTTP and NFS [6, 20, 21]. To make network-attached storage compete with locally attached storage requires that NAS provide equivalent performance. NAS server performance is well understood, but client performance has long been *terra incognita*.

As Linux servers proliferate within enterprise information infrastructures, performance of the Linux NFS client emerges as a factor critical to the success of complex applications such as database and mail services that use network-attached storage. Efficient access to shared data in laboratories that make extensive use of Linux workstations also depends on superior NFS client performance.

We are interested in two equally important goals. Our first goal is narrow: to improve the performance of the Linux NFS client. To do this, we also pursue a broader goal of identifying factors that influence NFS client performance.

To understand NFS client performance issues, we developed a simple file system benchmark that measures write latency and throughput. In this paper, we describe and rationalize such a benchmark, and use it to identify several means to improve application write performance to files accessed via the Linux NFS client. We also suggest ways to apply the benchmark and comparative techniques to client performance in general.

The remainder of this paper is organized as follows. In Section 2, we detail the development of the benchmark and identify issues that distinguish client from server performance benchmarking. In Section 3, we use this benchmark to expose and correct latencies in the Linux `write()` system call. In Section 4, we outline future areas of exploration and conclude the paper.

2. Measuring NFS client performance

In this section we develop a rationale for a simple sequential write benchmark based on Bonnie [1]. This benchmark was developed on specialized hardware (described in Section 3.1) that includes SMP Linux NFS clients connected to a prototype Network Appliance F85 filer via gigabit Ethernet.

2.1. Client performance issues

NFS is based on a "client makes right" design: the client is responsible for ordering bytes, managing network and server congestion, and otherwise handling the complex issues of implementing a distributed file system. This leaves the server simple and scalable [15]. In fact, NFS servers maintain very little state. Satyanarayanan, *et al.* [16] justify this architecture by pointing out that

[1] This document was written as part of the Linux Scalability Project at CITI, U-M. The work described here was supported via a grant from Network Appliance, Incorporated.

in typical client/server distributed systems, "workstations have cycles to burn." Consequently, an NFS client tends to be complex, which can interfere with efficiency and correct behavior.

Measuring NFS server performance is well understood. Computer science literature contains many examples of benchmarks meant to quantify NFS system performance or server performance [13, 17, 23]. SPEC SFS is a typical NFS server benchmark [19]. To remove client behavioral and performance variations from benchmark results, SPEC SFS uses its own user-space NFS client to access NFS servers under test.

Client performance measurement differs from server performance measurement. Generic file system benchmarks are biased towards exercising the weaknesses of disk storage, which is not terribly useful in divining the nature of a file system implementation that uses a network device as its back end. For example, `iozone`, a typical file system benchmark, tests both random and sequential read and write requests [13]. A client sends random write requests to a server as fast as it sends sequential ones. If performance differences exist between random and sequential NFS accesses, it is likely we are measuring server disk performance and not client behavior.

NFS client performance depends on the performance of networks and servers. It is problematic, however, to operate an NFS client without any server, thus it is difficult to isolate performance problems specific to a client. A slow server or network can cause application performance problems that are relatively easy to identify and fix; as we demonstrate, *faster* server performance can also degrade client performance.

A client's on-the-wire write request behavior sometimes affects server performance and scalability. Clients can modulate an application's unfortunate (random) access pattern to help servers scale better [7, 9]. The relationship between client and server must be carefully considered when dissecting client performance issues. In this paper, we focus only on a client's ability to get requests to the server. In later work, we may approach issues of server scalability that arise from client misbehavior.

One way to measure client performance is to eliminate performance bottlenecks from downstream components, using fast networking technologies and non-volatile RAM on the server, and to push the client as hard as possible to see what breaks. Just as SFS uses the same client to test different servers, a simple memory-based server could be developed to compare clients more fairly.

Another approach compares the performance and behavior of a single client under more typical workloads across a variety of networking conditions and server types. For both approaches, it is necessary to be wary of the bias of traditional file system benchmarking towards measuring disk behavior instead of other factors that are more important to client performance.

We borrow from both approaches in this study. Our hardware test bed consists of high-performance SMP Linux client hardware connected via a high-performance gigabit Ethernet switch to a prototype Network Appliance F85 filer. Also included in our test bed are a four CPU Linux server, and several single-CPU Solaris NFS clients (not used in this report). Comparing behavior and performance among these clients and servers exposes performance issues that might otherwise escape attention.

2.2. Related work

Little related work focuses specifically on NFS client performance. Improving NFS performance often amounts to helping the server use its disks more efficiently by improving client caching strategies, as in Dahlin, *et al.* [3]; or as in Juszczak, who adds write clustering to clients to help server scalability [7]; or by adding new features to the protocol, as in Macklem's Not Quite NFS [9].

Martin and Culler investigate NFS behavior on high performance networks, but do not address implementation specific issues in existing clients [10].

The closest previous work we found describes performance improvement (reduced CPU loading) through elimination of data copies in the 4.3BSD Reno NFS implementation [8].

2.3. Inter-run variance on Linux

Our experience with performance measurement on Linux has taught us to expect large variations in results between individual benchmark runs on the same O/S version and software and hardware configurations.

Other benchmarks performed by the authors in the past have revealed inexplicable variations in performance of several parts of the Linux kernel, including the virtual memory subsystem, the scheduler, and parts of the system whose correctness depend on the global kernel lock. There are often one or more outlying data points that skew average results, often masking relevant behavior. Such variations are not common in commercial operating systems such as Solaris. The best

results on Linux are excellent, but they are too often hampered by the outliers, leaving only moderate to good performance on average. Several measurements reported here illustrate this phenomenon.

To make forward progress we must often ignore these variations. Over time, our experience resolves many of these issues, but one could wish that untuned system behavior were more consistent.

To address this, we generally report single run results in this paper. The "shape" of the results is typically consistent from run to run, including any highly variable outlying results. We are most interested in trends rather than precise measurements, noting any anomalies.

2.4. Introducing our simple write benchmark

We started by measuring the Linux NFS client with Bonnie to understand several aspects of Linux client performance in combination, under a simple but typical load. We refined our benchmark to include only a small part of the suite of tests performed by Bonnie. In this section we discuss what was left out, and why.

Using a simple microbenchmark rather than a complex application simulation provides immediate and uncomplicated feedback without the additional effects of other application processing, improving the repeatability of results. It also offers a workload that drives specific components of a client with surgical precision. However, a microbenchmark does not offer a clear assessment of real world application performance impact.

We based our benchmark program on the block sequential write portion of the Bonnie file system benchmark. This test measures how quickly an application can write 8 KB chunks into a fresh file. Writing into a fresh file narrows our focus to write code pathways because the client does not read any preexisting file data from the server to complete write requests. Write throughput depends on the behavior of the kernel's VM, networking, and RPC layers, and offers a generic picture of file system performance. In addition, raw write performance is important to many typical real world workloads.

Both read and write operations are network-intensive because data is transmitted along with these requests. However, client O/S caching moderates the performance of application read requests on the client; writes reflect network efficiencies and latencies more directly [14]. Using sequential writes we minimize disk latency (*i.e.*, seek time) on typical disk-based servers. As pointed out in Section 2.1, we gain little new information about a client by comparing random and sequential results. We considered testing against a memory-only server, but we chose to start with a benchmark that does not require atypical server modifications. Thus we have a simple and typical application to run on a client that exercises many of the critical paths between client and server.

Bonnie includes the final `close()` call in elapsed time and throughput calculations to capture I/O that occurs after the last `write()`. However, for many local file systems, dirty data remains in the system's data cache after the final `close()` operation. To make fair comparisons between NFS (which always flushes completely before last close due to close-to-open semantics) and local file systems (which may delay flushing to improve perceived performance), our benchmark reports three throughput results: one for all writes, one for the subsequent flush operation, and one for the final close operation. Each result is a throughput measurement reported in megabytes per second (MBps), and is calculated by dividing the total number of bytes written by the amount of time from the beginning of the benchmark until just after the respective operation (writes, flush, close).

Our benchmark also reports system call latency. One can calculate throughput by dividing average system call latency into the average byte size of each request. Reducing system call latency has immediate positive effects on throughput. However, to get to the heart of system call misbehavior, it is sometimes necessary to record *actual*, and not *average* latency. As we demonstrate, jitter (variation in latency from one call to the next) drastically degrades data throughput in our test, and is easily revealed when examining actual results rather than computed averages.

3. Write latencies in the Linux NFS client

Here we report the results of our benchmark when run on an SMP Linux client against files on a Linux NFS server and a Network Appliance filer. Our goal is to identify and correct write performance problems.

The first section describes our software and hardware configuration, and the following sections report our measurements and describe our fixes. We finish with a description of recent improvements to the Linux NFS client resulting from our work.

3.1. Systems under test

In this section, we describe the systems used during these tests.

Client system: Our client software runs on a dual processor Pentium III system based on the ServerWorks III LE chipset. The processors are 933 MHz FC-PGA packages with 256 KB of level 2 cache. The front-side bus and SDRAM speed is 133 MHz. There is 256 MB of PC133 registered SDRAM in each system. The client has one 30GB IBM Deskstar 70GXP EIDE UDMA100 drive. Because of limitations in the ServerWorks south bridge, the IDE controller runs in multiword DMA mode 2. The ServerWorks chipset supports two 64-bit/66 MHz PCI slots; there is a Netgear GA 620T gigabit Ethernet NIC in one of these that supports 1000base-T (copper). The Netgear card uses the Alteon Tigon II chipset. This system runs a Linux 2.4.4 kernel with the Red Hat 7.1 distribution.

NetApp filer: The Network Appliance filer is a prototype F85 with eighteen 36 GB Seagate 336704LC SCSI drives. The F85 has a single 833 MHz FC-PGA Pentium III with 256 KB of level 2 cache, 256 MB of RAM, and 64 MB of NVRAM on a PCI card. The system supports several 64-bit/66 MHz PCI slots that contain a Q-Logic ISP 1280 SCSI controller and a fiber optic gigabit Ethernet card based on Intel's GbE chipset. Data stored on this system is contained in RAID 4 volumes. This system runs a pre-release of Network Appliance's DATA ONTAP operating system[2]. Special options enabled on the test volume include the `no_atime_update` option, which eliminates seek-intensive inode write activity during workloads that consist mostly of read requests. This option probably has no effect for our write-intensive workloads. The test volume contains eight disks in a single raid group. Snapshots are enabled during these tests.

Linux server: Our Linux NFS server is a four-way Intel system based on the i450NX mainboard. There are four 500 MHz Katmai Pentium III CPUs, each with 512 KB of level 2 cache. The front-side bus and SDRAM speeds are 133 MHz. The system contains 512 MB of SDRAM and six Seagate SCSI LVD drives of varying model, controlled by a Symbios 53c896 SCSI controller. The system is network-connected via a Netgear GA 620T 1000base-T Ethernet NIC installed in a 32-bit/33 MHz PCI slot. This system runs a Linux 2.4.4 kernel with the Red Hat 7.1 distribution. NFS files stored on this system reside on a single physical disk (no RAID). To maximize server write performance, we use the `async` export option; throughput results reported for the Linux server are therefore not comparable to a production server.

[2] Benchmark results produced on prototype hardware and software do not necessarily reflect the performance of any released product.

These systems are connected to a single Extreme Networks Summit7i Ethernet switch. The copper connections are made via CAT6 UTP cabling, and the fiber connection to the filer is standard multi-mode. Jumbo packets are not enabled on the switch or on any of the systems under test during these benchmarks. Unless otherwise mentioned, all network links are one Gbps, full duplex.

Both the Network Appliance filer and the Linux NFS server are mounted with typical mount options: NFS version 3 via UDP, rsize=wsize=8192. As of kernel 2.4.4, the Linux kernel NFS server does not support sizes larger than 8K. Using a smaller wsize causes the Linux client to use only synchronous network writes, resulting in a significant drop in write throughput. In addition, these sizes match the block size of our simple write benchmark.

The Network Lock Manager is disabled during our testing to reduce protocol overhead. Later we can test how much overhead is caused by Lock Manager interaction, after we quantify the baseline overhead for data transfer.

Using jumbo Ethernet frames is an easy optimization that can improve data throughput. However, jumbo frames work only for networks that allow large frame sizes from end to end, which makes them unsuitable in many situations. Because many realistic local and wide-area networks use smaller frames, it is useful to study the cost of packet fragmentation and reassembly. Thus, we chose to leave jumbo frames disabled during our tests.

3.2. Local versus network write performance

To begin, we compare the performance of sequential writes into a local file system (ext2fs [2] on the client) to the performance of sequential writes into a networked file system (NFS served from the filer and from the Linux NFS server). This compares the cache performance of ext2 against the cache performance of NFS *without regard to back end performance*. Ext2 cached write performance is a target for NFS client cached write performance.

This test calculates write throughput by dividing the total number of bytes written by the elapsed time required for all of the `write()` system calls to complete. Figure 1 shows throughput results that include only write calls, not including the final `flush()` and `close()` calls included. To allow a better comparison, the latter results are not included because ext2fs usually does not flush after `close()`.

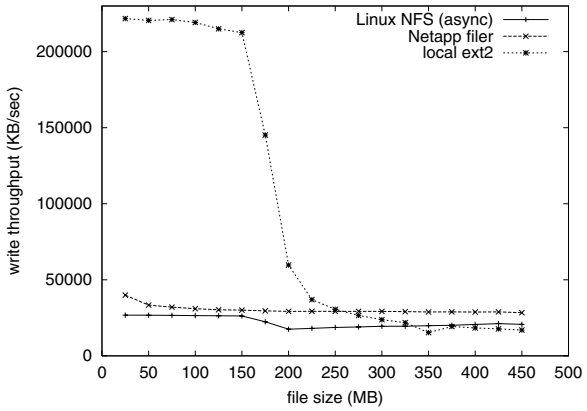

Figure 1. Local v. NFS cached write performance. *Write throughput is measured for test files between the sizes of 25 MB and 450 MB. Note that the large peak in memory write performance for local files does not appear for NFS files. NFS memory write throughput remains constrained to network/server throughput.*

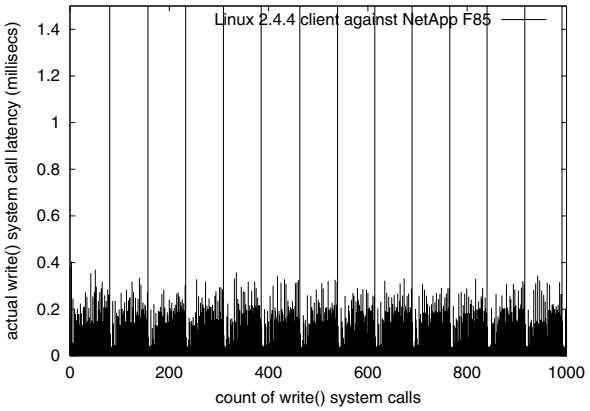

Figure 2. Write() system call latency. *This figure shows the first 1000 write system calls during a 40 MB benchmark run. Periodically, write system calls take more than 19 milliseconds, increasing the mean latency, thus decreasing overall throughput.*

Writes to local files are very fast while there is still memory available to cache dirty data. As the test file size approaches the size of client memory, performance drops to raw disk speeds. In contrast, the NFS client constrains write throughput even though there is ample memory available to cache writes. During the test, the application can generate data only as fast as the NFS server can take it, no matter how small the file is; little or no write buffering appears to occur on the client. In the next subsection, we explore this limitation.

3.3. Periodic latency spikes

Early in our testing we discovered that `write()` system call latency varies wildly but periodically. To explore `write()` system call latency, we execute our benchmark against a single 40 MB file residing on the Network Appliance filer, and report latency for `write()` system calls during the test. A typical result is shown in Figure 2.

While most writes complete within 300 microseconds, there is a periodic jump in latency approximately every 85 system calls. The latency for these slow system calls is over 19 milliseconds. While there are relatively few of these slow calls (37 out of 2560 calls in this run, or about 1.5%), they inflate the mean latency for the run from 139.6 microseconds per call (excluding the 37 calls exceeding 1 millisecond) to 482.1 microseconds per call, a factor of almost 3.5.

We observe similar results with both the Network Appliance filer and the Linux NFS server. The latency spikes do not appear in write requests on the wire.

Eliminating spiky latency behavior seems likely to lower average write latency and improve write throughput. We instrumented the Linux NFS client's write code path to record the time required for each step of a `write()` system call. We use the Linux kernel's `do_gettimeofday()` kernel function to capture wall clock time on either side of a target section of code, then record the timings in the kernel log.

We discovered several places where the Linux NFS client delays writer threads to keep memory usage in check. It delays writers when the number of pending write requests for an inode or mounted file system exceeds fixed limits. When the per-inode request count grows larger than `MAX_REQUEST_SOFT` (whose value is 192 in the 2.4.4 kernel) the NFS client forces the writer thread to schedule all pending writes for that inode and wait for their completion before resuming the current request. When the per-mount request count grows larger than `MAX_REQUEST_HARD` (whose value is 256 in the 2.4.4 kernel) the NFS client suspends any thread writing to that file system until another thread signals there are fewer than `MAX_REQUEST_HARD` requests. Each internal write request is no larger than a page. This implementation does not employ hysteresis to smooth the request load.

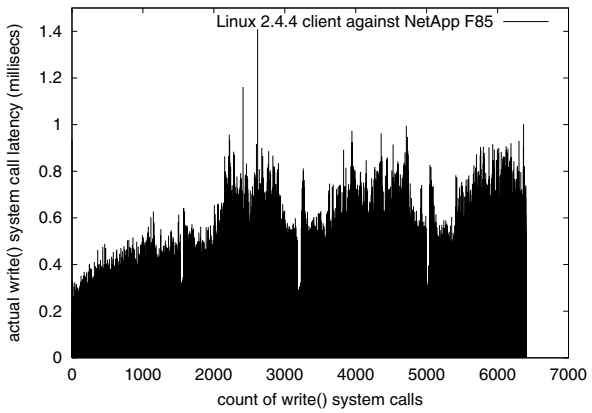

Figure 3. Write() system call latency without periodic flushes. *We show an entire benchmark run with a 100 MB file. The latency axis is the same as Figure 2. The periodic spikes in write system call latency are gone, but average latency grows worse over time.*

These limits prevent a large backlog of write requests. This is a classic time-space tradeoff. By limiting the amount of space available to buffer NFS writes the operating system avoids the expense of reclaiming pages at network and server latency speeds. It also avoids memory starvation if many write requests are pending when a server becomes unavailable.

Every system call in our test generates two write requests (8192 bytes is two pages, thus two requests). After the test application makes 90 `write()` calls, at least 180 internal requests are queued on the test file's inode. If the server is lagging, there may be requests from writes older than the past 90 system calls. Therefore, every 80 to 90 system calls, the client flushes the inode's write request queue. This produces the spiky latency seen in Figure 2.

In the Linux NFS client, a separate daemon, called `nfs_flushd`, flushes cached write requests behind a writing application. To minimize the cost of writes, the client should cache as many requests as it can in available memory [9]. The Solaris NFS client, for example, flush write requests only when the application requests it (via `fsync()` or `close()`), or unless the client cannot allocate more memory for new requests, in which case the VFS layer blocks the writer [4].

We see in Figure 3 that the periodic latency spikes are gone. However, mean latency does not improve: for the entire run (6400 writes in this case) the average latency is 484.7 microseconds. Furthermore, latency increases over time. We investigate this behavior in the next section.

3.4. List scans and sequential write performance

Scalability problems are often the result of lengthy data structure traversals. To establish whether data structure traversal limits throughput, we used a kernel-profiling tool that provides a sample-driven histogram of kernel execution to pinpoint areas of heavy CPU usage.

The profiler exposed two functions in the NFS client that consume significant CPU resources during the benchmark run: `nfs_find_request()` and `nfs_update_request()`, both of which use the inline function `_nfs_find_request()`. This helper function scans a sorted list of an inode's write requests to find a request that matches an application's current write request. The list is maintained in order of increasing page offset in the file.

Eliminating periodic write request flushing makes this per-inode list much longer. The sequential benchmark causes the client to traverse the list completely during each write system call, only to find no matching request, whereupon the client adds the new request to the end of the list.

To improve scalability, we implemented a hash table, similar to other hash tables in the Linux kernel, to manage the client's outstanding write requests. This hash table *supplements* the per-inode write request list. Finding a pending write request is now much faster, at a memory cost of eight bytes per request and eight bytes per inode, plus the size of the hash table itself.

The Linux VFS layer passes write requests no larger than a page to file systems, one at a time. Before the NFS client builds an RPC request, it maintains these page write requests on a per-inode list, ordered by page offset. Our modification inserts requests into a hash table based on the requesting inode and the page offset of the request.

All requests to the same page in the same inode are kept in the same hash bucket, so any overlapping requests are detected by searching all the requests in a single bucket. The client usually caches only a single write request per page to maintain write ordering, so this is normally not an issue. Write requests are coalesced into `wsize` chunks just before the client generates write RPCs.

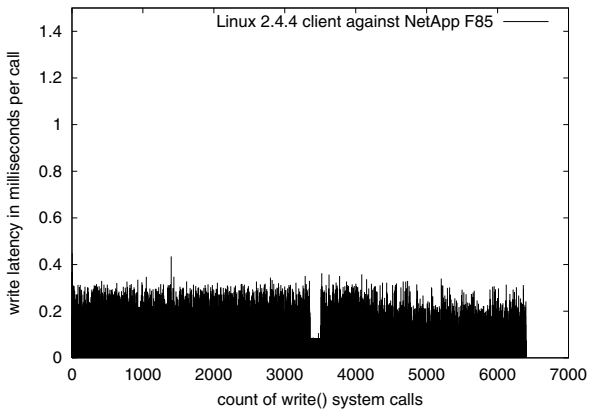

Figure 4. Write() system call latency with scalable data structures. *Write latency remains low even as the number of outstanding requests increases for the entirety of this benchmark run against a 100 MB file. For comparison, the latency axis is the same as in Figures 2 and 3.*

We see the improvement of using a hash table to track write requests in Figure 4. Write system call latency during this run averages 136.9 microseconds per call, about the same as the mean for the original 2.4.4 client when latency spikes are excluded (see Figure 2). The sustained memory throughput of our sequential write benchmark is now almost 115 MBps, compared to 28 MBps in Figure 1 for a 100 MB file.

We also notice a gap of greatly reduced jitter for a few hundred calls in the middle of Figure 4. This gap appears in several runs against the filer. We investigate this further in the next section.

3.5. Global kernel lock on SMP hardware

Having eliminated the extra flush in the write path, and implemented a scalable hash table to track write requests, we now compare write throughput performance of our client against a Network Appliance filer and against a four-way Linux NFS server.

During a typical run of our write benchmark with a 5 MB file, the filer sustains about 38 MBps of network throughput. Our benchmark reports it can generate about 115 MBps of writes. On the other hand, the Linux server can sustain only 26 MBps of network throughput, less than 70% of the filer's network throughput, yet our benchmark writes at a rate greater than 138 MBps 20% faster than the filer run.

To explore this unexpected behavior, we again examine write latency. Figure 5 shows a histogram of `write()` system call latencies. While some of these calls take less than 100 microseconds, many take longer. The distribution shows there are more slow calls when the file resides on the faster of the two servers.

Surprisingly, the client requires less overhead to buffer writes when it is sending data to a slow server. We verified this result with a server on 100 Mbps Ethernet. The benchmark writes to memory even faster with this server, which sustains less than 10 MBps per second of network throughput.

Kernel execution profiling shows that, during benchmark runs, the global kernel lock taken in `nfs_commit_write()` is under contention on SMP hardware. The lock section is the fourth largest CPU consumer in the kernel, exercised more than twice as often as the fifth largest consumer. A profile analysis of this section shows that the lock taken in `nfs_commit_write()` is the only contributor to CPU time sampled in the lock section.

On SMP hardware, even a single writer thread uses more than one CPU, because data that is not flushed during a `write()` system call is flushed later by the NFS client's write-behind daemon, `nfs_flushd`. Kernel lock contention results when both the single writer thread and the flush daemon generate network write requests. `Nfs_flushd` holds the global kernel lock whenever it is awake and flushing requests. We suspected the flush daemon was causing contention, but after removing the global kernel lock from the daemon, we found little improvement.

Next we instrumented the write path to find out where the most time is spent, and found that the kernel spends 50 microseconds per write request in the network layer (`sock_sendmsg()` is called from the RPC layer for each RPC request). This accounts for almost 90% of the time per request spent waiting in the NFS client's write path to acquire the kernel lock.

During the development of the Linux 2.3 kernel, the global kernel lock was removed from Linux's network implementation. Because it is now no longer necessary to hold the kernel lock while calling the network layer, we release the lock then reacquire it when `sock_sendmsg()` returns. This permits other writing processes to make progress while the network layer sends the current request.

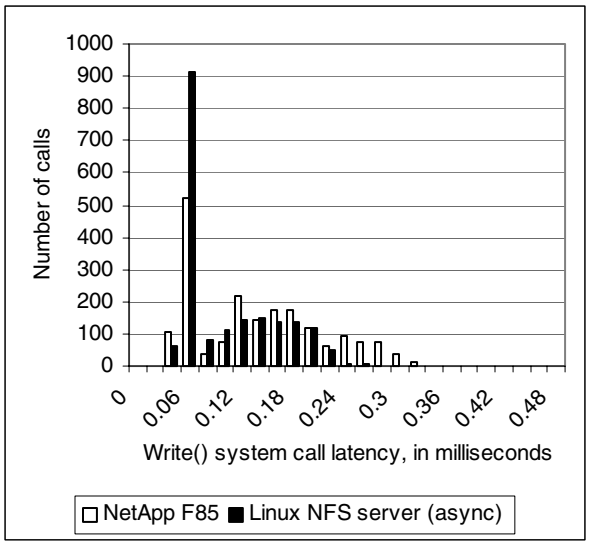

Figure 5. Write() system call latency against different servers. *This figure shows the latency of write calls during a benchmark run against a 30 MB file. Both runs have about the same minimum latency, but the filer run has a large number of lengthy calls. The* ***average*** *latency of client memory writes increases when a file is stored on a faster server.*

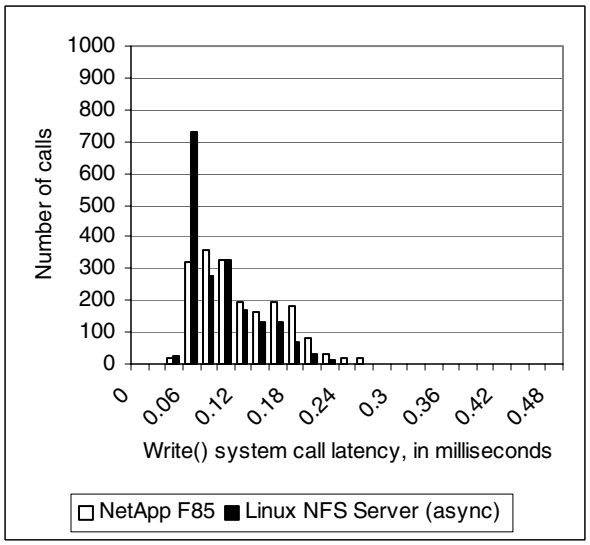

Figure 6. Write() system call latency with less lock contention. *This figure shows that maximum latency and latency variation (jitter) are clearly reduced. On average, filer writes still take longer than writes to the Linux NFS server, but the difference is much smaller. Minimum latency remains roughly the same, suggesting that latency variation, in this case, is the result of lock contention.*

Figure 6 shows the improvement in `write()` system call latency that occurs after removing the kernel lock around `sock_sendmsg()`. During this run, our calculated results also improve: the mean `write()` system call latency drops for both benchmark runs on the new client (127 versus 149 microseconds for the filer, 105 versus 113 microseconds for Linux), and the filer's maximum latency also drops, from 381 microseconds to 292 microseconds.

In calculating these averages, we excluded the first data point in all four runs. The latency for the first `write()` system call was almost a millisecond during two of the runs.

We note that the minimum latency hardly changes. This agrees with the idea that the latency variation is not a code path issue, but results from the writer waiting to acquire a resource, such as a lock.

We ran our 5 MB benchmark with the lock modification. Against the filer, the benchmark runs at almost 140 MB per second, better than a 20% increase over the earlier 115 MB per second. The benchmark runs at 147 MB per second against the Linux server, a 6.5% improvement. Lock contention measured by the profiler is almost entirely gone. These results are summarized in Table 1 (overleaf).

Although most of the unexpected inversion of performance is gone, the client still runs 5% faster against a server that is 40% slower. We discuss this further in the next section.

3.6. The cost of responding to server replies

Even though the Network Appliance filer provides better network throughput than the Linux NFS server, applications writing to the filer run slower. Despite the fact that less client processing is required for filer writes, which don't require an additional COMMIT RPC, client throughput to a fast server is hampered by lock contention and the cost of handling server replies at a higher rate. A faster server forces the client to do the same amount of work in less time. This explains the unexpected inversion of client performance.

	Network	*Lock*	*No lock*
NetApp F85	38 MBps	115 MBps	140 MBps
Linux (async)	26 MBps	138 MBps	147 MBps

Table 1. Application write throughput, before and after lock modification. *Network write throughput is compared to application write throughput when writing a 5MB file. Removing the global kernel lock from the RPC layer improves cached write throughput for files residing on both the Network Appliance filer and the Linux NFS server. Section 3.6 explains why applications can write faster to a slower server.*

Tests with a single application writer thread contending with a single flusher thread show less than ideal scaling. On a client with a single CPU, we expect to find the flusher thread taking some CPU time away from a user-level writer thread, increasing as server throughput increases. On a client with more than one CPU, however, the writer thread and the flusher thread should not interfere. We suspect that faster servers will exacerbate on SMP Linux clients until this issue is addressed.

Recall the short period in Figure 4 during which `write()` system call latency is much lower on average. This can now be explained by reduced SMP lock contention and interrupt load when the filer briefly stops responding to network write requests during a file system checkpoint [5]. In effect, the filer behaves like an infinitely slow server during this period, momentarily eliminating SMP lock contention on the client. While the flusher thread is blocked, only the application writer thread is active. Other threads do not compete with the writer, allowing the client to return control quickly to the application. In other words, the difference between the slowest and fastest writes in Figure 4 is due to the client's cost of responding to server replies.

Moreover, fast networking introduces significantly increased interrupt loads. The new network device driver API ("NAPI") in Linux 2.5 may help here, especially on single processor systems, by improving system behavior during intense interrupt loads that can result when a client is communicating with a high-performance server over a low latency network. When the system recognizes that a device is producing interrupts at a high rate, it masks the device interrupt and polls instead. As the workload decreases, the interrupt is re-enabled to keep I/O latency reasonable. This technique is further expanded by Mogul and Ramakrishnan [12].

In future work, we hope to explain why the network layer takes more than 50 microseconds per RPC request on a 933 MHz processor. We suspect IP fragmentation is a major expense. Jumbo frames, a feature of gigabit Ethernet, may help by reducing the need for fragmenting and reassembling large RPC requests in the IP layer, although this does not extend to WANs, in general.

Removing the global kernel lock from the write path yields considerable improvements in throughput and application concurrency. As it happens, the RPC layer also acquires the global kernel lock to ensure the integrity of its internal data structures. Removing the global kernel lock from the RPC layer will allow an SMP system with multiple network interfaces to process more than one RPC request at a time, allowing concurrent writes to separate files and to separate servers from separate client CPUs.

3.7. Final measurement

Figure 7 illustrates how our modifications have improved client write performance. With our modifications, NFS write performance is very good while memory is available to buffer write requests, but drops to the server's throughput rate when the client exhausts memory.

The left side of Figure 7 shows that memory write performance to NFS files is considerably improved. Write performance is no longer limited to network and server speeds. Client scalability defects continue to cause memory writes to files on the Network Appliance filer to be 7 MBps slower than to files on the Linux NFS server. The right side of Figure 7 shows that as client memory is exhausted, the filer sustains greater network write throughput than the Linux NFS server can.

NFS write performance is still not as good as writes to local files, however. We believe this is due to the costs on the client of responding to the server's replies. These costs, which include interrupt handling and network processing, are clearly greater than simply managing disk I/O.

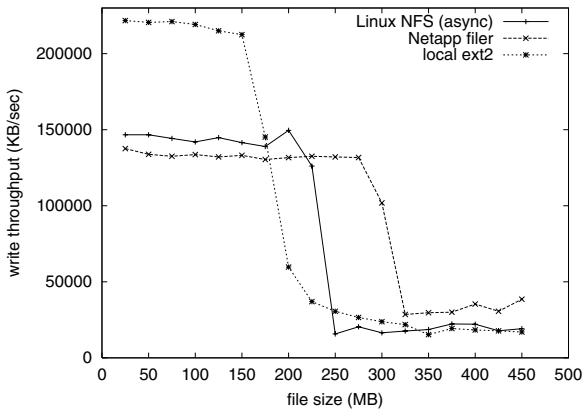

Figure 7. Local v. NFS cached write performance, revisited. *This figure shows write throughput for test files between the sizes of 25 MB and 450 MB. NFS write throughput is considerably improved compared to Figure 1. Application write throughput no longer tracks network write throughput for small NFS files. Maximum cached write throughput is nearly the same against both servers.*

Throughput for the local test and the test against the Linux NFS server immediately trail off for file sizes that exceed the physical memory size of the client, but the benchmark is able to sustain high data throughput longer when the test file resides on the Network Appliance filer. We conjecture that the filer's NVRAM acts as an extension of the client's page cache, allowing writes to the server to proceed at near local memory speed until the server's NVRAM is full. The fact that the filer is able to process requests faster also makes more client buffers available for a little while.

With workloads that hold a file open for a long time and write asynchronously (that is, without the requirement that data be made permanent before the `write()` system call is complete), the slower Linux NFS server has a slight advantage. Where applications write then immediately flush or close, or where applications require data permanence before a `write()` system call returns (*e.g.* databases), the Network Appliance filer, with its greater network and disk throughput, performs better. Though cached writes are slightly slower on the client, applications regain control sooner after they flush or close a file when writing to a faster server. As client scalability improves, applications can take advantage of improved memory write throughput *and* better network throughput.

3.8. Recent releases of the Linux NFS client

After writing the initial drafts of this paper, we shared our work with Trond Myklebust, the maintainer of the Linux NFS client. Trond built on our ideas, creating a safe version of our patch to remove the global kernel lock from the RPC layer and the NFS client's write path. This patch is available on his web site in the experimental patches section [22].

Trond also made a simple change to the write request queuing logic to reverse the order of the list, based on the results of our hash table experiment, to allow sequential writes to insert new requests into the request list in constant time, rather than walking the entire list. Finally, based on this paper and other recent work at Network Appliance, he replaced the flushing logic described in Section 3 with an entirely new system. This work now appears in Linux kernel releases following 2.4.15. Because so many other changes have occurred since the 2.4.4 kernel, a direct comparison is not meaningful. However, we hope to analyze some of these improvements in future work.

4. Discussion and future work

In this paper, we describe a simple sequential write benchmark to measure file system write latency and throughput. We show how this benchmark reveals performance and scalability problems in the Linux NFS client, and we describe several modifications to the Linux NFS client that improve application write latency and throughput.

4.1. Observations on client benchmarking

An NFS server's job is to store data and metadata in an organized way, and to move data between network cards and disks as efficiently as possible. Measuring these behaviors is well understood. On the other hand, a client's role is to translate and adapt the NFS protocol to its local environment efficiently. This is a much more subtle task.

Because a client is complex and completely dependent on the performance of servers and their disks, we use a microbenchmark, rather than a large suite of tests, to focus analysis on small parts of client behavior. As a result of our studies, we have identified several areas where client implementation directly affects application throughput. Some of these areas are already documented by previous work.

Networking efficiency

Packet fragmentation and reassembly, handling packet loss, eliminating data copies, handling heavy interrupt loads, and optimizing the number of network requests all contribute to the cost of handling server replies.

Caching efficiency

Effective caching makes NFS clients perform almost as well as local file systems. This means making the best use of available memory, as well as properly implementing cache coherency.

I/O scheduling

Unfortunate write scheduling can decimate application performance and server scalability.

Lock contention

With any number of CPUs, avoiding lock contention is critical. This has direct bearing on how well client performance scales when adding more CPUs and network interfaces.

Data structure efficiency

As the power of clients and the amount of cached data grows, it is vital to manage both efficiently.

Our current efforts focus on developing a suite of microbenchmarks of these aspects, in the style of McVoy's lmbench [11].

4.2. Future work

In this paper, we identified several specific issues with the Linux client that deserve further investigation. As our work continues, we hope to evolve benchmarks that measure each of these areas.

We want to assess further the impact of the global kernel lock on the scalability of the Linux NFS client. We also want to continue investigating why slower servers allow faster memory write throughput on Linux NFS clients, and why, in general, there continues to be so much variance between benchmark runs on Linux.

We especially want to prove our comparative methodology within real application domains. To keep our study on point we have focused mainly on our microbenchmark; future work will determine the real world impact of these changes. These techniques are also valuable for surveying NFSv4 client implementations [18]. Finally, we hope to explore improvements to the Linux NFS client that affect its behavior in corner cases that face advanced deployments outside the research lab, such as its file locking and specialized caching behavior, and its performance with databases and massively parallel applications combined with network-attached storage.

Acknowledgements

The authors gratefully acknowledge the assistance of our colleagues Andy Adamson, Kendrick Smith, James Newsome, and Steve Molloy at the University of Michigan; Brian Pawlowski and Sudheer Miryala at Network Appliance; Spencer Shepler and Sun Microsystems; and especially Trond Myklebust for his helpfulness and thorough work on the Linux NFS client. Special thanks also to FreeNIX reviewers and to our shepherd, Jim McGinness. The Intel Corporation loaned CITI hardware used in this study.

The source for our simple write benchmark and a patch against Linux kernel 2.4.4 that includes the modifications discussed in this paper are available at the CITI web site:

http://www.citi.umich.edu/projects/nfs-perf/patches/

References

1. Bray, T. Bonnie Source Code. Netnews Posting, 1990.

2. Card R., Ts'o, T., Tweedie, S. "Design and Implementation of the Second Extended Filesystem." *Proceedings of the First Dutch International Symposium on Linux*, December 1994.

3. Dahlin, M., Mather, C., Want, R., Anderson, T., Patterson, D. "A quantitative analysis of cache policies for scalable network file systems." *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

4. Eisler, Michael. Personal communication, September 2001.

5. Hitz, D., Lau, J., and Malcolm, M. "File System Design for an NFS File Server Appliance." *USENIX Technical Conference Proceedings*, January 1994.

6. "HTTP: A protocol for networked information." W3C, 1992. www.w3.org/Protocols/HTTP/HTTP2.html

7. Juszczak, C. "Improving the Write Performance of an NFS Server." *USENIX Technical Conference Proceedings*, January 1994.

8. Macklem, R. "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol." *USENIX Technical Conference Proceedings*, January 1991.

9. Macklem, R. “Not Quite NFS, Soft Cache Consistency for NFS.” *USENIX Technical Conference Proceedings*, January 1994.

10. Martin, R., Culler, D. “NFS Sensitivity to High Performance Networks.” *SIGMETRICS '99/PERFORMANCE '99 Joint International Conference on Measurement and Modeling of Computer Systems*, May 1999.

11. McVoy, L., Staelin, C. “lmbench: Portable Tools for Performance Analysis.” *USENIX Technical Conference Proceedings*, June 1996.

12. Mogul, J., Ramakrishnan, K. “Eliminating Receive Livelock in an Interrupt-driven Kernel.” *USENIX Technical Conference Proceedings*, January 1996.

13. Norcott, W., *et al*. IOzone benchmark. See www.iozone.org .

14. Ousterhout, J. and Douglis, F. “Beating the I/O Bottleneck: A Case for Log-Structured File Systems.” *Proceedings of the ACM Symposium on Operating System Principles*, 23, January 1989.

15. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D. “NFS Version 3 - Design and Implementation.” *USENIX Technical Conference Proceedings*, June 1994.

16. Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A., and West, M. “The ITC Distributed File System: Principles and Design.” *Proceedings of the 10th ACM Symposium on Operating System Principles*, December 1985.

17. Shein, B., Callahan, M., Woodbuy, P. “NFSStone A network file server performance benchmark.” *USENIX Technical Conference Proceedings*, June 1989.

18. Shepler, Spencer, *et al*. “RFC 3010 – NFS Version 4 Protocol specification.” IETF draft standard, December 2000.

19. Standard Performance Evaluation Corporation. SPEC SFS97. www.spec.org/osg/sfs97/ .

20. Sun Microsystems, Inc. “RFC 1094 - NFS: Network File System Protocol specification.” IETF Network Working Group. March 1989.

21. Sun Microsystems, Inc. “RFC 1813 - NFS: Network File System Version 3 Protocol Specification.” IETF Network Working Group. June 1995.

22. Myklebust, Trond. Linux NFS client pages, 2002. See www.fys.uio.no/~trondmy/src/ .

23. Wittle, M., Bruce, K. “LADDIS: The Next Generation in NFS File Server Benchmarking.” *USENIX Technical Conference Proceedings*, June 1993.