CITI Technical Report 01-12

Linux NFS Client Write Performance

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 the Linux NFS client's write performance. We reduce the latency of the `write()` *system call, improve SMP write performance, and reduce kernel CPU processing during sequential writes. Memory write throughput to NFS files improves by more than a factor of three.*

October 2, 2001

Center for Information Technology Integration
University of Michigan
535 West William Street
Ann Arbor, MI 48103-4943

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. For more information, see our home page.

If you have comments or suggestions, email <nfs-perf@citi.umich.edu> .

Linux NFS Client Write Performance

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

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

1. Introduction

As Linux becomes a permanent fixture of many corporate infrastructures, the performance of its Network File System client emerges as critical to the success of complex corporate applications such as database and mail services [4, 5]. Efficient access to shared data in laboratories that make extensive use of Linux workstations also depends on good NFS client performance.

To understand NFS client performance issues, we developed a simple file system benchmark that measures write latency and throughput. Our interest is not simply to identify specific problems in the Linux client, but also to understand general challenges to NFS client performance measurement. In this paper, we describe the benchmark and use it to identify several opportunities to improve application write performance to files stored in NFS.

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. Benchmarking NFS clients

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 later in this report) that includes SMP Linux NFS clients connected to a prototype Network Appliance F85 filer via gigabit Ethernet.

2.1. Client versus Server benchmarking

NFS is 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 [7, 8]. Satyanarayanan, *et al.* [5] justifies this architecture by pointing out that in typical client/server distributed systems, “workstations have cycles to burn.” Consequently, an NFS client tends to be complex, which interferes with performance and correct behavior.

Benchmarking NFS servers is fairly well understood. A typical NFS server benchmark is SPEC SFS [6]. 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. Likewise, NFS client performance depends on the performance of networks and servers, but it is problematic to operate an NFS client without any server. A slow server or network can cause application performance problems. As we demonstrate, faster server performance can also hamper client performance on naïve benchmarks. The relationship between client and server must be carefully considered when dissecting client performance issues.

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. Another approach compares client performance and behavior under more typical workloads across a variety of networking conditions and server types.

We use 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. Comparing behavior and performance among these clients and servers exposes performance issues that might otherwise escape attention.

2.2. Benchmarking on Linux

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

Other benchmarks run 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 benchmark 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 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.

2.3. 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 then refined our benchmark to include only a small part of the suite of tests performed by Bonnie.

The benchmark we describe here measures sequential write throughput. 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 [4]. Using sequential writes we minimize disk latency (*i.e.*, seek time) on the server. We considered testing against a memory-only server, but we chose to design a benchmark that does not require atypical server modifications. Thus we have a simple and typical application to run on the client that exercises many of the critical paths between client and server.

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.

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) and local file systems (which may delay flushing), 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

We now report results of our benchmark 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.

3.1. Systems under test

In this section, we document 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 this 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. 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, probably based on the Alteon 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[1]. 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. The test volume contains eight disks. 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).

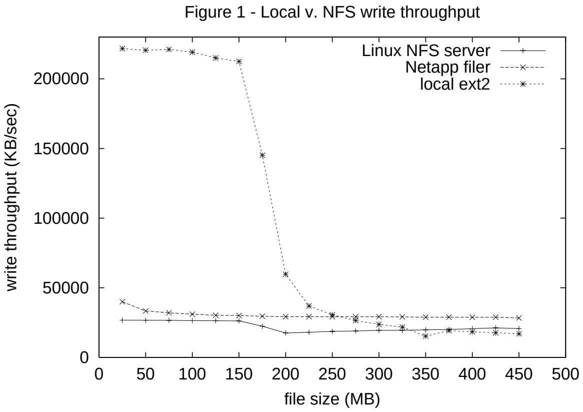

Figure 1. Local v. NFS memory write performance. *Write throughput is measured for test files between the sizes of 25 MB and 450 MB. Note that the large peak in memory write performance for local files does not appear for NFS files. NFS memory write throughput remains constrained to network/server throughput.*

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 networking speeds are fixed at one Gbps, full duplex.

Both the Network Appliance filer and the Linux NFS server are mounted with NFS version 3, rsize=wsize=8192. The Network Lock Manager is disabled.

3.2. Local versus network write performance

To begin, we compare the performance of sequential writes into a local file system (ext2 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). Ext2 memory write performance is a target for NFS client memory 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. The latter results are not included because ext2 usually does not flush after `close()`.

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

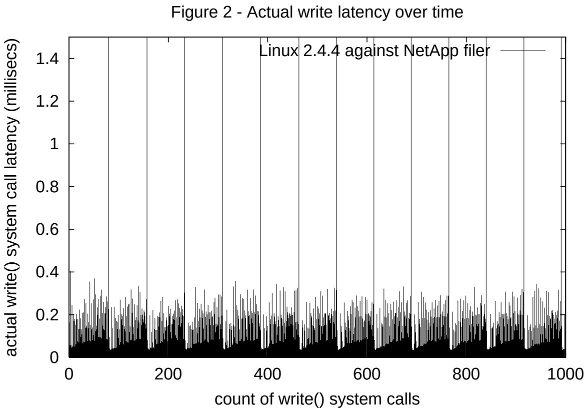

Figure 2. Write() system call latency. *This figure shows the first 1000 write system calls during a 40 MB benchmark run. Periodically, write system calls take more than 19 milliseconds, increasing the mean latency, and thus overall throughput.*

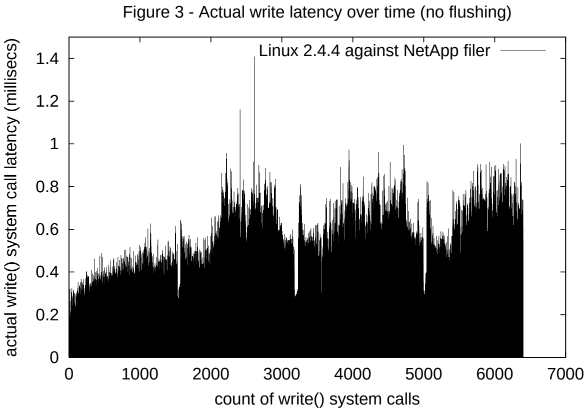

Figure 3. Write() system call latency without periodic flushes. *We show an entire benchmark run with a 100 MB file. The latency axis is the same as Figure 2. The periodic spikes in write system call latency are gone, but average latency grows worse over time.*

Writes to local files are very fast while there is still memory available to cache dirty data. Yet, the NFS client constrains write throughput to network speeds. In the next section, 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 1.4%), 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, multiplying the mean `write()` system call latency by 3.45.

We observed 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 writing 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 completing 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 puts any thread writing to that file system to sleep until another thread signals there are fewer than `MAX_REQUEST_HARD` requests. Each internal write request is no larger than a page.

Every system call in our test generates two write requests (8192 bytes is two pages, thus two requests). After the test 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 is the cause of the spiky latency seen in Figure 2.

In the Linux NFS client, there is a separate daemon that flushes cached write requests behind a writing application, called `nfs_flushd`. Ideally, the client should cache as many requests as it can in available memory [3]. There is no need to flush write requests unless the application requests it (via `fsync()` or `close()` for example), or unless the client cannot allocate more memory for new requests, in which case the VFS layer blocks the writer.

After removing the redundant flushing logic from the client, we ran our benchmark again. We see in Figure 3 that this eliminates the periodic latency spikes. However, mean latency does not improve: for the entire run (6400 writes) the average latency is 484.7 microseconds. The figure shows that latency increases over time. This suggests that as write requests build up in the client, data structure traversal becomes a performance-limiting factor.

3.4. List scans and sequential write performance

Experience tells us that scalability problems are often the result of lengthy data structure traversals. To establish whether data structure traversal limits throughput in this case, we use a kernel-profiling tool that provides a sample-driven histogram of kernel execution that pinpoints areas of heavy CPU usage in the kernel.

The profiler reveals 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.

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

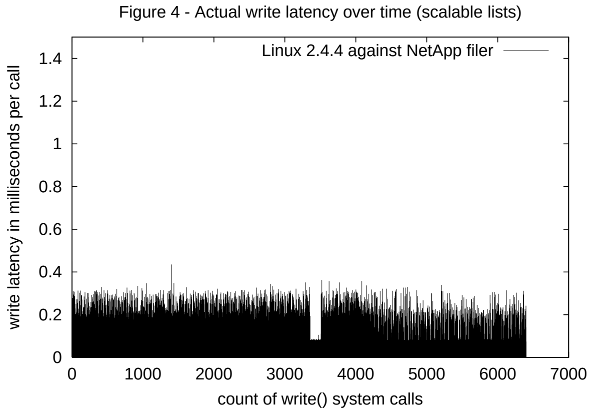

Figure 4. Write latency with scalable data structures. *Write latency remains low even as the number of outstanding requests increases for the entirety of this benchmark run against a 100 MB file. For comparison, the latency axis is the same as in Figures 2 and 3.*

To correct this scalability defect, 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.

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.

We see the improvement 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.

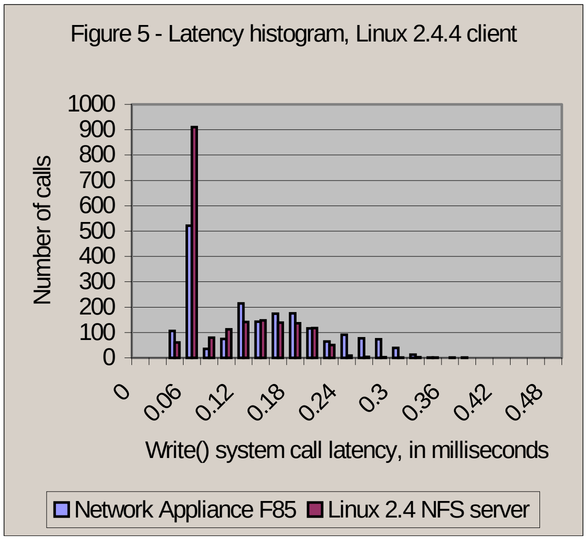

Figure 5. Write latency against different servers. *This figure shows the latency of write calls during a benchmark run against a 30 MB file. Both runs have about the same minimum latency, but the filer run has a number of calls that take longer than the Linux run. The* ***average*** *latency of client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

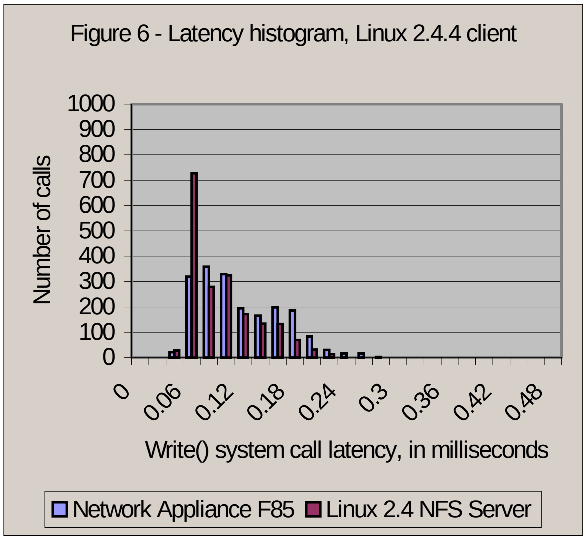

Figure 6. Write latency with less lock contention. *This figure shows maximum latency and latency variation (jitter) is clearly reduced. On average, filer writes still take longer than writes to the Linux NFS server, but the difference is small. Minimum latency remains roughly the same, suggesting that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

The client attempts to find a matching previous write request twice during each `write()` system call. Before it handles the current request, it looks for incompatible requests that might need to be flushed first. An example of an incompatible request might be a request generated by another application in a locked region outside the current request, but on the same page; to maintain proper write ordering, such a request needs to be flushed before the current request. A slight additional improvement here might occur if the search for incompatible requests was combined with the second search for a matching request (in `nfs_updatepage`).

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. A possible explanation appears at the end of the next section.

3.5. Global kernel lock on SMP hardware

Using the above modifications (no extra flushing in the write path, and a scalable hash table to track write requests), we compare write throughput performance of our client against a Network Appliance filer and against a four-way Linux NFS server.

During a typical benchmark run 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 NFS server can sustain only 26 MBps of network throughput (less than 70% of the filer's network throughput), yet our benchmark can write at a rate greater than 138 MBps (20% faster than the filer run).

To explore this discrepancy, 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 buffers writes more efficiently 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. This suggests that the RPC implementation or network layer is impeding the NFS client's write path.

	Normal	*No lock*
NetApp filer	115 MBps	140 MBps
Linux NFS server	138 MBps	147 MBps

Table 1. Client memory write throughput, before and after lock modification. *Removing the global kernel lock from the RPC layer causes improvement in memory write throughput for files residing on both the Network Appliance filer and the Linux NFS server. Even though the Network Appliance filer is faster than the Linux NFS server is, the client's lack of scalability slows memory write throughput to it more.*

Kernel execution profiling shows that the global kernel lock taken in `nfs_commit_write()` is under contention on SMP hardware. The lock text 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.

We therefore 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, it is safe to release the lock before calling `sock_sendmsg()`, then reacquire the lock when it returns, as long as the RPC layer does not require that the lock be held over the call. This allows other writing processes to make progress while the network layer sends the current request.

Figure 6 illustrates 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.

Note also 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.

Running our 5 MB benchmark again with the lock modification, the benchmark application generates almost 140 MB of data every second (almost a 22% increase over the original throughput of 115 MB per second). The benchmark generates 147 MB per second of data against a test file on the Linux server. Lock contention measured by the profiler is almost entirely gone. Application write throughput for the F85 and for the Linux NFS server are now almost in the same ballpark. These results are summarized in Table 1.

Even though the Network Appliance filer provides better network throughput than the Linux NFS server does, applications writing to files on the filer are slowed by the lack of client scalability. Despite the fact that less client processing is required for filer writes because they don't require an additional COMMIT RPC, client throughput to a fast server is hampered by lock contention, the cost of sending data to the server faster, and the cost of handling reply interrupts at a higher rate. Simply put, as servers become faster, a client must do the same amount of work in a shorter amount of time.

During a test with a single application writer thread contending with a single flusher thread, we find 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 exhibit even worse performance on SMP Linux clients until this issue is properly addressed.

Recall the short period in Figure 4 during which `write()` system call latency is much lower on average. This is likely due to reduced SMP lock contention on the client that occurs when the filer briefly stops responding to network write requests during a file system checkpoint [2]. 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 does not make any forward progress, only the application writer thread is active. Other threads do not interfere with the writer, allowing the client to return control quickly to the application.

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 packets, a feature of gigabit Ethernet, may help by reducing the need for fragmenting and reassembling large RPC requests in the IP layer.

However, we have demonstrated that removing the global kernel lock from the write path yeilds considerable improvements in throughput and application concurrency. Currently the RPC layer requires the global kernel lock to ensure the integrity of its internal data structures. Removing the global kernel lock from the RPC layer will allow a system with multiple network interfaces to process more than one RPC request at a time and allow concurrent writes to separate files and to separate servers from separate client CPUs.

3.6. Final measurement

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

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.

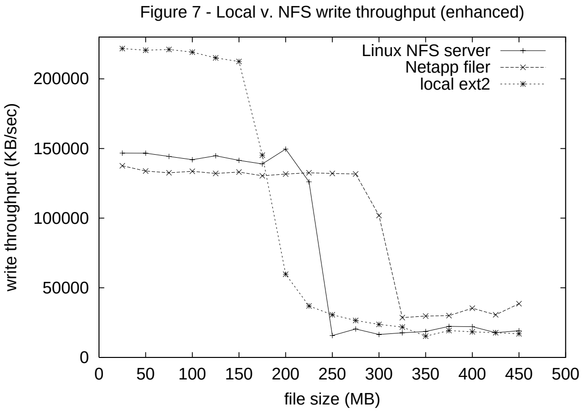

Figure 7. Local v. NFS memory write performance, revisited. *Write throughput is measured for test files between the sizes of 25 MB and 450 MB. NFS write throughput is considerably better than in Figure 1. Application write throughput no longer tracks network write throughput for NFS files. Maximum memory write throughput is nearly the same for both servers tested.*

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 Linux NFS server has a slight advantage. This advantage disappears as client scalability concerns are addressed. Where applications write then immediately flush or close, or where applications require data permanence before a `write()` system call returns, the Network Appliance filer, with its greater network and disk throughput, performs better. Though memory 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.

4. Future work and conclusion

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.

We also demonstrate some interesting aspects of client benchmarking. Where NFS server benchmarking is a direct measurement of on-the-wire behavior, NFS client measurement is a subtle and indirect affair, best accomplished using comparison. Standard file system performance benchmarks are useful in assessing client performance, but a single benchmark run may not tell the whole story. Interesting client behaviors emerge when comparing benchmark runs against several different servers.

Using comparison, we have confirmed interactions between client and server implementations that hamper application performance with fast servers and networks. A Linux client paired with a fast server exposes scalability issues in the client. Escalating server performance must be matched by attention to client scalability.

We want to assess further the impact of the global kernel lock on the scalability of the Linux NFS client. Further, we 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. These techniques are also valuable for surveying NFSv4 client implementations. 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 combined with network-attached storage.

Acknowledgements

The authors gratefully acknowledge the assistance of the following: Andy Adamson, Kendrick Smith, Brian Pawlowski and Sudheer Miryala at Network Appliance, Spencer Shepler and Sun Microsystems, James Newsome, Steve Molloy, and especially Trond Myklebust for his helpfulness and thorough work on the Linux NFS client so far. Special thanks also to reviewers. The Intel Corporation provided hardware used in this study.

A patch against Linux kernel 2.4.4 that includes the modifications discussed in this paper is available at the CITI U-M web site:

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

References

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

2. Hitz, D., Lau, J., and Malcolm, M. “File System Design for an NFS File Server Appliance.” *USENIX Technical Conference Proceedings*, Winter 1994.

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

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

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

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

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

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