Linux NFSv4.1 Performance Under a Microscope

Ming Chen,[1] Dean Hildebrand,[2] Geoff Kuenning,[3] Soujanya Shankaranarayana,[1]
mchen@cs.stonybrook.edu, dhildeb@us.ibm.com, geoff@cs.hmc.edu, soshankarana@cs.stonybrook.edu
Vasily Tarasov,[1,2] Arun O. Vasudevan,[1] Erez Zadok,[1] and Ksenia Zakirova[3]
{vass, aolappamanna, ezk}@cs.stonybrook.edu, kzakirova@g.hmc.edu
[1]*Stony Brook University,* [2]*IBM Research—Almaden, and* [3]*Harvey Mudd College*

FSL Technical Report FSL-14-02

Abstract

NFS is a highly popular method of consolidating file resources in today's complex computing environments. NFSv4.1, the latest version of the NFS protocol, has improvements in security, maintainability, and performance. Before system administrators can take advantage of NFSv4.1 in production systems, we need a good understanding of its performance. For this purpose, we present a detail-oriented benchmarking study of Linux's implementation of NFSv4.1 using micro-workloads.

The NFSv4.1 servers in the 2.6.32 and 3.12.0 kernels performed well in most of our experiments. However, we also observed that Linux's memory management can waste up to 80% of NFS I/O throughput. NFS performance also suffers from unfair networking behavior that causes the throughputs of identical NFS clients to differ by a factor up to $19\times$. We show that NFS delegations can boost performance by saving up to 90% of network traffic, but a delegation conflict can cause a delay of at least 100ms—more than $500\times$ the RTT of our network. We also found that NFS can exhibit counterintuitive performance behavior due to its intricate interactions with networking and with journaling in local file systems.

Tags: benchmarking, file system, NFS, performance analysis, storage systems.

1 Introduction

NFS has a 30-year-long history. The first publicized version of NFS was NFSv2 [44]; since then, two more major versions (NFSv3 [4] and NFSv4 [41]) and one minor version (NFSv4.1 [40]) have evolved. NFS is an unusually popular network-storage solution; even in 1997 it was estimated that there were 10–12 million NFS clients globally [25] and the number has surely increased since then [45]. Faster networks, the proliferation of virtualization, and the rise of cloud computing all contribute to continued increases in NFS deployments. NFSv4's improvements in security, scalability, cross-OS interoperability, and performance (e.g., pNFS) will further increase its popularity.

NFSv2 and NFSv3 are stateless, enabling a simple server implementation and easy crash recovery. NFSv2 used UDP; NFSv3 added TCP support, 64-bit file sizes and offsets, asynchronous COMMITs, and improved performance using specialized calls such as READDIRPLUS.

The stateless nature of NFSv2 and NFSv3 means that they depend on auxiliary RPC protocols for state-involved tasks such as mounting, locking, quota management, etc. Historically, these auxiliary protocols were implemented as stand-alone services working on separate ports. This made NFS difficult to use in WANs as it requires system administrators to configure all the port numbers properly to pass through firewalls. NFSv2 and NFSv3 also lack strong authentication and security mechanisms.

To solve the above problems, NFSv4, a completely new generation of NFS, was ratified as an Internet standard in 2003 [41]. Important changes made by NFSv4 include: (1) NFSv4 is stateful. (2) It does not use auxiliary protocols and incorporates all services into a single network port. (3) NFSv4 mandates a security model based on the RPCSEC_GSS [10]. (4) It enables advanced and aggressive caching using delegation. (5) Cross-OS interoperability is improved. (6) Seven years after NFSv4, NFSv4.1 introduced *Sessions* to provide exactly-once semantics and pNFS to allow direct client access to multiple data servers [11, 24, 40].

NFSv4 also offers new features for improving performance, such as (1) delegations, which enable the control of a file or directory to be passed to client(s); (2) operation coalescing via COMPOUND procedures; and (3) multi-component lookup to reduce the number of network messages.

NFSv4.1 has been "ready" [15, 31] for prime time deployment for a couple of years. However, as it is still new and complex, it is less understood than older versions. Before adopting it in production environments, it is important to understand how it behaves in realistic environments. We believe that completely new evaluations are necessary to make system administrators well-informed before migrating systems to NFSv4.1. We chose to evaluate NFSv4.1 rather than NFSv4.0 because we believe that it meaningfully alters client-server communication as compared to NFSv4.0 and that it will eventually replace NFSv4.0 in production environments.

As the first step of a comprehensive evaluation of NFSv4.1, we benchmarked the Linux NFSv4.1 implementation using a variety of micro-workloads. We be-

gin our study with benchmarking micro-workloads in a simple setup in the belief that understanding NFSv4.1 in simple workloads and environment is the foundation of understanding it in complex workloads and environment. Our in-depth analysis of the micro-workload results reveals that NFS's performance depends heavily on its interactions with many other system components such as memory management, networking, and local file systems. We identify several unexpected interactions that lead to low throughput or unfairness among NFS clients. We demonstrate that NFS delegation can significantly boost performance, but can also incur large latencies in the presence of delegation conflicts. Where possible, we make recommendations for improving NFS's performance under specific workloads.

The rest of this paper is organized as follows. Section 2 describes our benchmarking methodology. Sections 3 and 4 discuss the results of random-read and sequential-read tests, respectively. Section 5 covers NFS delegations. Section 6 examines other micro-workloads including random write, sequential write, and file creation. As a forerunner of our further study, Section 7 studies one macro-workload, a File Server. We discuss related work in Section 8 and conclude in Section 9.

2 Methodology

This section details our experimental setup, benchmarking methodology, and workloads.

2.1 Experimental Setup

We used six identical Dell PowerEdge™ R710 machines for our study. Each has a six-core Intel Xeon™ X5650 2.66GHz CPU, 64GB of RAM, a Broadcom BCM5709 1GbE card, and a Dell PERC 6/i RAID controller with a 256MB battery-backed write-back cache. Five NFS client machines are configured to mount from one NFS server machine. Each computer had a dedicated 150GB SEAGATE ST9146852SS SAS drive for the OS. On the server, we set up a RAID-0 (often referred as "disk" for simplicity) for the NFS data, with a stripe size of 64KB, using two additional SEAGATE ST9146852SS drives.

We connected the six machines to a LAN using a PowerConnect J-EX4200 48-port 1GbE switch. We measured a round-trip time (RTT) between two machines of 0.17ms, and a raw TCP bandwidth of 117MB/s. We note that our setup of 1GbE network and HDDs is not the state-of-the-art. However, as we will show, its simplicity allows us to identify problems that are generic and applicable to different setups as well.

All machines ran CentOS 6.4, the latest version at the time when we started this study. We chose CentOS because it is a freely available version of Red Hat Enterprise Linux, often used in enterprise environments. We experimented with both the 2.6.32-358.el6 (referred as 2.6.32el6) kernel that comes with CentOS 6.4, and a locally compiled vanilla 3.12.0 kernel. As our benchmarks will show, there are many significant NFSv4.1 performance differences between the old and new kernels.

Many parameters affect NFS performance, including local file system type, format and local mount options, network parameters, NFS and RPC parameters, and NFS export and client mount options. We did not change any OS parameters if not stated otherwise. We used the default `ext4` file system, with default settings, for the RAID-0 NFS data drive, and chose the in-kernel implementation of the NFS server. Specifically, we used NFS Version 4 Minor Version 1, the latest minor version of NFSv4. By default, NFSv4.1 implementations do not use RPCSEC_GSS (although it is supported). Since we focused on NFSv4.1 performance, we avoided security overhead and chose not to use RPCSEC_GSS.

We exported an `ext4` directory via NFSv4.1 using the default options, ensuring that `sync` was set and writes were faithfully committed to stable storage as requested by clients. We used the default RPC settings, where the number of RPC slots is dynamically allocated. The number of NFSD threads in our benchmarks is 32. (We have tried different thread counts but observed no noticeable performance difference because the performance was primarily constrained by either the disk or the network.) We also used the default NFS mount options. The NFSv4.1 `rsize` and `wsize` are 1MB.

2.2 Benchmarks and Workloads

We developed a benchmarking framework, named *Benchmaster*, that can launch workloads on multiple clients concurrently to minimize the potential negative effects of time-unaligned benchmark startups. To verify that Benchmaster can launch time-aligned workloads, we measured the time difference by NTP-synchronizing the clients and then launching a trivial program that simply writes the current time to a local file. We ran this test 1,000 times and found an average delta of 235ms and a maximum of 432ms. This variation is negligible compared to the 5-minute running time of our benchmarks.

Benchmaster also periodically collects system statistics from various tools (`iostat`, `vmstat`, etc.) and `procfs` entries (`/proc/self/mountstats`, `/proc/fs/jbd2/sda/info`, etc.). This allows us to see the changes in system behavior over time.

We ran tests for 5 minutes and performed at least three runs, and computed the 95% confidence intervals for the mean using the Student's t-distribution. Unless otherwise noted, we plot the mean of three runs' results, with the half widths of the intervals shown as error bars.

We benchmarked 5 common micro-workloads including random read, sequential read, random write, se-

quential write, and file creation. We chose these workloads for four reasons: (1) They exercise NFS and help identify performance bottlenecks and anomalies—they revealed several problems that we discuss in this paper. (2) They are fundamental building blocks for many macro-workloads. We believe understanding these micro-workloads is essential to understanding more complex workloads. (3) They are simple and thus can better isolate the individual effects we are examining. (4) They can be easily tweaked to simulate many NFS workloads with different characteristics. For example, we can simulate disk-write-intensive workloads by specifying `O_SYNC` in the random-write workload, and simulate metadata-intensive workloads by creating a large number of small files in the file-creation workload.

We used two micro-workloads to examine the new NFSv4 delegation feature, exploring both its potential benefits and its cost in the presence of conflicting clients. We also benchmarked one File Server macro-workload.

3 Random Read

This section discusses an NFS random-read workload where five single-threaded NFS clients randomly read a 20GB file with a given I/O size. Since the server's page cache can play an important role in this workload, we started the experiment with both warm and cold NFS server caches to evaluate the effects of both the network and the disk.

We studied the warm-cache case first since it does not involve disk I/Os and is thus simpler. Figure 1 shows the NFS random read throughput results on the 2.6.32el6 and 3.12.0 kernels. We present two types of throughputs: one is *application read throughput* (ART), which measures the rate at which the application itself receives data; the other is *client read throughput* (CRT), which measures the rate at which the NFS client machine receives data from the server. We show the results for only one client because all five clients produced similar performance. The only noticeable difference is the CRTs of the 2.6.32el6 kernel, which have large variations but follow the same trend on all clients.

In Figure 1, the CRTs of the 3.12.0 kernel, for all three different I/O sizes, stabilize at around 22MB/s, because they are bounded by our network bandwidth, which is 117MB/s. Considering the overhead of extra RPC headers and NFS bookkeeping messages, an aggregated throughput of 110MB/s is reasonable. The ARTs of the 3.12.0 kernel start the same as the CRTs, but increase slightly over time. It is because of the client-side cache. As we read more data, the cache size increase and the cache hit-ratio increase correspondingly.

In Figure 1, the CRTs of the 2.6.32el6 kernel, for all three different I/O sizes, are also close to 22MB/s but have large variations. The ARTs of the 2.6.32el6 kernel start the same as the corresponding CRTs but then drop almost monotonically. As time goes by, the ARTs become much lower than the corresponding CRTs, suggesting that a smaller portion of the data read by the NFS client is actually being delivered to the application.

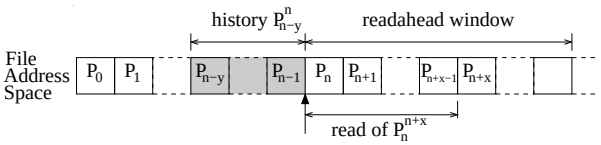


Figure 3: Linux readahead algorithm. The boxes represent pages, and the shaded ones are cached pages.

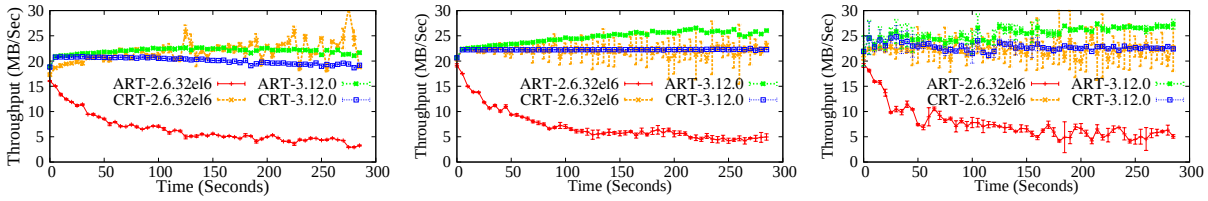
We note that the ARTs of the two kernels start about the same but increase in 3.12.0 and decrease in 2.6.32e16. For all three I/O sizes—4KB, 16KB, and 64KB—the speedup in average ART when going from 2.3.32el6 to 3.12.0 was 3.3–3.4×. The 3.12.0 kernel also has more stable CRTs over time.

The dramatic difference of ARTs between the two kernels is caused by the NFS clients' readahead mechanism, which is similar in both kernels except for one small but significant difference. The readahead algorithm is illustrated in Figure 3, where P_i is the i^{th} page in a file's address space, and P_i^{i+j} are the j consecutive pages starting from P_i. For each read request, the algorithm asks two questions: (1) whether readahead should be performed, and (2) how much readahead to do. Both questions depend on the *history* size, which is the number of consecutive cached pages immediately before the current read offset. Considering a read request of P_n^{n+x} with an offset at P_n and an I/O size of x pages, its history is P_{n-y}^n, where y is number of cached pages immediately preceding page n.

The 2.6.32el6 and the 3.12.0 kernels differ in how they answer the first readahead question. The old kernel tests whether $y > 0$, while the new one checks $y > x$ where x is the read request size. Readahead in the 2.6.32el6 kernel is aggressive but susceptible to false positives. Upon reading P_n^{n+x}, readahead takes place if P_{n-1} happens to be cached. This explains why we observed many readahead requests even in this random-read workload. Readahead in the 3.12.0 kernel is more conservative. For a 64KB I/O request, the required history size is 17 pages—17× the corresponding size in 2.6.32el6. Therefore, the probability of readahead for random reads is much lower in 3.12.0.

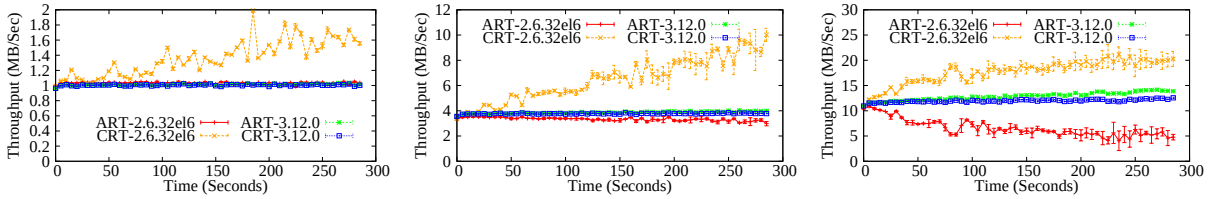
■ **Observation 1:** *The readahead algorithm in Linux 3.12.0 is less aggressive than in 2.6.32el6, leading to fewer false positives that are detrimental to NFS random-read performance.*

The answer to the second question of how much readahead to perform is the same in both kernels. But it is more complex and depends on many factors. Due to space constraints, we note only the two factors that are



(a) 4KB I/O (b) 16KB I/O (c) 64KB I/O

Figure 1: Random read: Application Read Throughput (ART) and Client Read Throughput (CRT) with default readahead in the 2.6.32el6 and 3.12.0 kernels. There are 5 clients; Client1 is shown here and the others are similar. The server's cache is warm.



(a) 4KB I/O (b) 16KB I/O (c) 64KB I/O

Figure 2: Random read: Application Read Throughput (ART) and Client Read Throughput (CRT) with default readahead in the 2.6.32el6 and 3.12.0 kernels. There are 5 clients; Client1 is shown here and the others are similar. The server's cache is cold.

relevant to our results: (1) the readahead size is large when the history size is large, and (2) the size is scaled against and constrained by the *maximum readahead size* (MRS), a tunable parameter. The first factor explains the ART drop of the 2.6.32el6 kernel in Figure 1: as we read more data over time, more pages in the address space get cached by the clients. Consequently, the history size and the readahead size both increase. Since we were doing random reads, a larger readahead size means more work is wasted and a smaller portion of the CRT is being converted to ART.

In both kernels, NFS has a default MRS as large as 15MB, which also contributes to the excessive readahead in 2.6.32el6. The MRS of an NFS mount point is the product of `NFS_MAX_READAHEAD`, a constant fixed at 15, and `rsize`, an NFS mount option with a default value of 1MB.

Figure 2 shows the random-read throughputs when the server's cache is cold. As expected, the throughputs are much lower than their counterparts in Figure 1. Readahead is observed in the 2.6.32el6 kernel for all I/O sizes; thus the CRTs are higher than the corresponding ARTs. This is true even in the 4KB I/O case, where all throughputs are lower than 2MB/s. In contrast, no significant readahead is observed in the 3.12.0 kernel. Comparing the two, Linux 3.12.0 has an ART that is on average 2× higher than Linux 2.6.32el6 for 64KB I/Os.

The 2.6.32el6 kernel behaved similarly to the new kernel when we set the MRS to small values. We did not completely disable it because in both kernels, disabling readahead has the unexpected side effect of breaking large I/O requests into numerous smaller ones. When there is no readahead, Linux calls `readpage` instead of `readpages`, so all NFS READs become single-page. With a 64KB I/O size, 1 client, and a warm server cache, we measured an average ART of only 22MB/s when MRS was zero, but 68MB/s when MRS was 64KB.

4 Sequential Read

This section discusses an NFS sequential-read workload, where five NFS clients repeatedly scan a 20GB file from beginning to end. All clients read the file as fast as possible. The benchmark I/O size is 64KB and the readahead settings have default values.

4.1 The Winner-Loser Pattern

In contrast to random workloads where the throughput bottleneck is the server-side disk, sequential workloads are limited by the 1GbE network. Both the application and client read throughputs of the clients sum to around 112MB/s in both the 2.6.32el6 and 3.12.0 kernels.

When we analyzed the read throughput of individual NFS clients, we frequently observed that two of the five had a throughput of around 14MB/s, while the other three reached about 28MB/s. Occasionally, two clients had throughputs around 37MB/s, and the other three had throughputs around 12MB/s. There were also cases when all five clients had the same throughputs, around 22.5MB/s. In all cases, the throughputs were stable over time within a single experiment, but not stable across experiments. For example, Client1 might have a constant throughput of 14MB/s in one run, but a constant throughput of 28MB/s in another run. Note that in all cases, the aggregated throughput was 112MB/s.

In approximately 80% of our experiments, we observed this *winner-loser pattern*, which has three characteristics: (1) there are two clusters of clients, one with high throughput (winners) and another with low

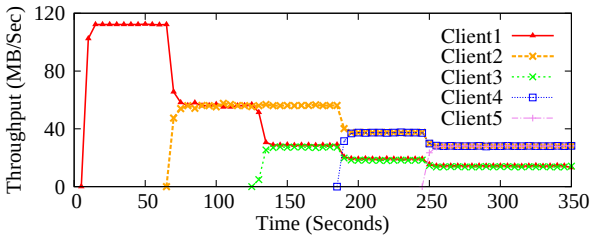


Figure 4: Sequential read: throughputs when clients are launched one after the other at an interval of one minute. The kernel version is 2.6.32el6.

throughput (losers); (2) the winners and losers are consistent over time within one experimental run but are not consistent across multiple runs; and (3) most often, the winner throughputs are about twice those of the losers—but occasionally the winners have about 3× the throughputs of the losers.

■ **Observation 2:** *When multiple NFS clients are reading data from an NFS server, there can be a winner-loser pattern where the network bandwidth is unfairly distributed among the clients.*

The winner-loser pattern was unexpected since all five clients in our experiments have the same hardware, software, and settings, and they are performing the same operations. Initially, we suspected that the pattern was caused by the order in which the clients launched the workload. To identify any correlation between launch order and the winner-loser pattern, we repeated the same experiments but launched the clients in a controlled order, one additional client every minute. Figure 4 shows the result of one such experiment, where Client1 started first but ended up as a loser, whereas Client5 started last but became a winner. It was clear that there was no correlation between experiment launch order and the winner-loser pattern. (Note that in Figure 4 the winners always have about twice the throughputs of the losers.)

Moreover, we were able to reproduce the winner-loser pattern at the TCP level using a simple test. A (non-NFS) client establishes a TCP connection to a server and then uses multiple threads to request data as fast as possible; the server sends null data to the clients as requested. The fact that the winner-loser pattern happens in TCP rather than NFS has broad implications since TCP accounts for 85% to 95% of wide-area Internet traffic [30]. We believe Hash-Cast has impact on a large number of TCP based applications as TCP Incast [35], another networking problem identified in storage systems, has.

4.2 Hash-Cast

To understand the winner-loser pattern, we analyzed the network traffic, but found no sign of network congestion (no packet losses, retransmissions, or Explicit Congestion Notifications), nor any significant difference among the TCP congestion window sizes (*cwnd*) of the clients'

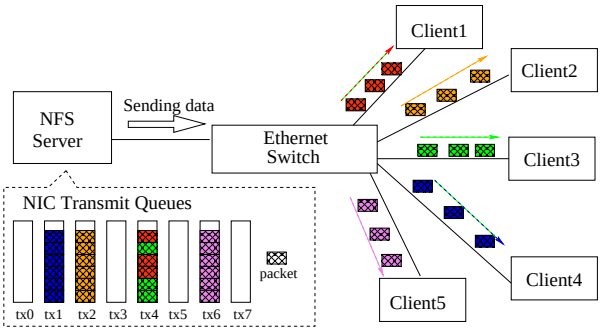


Figure 5: Illustration of Hash-Cast.

connections. We traced the networking stack on both the client and the server and discovered that the winner-loser pattern is closely related to the physical transmit queues of the server's network interface card (NIC).

A NIC typically has a physical transmit queue (`tx-queue`) holding outgoing packets, and a physical receive queue (`rx-queue`) tracking empty buffers for incoming packets [38]. The `tx-queue` is FIFO [42], but this does not prevent Linux from advanced traffic control because packets are regulated by queue disciplines (*qdisc*s) in upper-layer software queues.

Many modern NICs, especially advanced ones, have multiple sets of `tx-queues` and `rx-queues` [38]. Those multi-queue NICs are becoming increasingly common for at least four reasons: (1) They allow networking to scale with the number of cores in modern computers. Each queue has its own IRQ and can be configured to interrupt specific core(s). (2) They provide larger bandwidth, as seen in 10GbE and 40GbE NICs [16]. (3) They fit well with wireless devices [42]. (4) They facilitate increasingly popular virtualization techniques with better NIC virtualization, packet switching, and traffic management [38].

In the presence of multiple `tx-queues`, each outgoing packet needs to choose one `tx-queue` to use. Linux uses hashing for this purpose. However, not all packets are hashed; instead, each TCP socket has a field recording the `tx-queue` the last packet was forwarded to. If a socket has outstanding unacknowledged packets, further packets are placed in the recorded `tx-queue`. Otherwise the next packet is hashed to a random `tx-queue`. This approach allows TCP to avoid generating out-of-order packets by placing packet n on a long queue and $n+1$ on a shorter one. However, a side effect is that for highly active TCP flows, the hashing is effectively done per-flow rather than per-packet.

The winner-loser pattern is caused by uneven hashing of TCP flows to `tx-queues`. In our example, the server has five TCP flows (one for each client) and a NIC with eight `tx-queues`. If two of the flows are hashed into one `tx-queue` and the rest are hashed into three separate `tx-queues`, then the two flows shar-

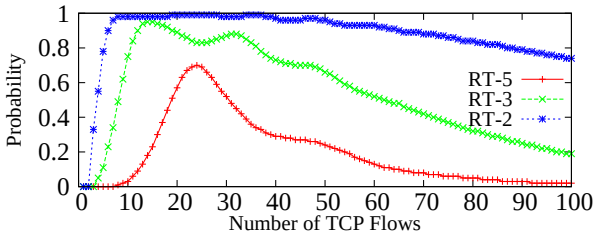


Figure 6: Probability of occurrence of Winner-Loser Pattern for different winner-loser ratio thresholds (RT).

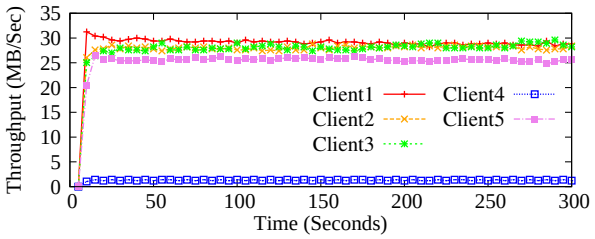


Figure 7: Sequential read: extreme winner-loser pattern with default settings in Linux 3.12.0.

ing a `tx-queue` will get lower throughput than the other three because all `tx-queues` are transmitting data at the same rate. We refer to this phenomenon—hashing unevenness causing a winner-loser pattern of throughput—as *Hash-Cast*. Hash-Cast is illustrated in Figure 5, which helps explain the performance in Figure 4. First, Client1 and Client2 were hashed into `tx4` and `tx2`, respectively. Then, Client3 was hashed into `tx4`, which Client1 was already using. Later, Client4 and Client5 was hashed into `tx1` and `tx6`, respectively. Hash-Cast also explains why the losers usually get half the throughputs of the winners: the $\{1,1,1,2\}$ distribution is the most probable hashing result. In the rarer case of a $\{1,1,3\}$ distribution, the winners get thrice the throughputs of the losers—and sometimes $\{1,1,1,1,1\}$ or the very rare $\{5\}$ distribution will achieve equality.

■ **Observation 3:** *The winner-loser pattern happens in the TCP layer and is caused by the Hash-Cast behavior of multi-queue NICs.*

Compared to explicit load balancing, hashing is a simpler and more efficient way to assign flows (or packets) to queues, since the uniform distribution of hash values promises stochastic fairness [32] over the long term. However, as noted by McKenney [32], stochastic fairness does not come for free, and the price is loss of determinism and of absolute fairness guarantees. Moreover, stochastic fairness reaches statistical uniformity only for large numbers of samples. With only a few `tx-queues` (8 in our NIC) and a small number of flows (less than 100), unfairness is highly likely.

A precise mathematical analysis of this phenomenon is beyond the scope of this paper, so to better understand the problem we defined the *winner-loser ratio* as the ratio of the highest to the lowest throughput. We then simulated the probabilities of observing a winner-loser ratio above n, for different values of n. The simulated probabilities for 8 `tx-queues` are shown in Figure 6. With a threshold of $n = 2$, the probability of seeing a winner-loser pattern is above 0.9 for any number of TCP flows between 6 and 60. Even with a high threshold of $n = 5$, the probability of the winner-loser pattern appearing is over 0.2 for any number of flows between 15 and 53, and above 0.4 for flow counts between 18 and 33.

■ **Observation 4:** *The winner-loser pattern shows up with a high probability when there is a small number of data-intensive TCP flows.*

4.3 Bufferbloat

To work around the Hash-Cast problem, we experimented with many networking parameters. One option is the TCP congestion-control algorithm, which defaults to TCP CUBIC. Of the 14 such algorithms available in the vanilla kernel, we found that only TCP VEGAS does not show the winner-loser pattern.

Comparing CUBIC with VEGAS in the 2.6.32e16 kernel, we found that the TCP congestion window size (*cwnd*) is markedly different between the two: 23 for VEGAS and 1,900 for CUBIC. Our 1Gbps network has an RTT of 0.17ms, so its bandwidth-delay product is about $10^9/8 \times 0.00017 = 21,250$. Since an Ethernet packet is about 1,500 bytes, this translates to about 14 packets that can be in transit at one time. VEGAS's 23-packet *cwnd* is of roughly comparable size, while CUBIC's 1,900 strongly suggests *bufferbloat* [20], a phenomenon where excessive network buffering causes unnecessary latency and poor system performance.

The TCP algorithm dynamically adapts to the connection speed by probing and responding to congestion; excessive buffers interfere with this mechanism by inserting artificial delay, eventually leading to unnecessary retransmissions and thus even greater congestion. The problem is worst when there are multiple flows, because one flow can create a queue that takes many seconds to drain, so that other flows see long delays. A shorter queue, in contrast, would allow each flow to adapt to the other's presence more quickly. In fact, when we changed TCP CUBIC and set its `cwnd_clamp`—an upper limit for `cwnd`—to 64, the winner-loser pattern disappeared.

■ **Observation 5:** *The winner-loser problem is exacerbated by bufferbloat.*

To combat bufferbloat, new features such as Byte Queue Limits (BQL) [7] and TCP Small Queue (TSQ) [8], have been introduced into new Linux kernels. We ran NFS sequential-read experiments under Linux 3.12.0 and observed a $10\times$ reduction in CUBIC's `cwnd`. Unfortunately, a winner-loser pattern still appeared.

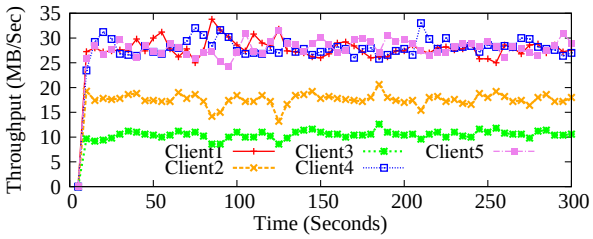


Figure 8: Sequential read: the winner-loser pattern when TSQ is 16KB in Linux 3.12.0.

However, the winner-loser pattern was different in three ways under 3.12.0: (1) The pattern can appear when there is no Hash-Cast, i.e., when all 5 clients have dedicated `tx-queues`. Out of 100 experimental runs, we observed 4 such cases, one of which is shown in Figure 7. (2) Losers do not always have one-half or one-third of the winners' throughput; sometimes, the loser can lose badly. For example, Figure 7 shows a case where Client4 has throughput below 1.3MB/s while all other clients receive more than 25MB/s; this is a 19× difference. (3) Unlike in Linux 2.6.32el6, the losers can have significantly different throughputs. For instance, in Figure 8, one loser (Client3) has a throughput of around 10MB/s and another (Client2) gets about 18MB/s.

Our investigation into Figure 7 revealed that the number of packets sent to each client is about the same, but Client4 has an average packet size of around 3KB while all other clients generate packet sizes over 50KB. (Our NIC supports TCP Segmentation Offload (TSO) [8], so packets can be larger than 1,500B.) The difference in packet sizes is caused by a new kernel feature called TSO packets Automatic Sizing (TSO-AS) [9]. The motivation for TSO-AS is to avoid bursty traffic by dynamically calculating the packet sizes of a TCP flow based on its previous sending rate. If a TCP flow starts with a low rate, this algorithm will assign it a small packet size. The packet size and the sending rate can form a feedback loop, which can force the TCP flow's rate to stay low. The author of the TSO-AS patch confirmed this in our discussion and submitted a patch to address this [17]. The author also pointed out that TCP is greedy by itself, and traffic control such as fair queueing is required to achieve fairness.

A smaller TSQ value can reduce the effects of TSO and TSO-AS, leading to uniform packet sizes among the clients. We lowered the value of TSQ but *still* observed the winner-loser pattern caused by Hash-Cast. Figure 8 is an example with TSQ set to 16KB. While the average packet size was about 7.3KB for all clients, Client2 and Client3 received fewer packets because they were sharing a `tx-queue`. Compared with results from the older kernel (not shown), Figure 8 is less stable and the two losers (Client2 and Client3) receive unequal throughputs—yet they still sum to 28MB/s, which is close to the throughputs of the winners in Figure 8.

We have also experimented with different settings of BQL, but found that it did not improve the winner-loser pattern. The throughputs of the clients were unpredictable, and we were not able to correlate the resultant values to any parameters, including Hash-Cast, number of packets sent, and the average packet sizes.

Note that TCP VEGAS does not show a winner-loser pattern in either Linux 2.6.32el6 or Linux 3.12.0. For subsequent experiments reported in this paper, we used VEGAS as our congestion-control algorithm to avoid the winner-loser pattern, which can distort the effects of other system components. We note that VEGAS does not interact well with non-VEGAS hosts and can result in low throughput in the presence of network rerouting [26]. However, considering our controlled network environment, the issues of VEGAS should be minor. Another alternative to work around the winner-loser pattern is using only one queue and fair queueing [17]. The argument is that TSO can be effective enough to achieve high utilization using just one queue. However, we are conservative about its performance impact on workloads with smaller I/O sizes, where TSO is less effective.

5 NFSv4 Delegation

This section discusses a key feature of NFSv4, delegations. We focus on read delegation of regular files because that is the simplest type, and also is the only one currently supported in the Linux kernel [18]. We discuss (1) how delegations are granted, (2) their benefits in the absence of inter-node conflicts on the delegated files, and (3) the costs of delegations when there are conflicts.

5.1 Delegation Grant

Delegation is a new feature of NFSv4 for improving performance. When the server delegates a file, the client obtains control of it. Until the delegation is released, the client does not need to ask the server to operate on the file. This can significantly boost performance since it saves the usually slow latency of network communications. Delegations are based on the observation that "file sharing is rarely concurrent" [27]. Thus it is expected to benefit performance most of the time. But it can also hurt performance if concurrent and conflicting file sharing does happen.

The NFS server grants delegations to clients in response to file opens. However, clients must not assume that a delegation will be granted, and after granting a delegation the server is free to recall it at any time via a back-channel connection.

To recall a delegation, an NFS server might have to exchange many messages with one or more clients, incurring a considerable delay. Therefore, the server should consider the probability of potential conflicts

NFSv4.1 Operations	No Delegation	Delegation
OPEN	1000	1000
READ	10000	1000
CLOSE	1000	1000
GETATTR	10001	1
LOCK	10000	0
LOCKU	10000	0
FREE_STATEID	10000	0

Table 1: NFSv4.1 operations performed by each client when a delegation is or is not granted. Each operation represents a compound procedure; trivial operations in the same compound procedure, such as PUTFH *and* SEQUENCE*, are omitted for clarity. Other seldom used operations, such as* FSINFO*, are also omitted. The kernel version is 3.12.0.*

when it is deciding to issue a grant. Relevant factors include back-channel availability, current conflicting operations, the client's delegation history, etc. However, because Linux supports only file read delegations, its NFS servers can use a simpler decision model. A delegation is granted if three conditions are met: (1) the back channel is working, (2) the client is opening the file with `O_RDONLY`, and (3) the file is not currently open for write by any client.

During our initial experiments we did not observe any delegations even when all three conditions held. We traced the kernel using `SystemTap` [36] and discovered that the Linux NFS server's implementation of delegation is outdated in that it does not recognize new delegation flags introduced by NFSv4.1. The effect is that if we get the filehandle of a file before we open it, for example by using `stat`, delegation will not be granted. To fix the problem, we created a kernel patch [6], which has been accepted into the Linux mainline.

5.2 Locked Read

To quantify the benefit of delegations when there are no client conflicts, we scaled up and performed the delegation experiment in Nache [23]. We pre-allocated 1,000 4KB files for each of five clients in five different NFS directories. For each of its files, a client opened the file once, then repeatedly locked it, read the entire file, and unlocked it. After ten repetitions the client closed the file and moved to the next one.

We list the number of NFS operations both with and without delegation in Table 1. Without a delegation, each application read introduced an NFS READ despite the fact that the same reads were repeated ten times. This happened because locked reads required the NFS client to ensure cache coherency by validating the latest file state with the server. An important detail is the timestamp granularity offered by the NFS server. Traditionally, NFS provides close-to-open cache consistency [29]. Timestamps are updated at the server when a file is closed, and any client subsequently opening the same file revalidates its local cache by checking the file's attributes with the server. If the locally-saved timestamp of the file is out of date, the client's cache of the file is invalidated. Unfortunately, some NFS servers offer only a one-second timestamp granularity, which is too coarse; clients might miss intermediate changes (within one second) made by other clients. In this situation, the NFS locking mechanism provides stronger cache coherency by first checking the server's timestamp granularity. If the granularity is finer than one microsecond, the client revalidates the cache by following the traditional approach of using GETATTR. For coarser granularities, the Linux client invalidates its cache of the locked file. Since the Linux in-kernel server uses one-second granularity, each client READ incurs a corresponding server READ as the preceding LOCK has invalidated its local cache.

Invalidating the entire cache of a file can be expensive, since NFS is often used to store large files such as virtual disk images, databases, audio and visual media, etc. This scenario is worsened by two factors: (1) cache invalidation happens even when we are just acquiring read locks, and (2) the entire cache of a file is invalidated even if we are just locking one byte of that file.

■ **Observation 6:** *Using the Linux NFS client and server, locking an NFS file is an effective way to achieve cache consistency, but it can be expensive because it invalidates the file's entire client-side cache.*

In contrast, the NFS client with delegation was able to satisfy nine of the repeated ten READs from the page cache. There was no need to revalidate the cache at all because its validity was guaranteed by the delegation.

The second difference we noticed between the no-delegation case and delegation case is the number of GETATTR operations, which is a side effect of the cache invalidation caused by locking. For each file read, the client revalidates the file's inode information (mapping) before sending the READ request. Note that the preceding locking invalidates not only the file's page cache, but also its cached inode. A GETATTR is thus sent for this revalidation. A potential optimization would be to have the client append a GETATTR operation to the LOCK and the server piggyback file attributes in its reply. This could save 10,000 RPCs for the GETATTRs.

The remaining differences between the experiments with and without delegations were due to locking. A LOCK/LOCKU pair is sent to the server when the client does not have a delegation; conversely, no NFS communication is needed for locking when delegation exists. One FREE_STATEID follows each LOCKU to free a stateid that no longer has any associated locks. A potential optimization here would be to append the FREE_STATEID operation to the compound procedure of LOCKU. This could save another 10,000 RPCs.

NFSv4.1 Operations	No Delegation	Delegation
OPEN	1000	1000
DELEGRETURN	0	1000
OPEN_NOATTR	0	1000

Table 2: NFSv4.1 operations performed by each DG client. OPEN_NOATTR is not an NFS operation, but rather a Linux shorthand for an NFS compound procedure containing three operations: SEQUENCE, PUTFH, and OPEN.

NFSv4.1 Operations	No Delegation	Delegation
OPEN	1000	2000

Table 3: NFSv4.1 operations performed by each RG client.

In total, the number of NFS operations performed by each client without delegation was over 52,000. That number dropped to just 3,000 with delegation. In terms of running time, the experiment finished in 7.8 seconds without delegation, but only 0.76 seconds with delegation. That is, delegation speeds up this particular experiment over 10×. In terms of the amount of data sent and received, the no-delegation case sent 9.6MB of data and received 45.8MB, whereas the delegation case sent 0.6MB and received 4.5MB. Delegation reduces both the incoming and the outgoing traffic by 90%.

■ **Observation 7:** *NFS read delegation can effectively improve performance by reducing the number of NFS operations when files are locked and there are no delegation conflicts among clients.*

5.3 Delegation Recall

To explore the overhead of conflicting delegations, we created two groups of NFS clients. Clients in the first group, called Delegation Group (DG), grab and hold NFS delegations on 1,000 files. Clients in the second group, Recall Group (RG), recall the NFS delegations of the 1,000 files held by the DG clients. DG clients obtain their delegations by opening the files with `O_RDONLY`; RG clients recall them by opening the same files using `O_RDWR`. We varied the number of clients in RG from one to four to test the scalability of delegation recall. For n clients in the DG, there were n recalls when a RG client opened a file because each DG client's delegation had to be recalled separately.

We compared the two cases when the DG clients were and were not holding delegations. Table 2 and 3 list the number of NFS operations performed by each DG and RG client, respectively. Each DG client took two NFS operations to respond to a delegation recall. The first is a DELEGRETURN, which returns the delegation to the server. The second is an OPEN, which the DG client used to re-open the file since the old stateid associated with the delegation was no longer valid.

For the RG client, the presence of delegation incurred one additional NFS OPEN per file. The first OPEN failed with `NFS4ERR_DELAY` because the server needed to recall outstanding delegations. The second open was sent as a retry and succeeded.

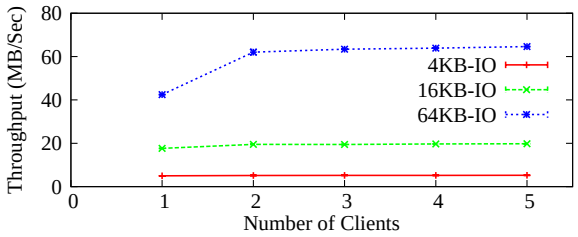


Figure 9: Random write: throughput with O_SYNC. When there are multiple NFS clients, the throughput plotted is the sum of all clients' throughputs. The kernel version is 3.12.0.

The running time of the experiment varied dramatically: 0.2 seconds in the no-delegation case and 100 seconds in the delegation case. The 500× delay was introduced by the RG client, which failed in the first OPEN and retried it after a timeout. The timeout is initialized to 0.1 second, and is doubled every time the retry fails. This timeout dwarfed other delays, and the running time was 100 seconds no matter how many clients existed in DG. An initial timeout as long as 0.1 second is questionable considering our small networking latency. Also, without write delegation, there is no need to allow time to flush dirty data upon recall. We believe it would make sense to start with a smaller timeout; if that turns out to be too small, we can back off quickly since timeouts increase exponentially.

■ **Observation 8:** *In case of delegation conflict, an NFS open will be delayed for at least 100ms—more than* 500× *the RTT of our 1GbE network.*

6 Other Micro-Workloads

This section discusses three more micro-workloads: random write, sequential write, and file creation.

6.1 Random Write

The random-write workload is exactly like the random-read one discussed in Section 3 except that the clients are writing data instead of reading. Each client has one thread that repeatedly writes a specified amount (I/O size) of data at random offsets in a preallocated 20GB file. All writes are in-place and do not change the file size. We set the `O_SYNC` flag when we open the file, to ensure that the clients write data back to the NFS server instead of just caching it locally. This setup is similar to many I/O workloads in virtualized environments [45], which are big users of NFS.

Figure 9 shows the NFS random-write throughput for three different I/O sizes. The throughputs shown in Figure 9 are higher than commodity hard disks because we use a two-drive RAID-0 and our controller has a 256MB battery-backed writeback cache. We ran the experiments

long enough to ensure that the working sets, including in the 4KB I/O case, were much larger than 256MB. As expected, larger I/O sizes led to higher throughput because disk seeks were reduced.

In Figure 9, the number of NFS clients also influenced the throughput, although not as much as the I/O size. The influence is most obvious at 64KB: when we went from 1 to 2 single-threaded clients, the 64KB write throughput increased from 42.5MB/s to 62MB/s, which approaches our disk's maximum bandwidth of 66MB/s at that size. These numbers closely match calculations based on our 1Gbit/s network speed: a 64KB packet takes approximately 0.5ms to transmit, and at maximum throughput the disk latency is about 1ms. Thus, the single-client NFS throughput will be $64\text{KB}/1.5\text{ms} \approx 42\text{MB/s}$. Note that our calculation assumes that the disk latency is the same regardless of network delays. In reality, a 0.5ms network latency has at least two effects on disk latency: (1) The network latency reduces disk utilization, which increases the probability of missing disk revolutions and makes head scheduling less effective. (2) It reduces congestion, which decreases disk queueing latency. Although the first effect tends to lengthen the I/O latency, the second tends to shorten it.

When we increased to two clients, the disk became the bottleneck because it cannot write data faster than 66MB/s, but the network can transmit data for one client while the other is waiting for the disk. Thus, multiple NFS clients interleave network transmission and keep the disk busy, which leads to a NFS write throughput close to the disk's maximum capability.

When the I/O size was 4KB or 16KB, the impact of the number of clients on write throughput was smaller. The main reason is that the disk became 6× and 25× slower than the network when the I/O size shrank to 16KB and 4KB, respectively. Disk latency became the primary factor in write latency (86% in the 16KB case and 96% in the 4KB case). Therefore, adding clients did not significantly improve the throughput.

We also tried running the clients without setting `O_SYNC`, which generated a bursty workload to the NFS server. Clients initially realized high throughput (over 1GB/s) since all data was buffered in the client cache. Once the number of dirty pages in the caches passed a threshold, the throughput then dropped to zero as the NFS clients flushed the dirty pages to the server, which could take up to 5 minutes depending on the I/O size and the number of clients. After that, the write throughput became high again, and then repeated the same pattern.

6.2 Sequential Write

The NFS sequential-write workload has the same setup as the random-write one, except that the write offsets are chosen sequentially from the beginning to the end of the file. The I/O size is 64KB. We again set the `O_SYNC` flag when we open the file to ensure dirty data is immediately committed to disk.

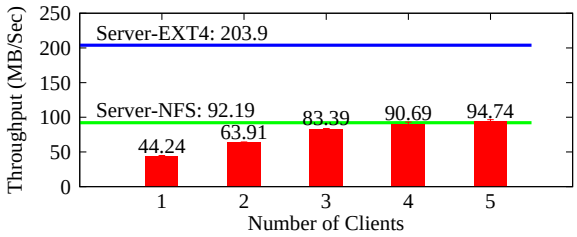


Figure 10: Sequential write throughput. When there are multiple NFS clients, the plotted throughput is the sum of all clients' throughputs. The blue line marks the throughput when we ran the workload directly on `ext4` without NFS. The green line marks the throughput when we ran the workload via NFS on the same machine the NFS server resides on, i.e., without any network traffic. `O_SYNC` is set. The kernel version is 3.12.0

The throughput results are shown in Figure 10. All throughputs were lower than 50% of the maximum sequential-write throughput of our disk. We initially suspected that the low throughputs were caused by network issues. However, the sequential-write throughput was only 92MB/s even when executed directly on the NFS server over the loopback device.

We found that the low sequential-write throughputs were actually caused by metadata updates and associated `jbd2` journaling. We used the default settings of `ext4`, so the data mode is *data=ordered*, which journals metadata but not file data. When we ran the workload directly on `ext4`, we observed only a negligible number of journal I/Os, and measured a throughput of 203.9MB/s, as shown in Figure 10. However, when we ran the same workload via NFS, we observed many 4KB journal I/Os. As an example, in the single-client case there were an average of 2.7 journal I/Os for every NFS write. The cause of this difference is the `O_SYNC` flag, which has different semantics on in `ext4` than in NFS. The POSIX semantics of `O_SYNC` require all metadata updates to be synchronously written to disk. On Linux, however, the `O_SYNC` flag is implemented so that only the actual file data and the metadata necessary to retrieve it are written synchronously. Since our workloads use only in-place writes, which update the file's modification time but not the block mapping, writing an `ext4` file introduces no journal I/Os. However, NFS implementation more strictly adheres to POSIX semantics by using the NFS `FILE_SYNC4` flag, which requires the server to commit both the written data and "all file system metadata" to stable storage before returning results. Therefore, sequential write workloads on NFS generate many synchronous journal I/Os in the exported server file system. (We also observed journal I/Os in the NFS random-write workload, but the effect of extra journal-

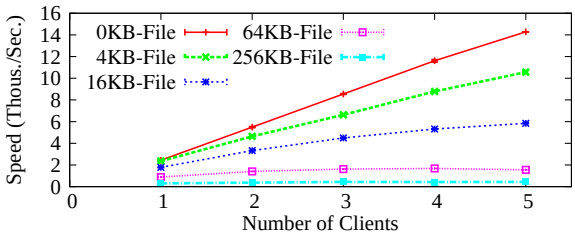


Figure 11: File creation speed. The kernel version is 3.12.0.

ing was less noticeable since performance was already limited by heavy seeking.)

■ **Observation 9:** *In Linux, the* `O_SYNC` *flag causes more metadata to be synchronously written to disk when used with NFS than with local file systems, reducing write performance by 50% or more for some workloads.*

In the sequential-write workload, the journal I/Os also disturb the sequentiality of disk I/O, causing additional seeks that lower write throughput. One way to avoid this is using a separate journal device. Also note that in Figure 10, the write throughput rose monotonically as we added clients. In addition to the increased client parallelism discussed above for random writes, journaling became more efficient because `jbd2` was able to group multiple atomic file updates into a single journal transaction. As the number of clients increase from 1 to 5, more could be grouped and the average number of updates per transaction monotonically increased from 19 to 53. Correspondingly, the ratio of journal to data I/Os decreased from 2.7 to 0.96. Thus, the cost of metadata journaling decreases as the number of clients increases.

We also tried the sequential-write workload without setting `O_SYNC`. The behavior was similar to that of the random-write workload without `O_SYNC`.

6.3 File Creation

Most of the workloads discussed so far are data-intensive, so they are more sensitive to network and I/O bandwidth than to latency. We now turn to a latency-sensitive workload. We chose file creation because it involves many metadata operations and we can easily control the latency sensitivity by adjusting the file size to change the relative weight of metadata operations.

We exported one directory via NFS and instructed the clients to create 100,000 files of a given size there, as fast as possible. Figure 11 shows the file creation speed when we varied the file size from 0KB to 256KB. The speed of file creation scaled well with the number of clients in the 0KB and 4KB cases. For these small sizes, metadata operations dominate the workload; neither network nor I/O bandwidth are the bottleneck. Larger files eventually reach a limit because they are constrained by network bandwidth.

As seen in Figure 11, creating smaller files was generally faster than creating larger ones. This is expected, since more data must be transmitted and written to disk. However, Figure 11 shows an anomaly: with only one client, creating 0KB files was no faster than creating 4KB ones. Creating a 0KB file requires two NFS operations: an OPEN and a following CLOSE. Creating a 4KB file also involves a WRITE. It is counterintuitive that the extra WRITE operation did not slow down performance. Our analysis reveals that the OPEN for 0KB files underwent a longer queueing delay (the latency from when the open was generated to when it was sent across the network) than the OPEN in the 4KB case. This extra delay was significant enough to compensate for the cost of the extra WRITE in the single-client, low-load case. As the number of clients increased, the system load rose and the WRITE delay became larger than the extra READ queueing delay. Thus, creating 0KB files became faster than creating 4KB files when there were multiple clients.

The extra queueing delay for the 0KB OPEN is caused by the TCP Nagle algorithm, which trades latency for bandwidth by coalescing multiple small packets. When the upper network layers queue a small outgoing packet, a TCP socket first waits in the hope of merging it with a future packet. The TCP Nagle algorithm reduces the average overhead of networking headers, which can be large in case of a lot of small messages. Since 0KB files required only OPEN and CLOSE, both of which are small, TCP Nagle waited unnecessarily. The extra 4KB WRITE was large and caused the preceding OPEN to be flushed, thereby improving performance.

We note that the socket API gives applications control over the TCP Nagle algorithm with the `SO_NODELAY` option. However, we cannot directly use this option in RPC because it influences the whole socket, not just particular messages sent through it. A possible workaround would be for RPC to reserve some TCP sockets with `SO_NODELAY` set, and use them for latency-sensitive messages. Alternatively, we could make TCP Nagle more adaptive, performing coalescing only when a high outgoing flow rate has been recently observed.

7 File Server Workload

This section discusses a general-purpose workload that is more complex than micro-workloads and more closely matches real-world workloads. Specifically, we study Filebench's File Server workload [19], which emulates multiple users working in their respective home directories. This workload is similar to that generated by SPECsfs [43]. Filebench creates a number of threads, each of which performs a sequence of operations including creates, deletes, appends, reads, writes, stats, and closes. The close operations ensure that dirty data is written back to the NFS server. We used Filebench's de-

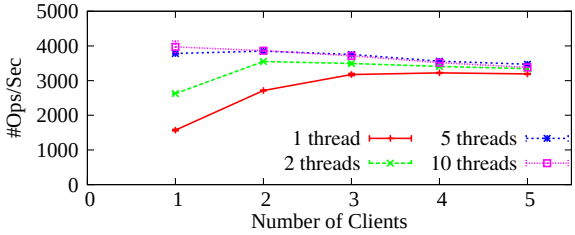


Figure 12: File Server performance. When there are multiple clients, the ops/s is the sum of ops/s across the clients. The kernel version is 3.12.0.

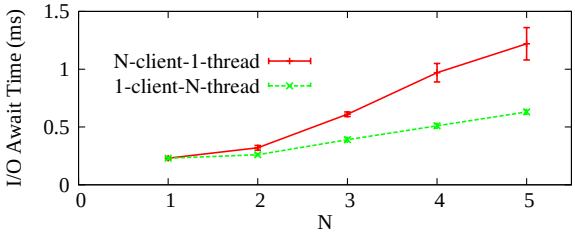


Figure 13: Average I/O wait time of File Server.

fault settings for the workload: each instance works with 10,000 files with an average size of 128KB. We ran the workloads for 5 minutes. The kernel version is 3.12.0.

To test NFSv4.1's scalability, we varied the number of threads and the number of NFS clients. To emulate multiple users on multiple NFS-supported workstations, we launched one instance of the File Server workload on each client.

Figure 12 shows the benchmarking results in terms of the number of File Server operations performed per second. Comparing the curves in Figure 12, the number of threads had a significant impact, although the difference became minor as the number of clients increased. The trends of the curves were also different. For the 1-thread and 2-thread cases, overall performance increased initially but dropped slightly as more clients were added, suggesting that the system quickly became overloaded. Because of the frequent close operations in the workload, large amounts of dirty data were regularly flushed to the disk. The disk quickly got saturated and became the performance bottleneck; eventually, the system began to thrash and performance dropped. This effect was seen sooner in the 5- and 10-thread cases, where even one client could generate dirty data faster than the disk's throughput, so the performance decreased monotonically as we added clients.

In Figure 12, adding clients did not increase performance as much as adding threads. The ops/s for one client running five threads was 18% higher than that for five clients running one thread each, because the I/O working set increased as we added clients, but not as we added threads. Recall that each client has one instance of File Server containing 10,000 files. A larger working

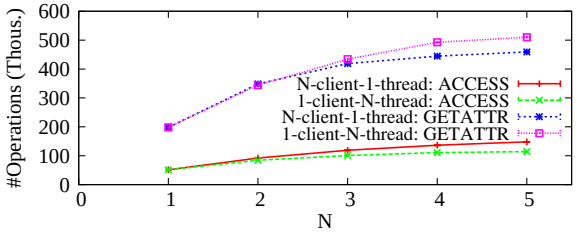


Figure 14: Number of NFSv4.1 ACCESS *and* GETATTR *operations of File Server.*

set increases the number and range of disk seeks. Figure 13, showing the wait time from `iostat`, confirmed this. (The I/O wait time includes both queuing and service time, which are correlated under heavy load. We do not report `iostat`'s `svctm` statistic because it is inaccurate and is scheduled to be removed [21].)

A larger working set (with more files and directories) also incurs more NFS metadata requests. Figure 14 lists the number of ACCESS and GETATTR requests—the two most frequent metadata requests—in the 1-client-N-thread cases and the N-client-1-thread cases. For any $x \in [2, 5]$, the x-client-1-thread case, notwithstanding its lower ops/s (in Figure 12), causes more ACCESS and GETATTR operations than the 1-client-x-thread case. Moreover, for cacheable content such as metadata and read-only file data, multiple threads in the same client need to request the content only once and share a common cache, whereas threads across different clients have to request it separately.

8 Related Work

The correctness and interoperability of the Linux NFS implementation are regularly tested by well known tools [2, 33, 34]. Its performance is usually benchmarked using either NFS-specific tools such as NFSometer [1] and SPECsfs [39, 43], or general file system tools such as Filebench [19], and IOzone [5]. Similarly, the performance problems of NFS can be identified using NFS-specific tools such as `nfsdump` and `nfsscan` [13], or general tools such as `tcpdump` [22] and `SystemTap` [36].

Although well-established macro-benchmark suites [1, 19, 43] provide convenient baselines for comparing performance among different systems, their results are frequently influenced by "obscure and seemingly tangential factors" [14]. Therefore, many researchers have used low-level micro-workloads to isolate, exercise, and improve specific aspects of NFS. Lever and Honeyman used a new sequential-write workload to measure and improve the Linux NFS client's write performance [28]. Ellard and Seltzer designed a simple sequential-read workload to benchmark and improve NFS's readahead performance [14]. Gulati et al. evaluated the performance of their NFSv4 proxy

using a locked-read micro-workload [23].

NFS has been widely deployed and studied. However, many prior studies [12, 14, 28, 39] were about NFSv2 and NFSv3. NFSv4, its latest major version, is less studied in the literature. Harrington et al. summarized major NFS contributors' efforts in testing the correctness and performance of Linux NFSv4 [2]. Radkov et al. compared the performance of NFSv4 and iSCSI in IP-networked storage [37]. Batsakis and Burns extended the NFSv4 delegation model to improve the performance and recoverability of NFS in computing clusters [3]. Gulati et al. built a NFSv4 cache proxy, also using delegations, to improve NFS performance in slow networks [23].

NFSv4.1 is the latest minor version of NFSv4 and its Linux implementation is still evolving [18]. To the best of our knowledge, there is no prior comprehensive benchmarking analysis of the Linux's NFSv4.1 implementation.

9 Conclusions

We have presented a benchmarking study of Linux's NFSv4.1 performance. We provided insightful analysis of the benchmarking results and identified the roots of performance problems disclosed by simple micro-workloads. We found that NFS's interactions with other OS components are complex and subtle. Specifically, we found that the aggressive readahead algorithm in the 2.6.32el6 kernel can reduce random-read performance by up to 80%. We identified a Bufferbloat-related Hash-Cast networking problem that causes unfairness among NFS clients. Hash-Cast influences not only NFS but any data-intensive TCP applications using a multi-queue NIC. We also noted that the TCP Nagle algorithm may hurt the performance of latency-sensitive NFS workloads. We found that writing NFS files with `O_SYNC` has a side effect on the journaling of `ext4`, which wastes more than 50% of disk write bandwidth in a sequential-write workload. We also showed that NFS delegation can save up to 90% of network traffic and significantly boost performance. However, it incurs a delay of at least 100ms in case of conflicts.

These findings are not specific to our hardware setup. The readahead algorithm, the `O_SYNC` effect, and the delegation mechanism are agnostic to network speed and storage media type, whereas the Hash-Cast problem and TCP Nagle algorithm are applicable to any distributed applications based on TCP.

Future work. We are benchmarking NFSv4.1 using more sophisticated macro-workloads in 10GbE networks with SSD as the storage backend. We plan to study the double readahead effect of NFS, where both the client and the server are doing readahead. We also plan to working on an efficient way to avoid the winner-loser pattern by using on-demand explicit load balancing of multiple `tx-queues`.

References

[1] Weston A. Adamson. NFSometer. `www.linux-nfs.org/wiki/index.php/NFSometer`.

[2] B. Harrington, A. Charbon, T. Reix, V. Roqueta, J. B. Fields, T. Myklebust, and S. Jayaraman. NFSv4 test project. In *Linux Simposium*, pages 115–134, 2006.

[3] A. Batsakis and R. Burns. Cluster delegation: High-performance, fault-tolerant data sharing in NFS. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*. IEEE, July 2005.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[5] D. Capps. IOzone file system benchmark. `www.iozone.org`.

[6] Ming Chen. nfsd: consider CLAIM_FH when handing out delegation. `https://lkml.org/lkml/2014/1/29/347`.

[7] Jonathan Corbet. Network transmit queue limits. `http://lwn.net/Articles/454390/`.

[8] Jonathan Corbet. TCP segmentation offloading. `http://lwn.net/Articles/9123/`.

[9] Eric Dumazet. TSO packets automatic sizing. `http://lwn.net/Articles/564979/`.

[10] M. Eisler, A. Chiu, and L. Ling. RPCSEC_GSS protocol specification. Technical Report RFC 2203, Network Working Group, September 1997.

[11] Mike Eisler. NFS version 4 and beyond, 2006. LISA talk.

[12] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003. USENIX Association.

[13] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, San Diego, CA, October 2003. USENIX Association.

[14] D. Ellard and M. Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003. USENIX Association.

[15] Sorin Faibish. NFSv4.1 and pNFS ready for prime time deployment, 2011.

[16] John Fastabend. Qdisc experiments at 10gbps. In *Linux Plumbers Conference*, San Diego, CA, August 2012.

[17] John Fastabend and Eric Dumazet. [BUG?] ixgbe: only num_online_cpus() of the tx queues are enabled, 2014. http://comments.gmane.org/gmane.linux.network/307532.

[18] Bruce Fields. NFSv4.1 server implementation. http://goo.gl/vAqR0M.

[19] Filebench. http://filebench.sf.net.

[20] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the Internet. *Commun. ACM*, 55(1):57–65, 2012.

[21] Sebastien Godard. iostat. http://linux.die.net/man/1/iostat.

[22] LBNL Network Research Group. The TCPDump/Libpcap site. www.tcpdump.org, February 2003.

[23] A. Gulati, M. Naik, and R. Tewari. Nache: Design and Implementation of a Caching Proxy for NFSv4. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 199–214, San Jose, CA, February 2007. USENIX Association.

[24] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of MSST*, Monterey, CA, 2005. IEEE.

[25] Yury Izrailevsky. WebNFS Report. http://goo.gl/wLYiUn, November 2009.

[26] R. J. La, J. Walrand, and V. Anantharam. *Issues in TCP Vegas*. Electronics Research Laboratory, College of Engineering, University of California, 1999.

[27] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, pages 213–226, Boston, MA, June 2008. USENIX Association.

[28] C. Lever and P. Honeyman. Linux NFS Client Write Performance. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 29–40, Monterey, CA, June 2002. USENIX Association.

[29] Chuck Lever. Close-to-open cache consistency in the linux NFS client. http://goo.gl/o9i0MM.

[30] Sam Liang and David Cheriton. TCP-RTM: Using TCP for real time multimedia applications. In *International Conference on Network Protocols*, 2002.

[31] Alex McDonald. The background to NFSv4.1. *;login: The USENIX Magazine*, 37(1):28–35, February 2012.

[32] Paul E. McKenney. Stochastic fairness queueing. In *INFOCOM'90*, pages 733–740. IEEE, 1990.

[33] Sun Microsystems. Cthon 2004 Test Suite. www.connectathon.org/nfstests.html, 2008.

[34] Jorge Mora. Nfstest. http://goo.gl/7QBl7O.

[35] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, 2008.

[36] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Linux Symposium*, pages 49–64, Ottawa, Canada, July 2005. Linux Symposium.

[37] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy. A performance comparison of NFS and iSCSI for IP-networked storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 101–114, San Francisco, CA, March/April 2004. USENIX Association.

[38] Scott Rixner. Network virtualization: Breaking the performance barrier. *Queue*, 6(1):37:36–37:ff, Jan 2008.

[39] D. Robinson. The advancement of NFS benchmarking: SFS 2.0. In *Proceedings of the 13th USENIX Systems Administration Conference (LISA '99)*, pages 175–185, Seattle, WA, November 1999. USENIX Association.

[40] S. Shepler and M. Eisler and D. Noveck. NFS Version 4 Minor Version 1 Protocol. Technical Report RFC 5661, Network Working Group, January 2010.

[41] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[42] Dan Siemon. Queueing in the Linux network stack. *Linux Journal*, July 2013.

[43] SPEC. SPECsfs2008. www.spec.org/sfs2008, 2008.

[44] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Network Working Group, March 1989.

[45] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2013. USENIX Association.