NFS Tricks and Benchmarking Traps

Daniel Ellard, Margo Seltzer
Harvard University
{ellard,margo}@eecs.harvard.edu

Abstract

We describe two modifications to the FreeBSD 4.6 NFS server to increase read throughput by improving the read-ahead heuristic to deal with reordered requests and stride access patterns. We show that for some stride access patterns, our new heuristics improve end-to-end NFS throughput by nearly a factor of two. We also show that benchmarking and experimenting with changes to an NFS server can be a subtle and challenging task, and that it is often difficult to distinguish the impact of a new algorithm or heuristic from the quirks of the underlying software and hardware with which they interact. We discuss these quirks and their potential effects.

1 Introduction

Despite many innovations, file system performance is steadily losing ground relative to CPU, memory, and even network performance. This is due primarily to the improvement rates of the underlying hardware. CPU speed and memory density typically double every 18 months, while similar improvements in disk latency have taken the better part of a decade. Disks do keep pace in terms of total storage capacity, and to a lesser extent in total bandwidth, but disk latency has become the primary impediment to total system performance.

To avoid paying the full cost of disk latency, modern file systems leverage the relatively high bandwidth of the disk to perform long sequential operations asynchronously and amortize the cost of these operations over the set of synchronous operations that would otherwise be necessary. For write operations, some techniques for doing this are log-structured file systems [18], journalling, and soft updates [21]. For reading, the primary mechanism is *read-ahead* or *prefetching*. When the file system detects that a process is reading blocks from a file in a predictable pattern, it may optimistically read blocks that it anticipates will be requested soon. If the blocks are arranged sequentially on disk, then these "extra" reads can be performed relatively efficiently because the incremental cost of reading additional contiguous blocks is small. This technique can be beneficial even when the disk blocks are not adjacent, as shown by Shriver *et al.* [23].

Although there has been research in detecting and exploiting arbitrary access patterns, most file systems do not attempt to recognize or handle anything more complex than simple sequential access – but because sequential access is the common case, this is quite effective for most workloads. The Fast File System (FFS) was the pioneering implementation of these ideas on UNIX [12]. FFS assumes that most file access patterns are sequential, and therefore attempts to arrange files on disk in such a way that they can be read via a relatively small number of large reads, instead of block by block. When reading, it estimates the sequentiality of the access pattern and, if the pattern appears to be sequential, performs read-ahead so that subsequent reads can be serviced from the buffer cache instead of from disk.

In an earlier study of NFS traffic, we noted that many NFS requests arrive at the server in a different order than originally intended by the client [8]. In the case of read requests, this means that the sequentiality metric used by FFS is undermined; read-ahead can be disabled by a small percentage of out-of-order requests, even when the overall access pattern is overwhelmingly sequential. We devised two sequentiality metrics that are resistant to small perturbations in the request order. The first is a general method and is described in our earlier study. The second, which we call *SlowDown*, is a simplification of the more general method that makes use of the existing FFS sequentiality metric and read-ahead code as the basis for its implementation. We define and benchmark this method in Section 6.

The fact that the computation of the sequentiality metric is isolated from the rest of the code in the FreeBSD NFS server implementation provides an interesting testbed for experiments in new methods to detect access patterns. Using this testbed, we demonstrate a new algorithm for detecting sequential subcomponents in a simple class of regular but non-sequential read access patterns. Such access patterns arise when there is more than one reader concurrently reading a file, or when there is one reader accessing the file in a "stride" read pattern. Our algorithm is described and benchmarked in Section 7.

Despite the evidence from our analysis of several long-term NFS traces that these methods would enhance read performance, the actual benefit of these new algorithms proved quite difficult to quantify. In our efforts to measure accurately the impact of our changes to the system, we discovered several other phenomena that interacted with the performance of the disk and file system in ways that had far more impact on the overall performance of the system than our improvements. The majority of this paper is devoted to discussing these effects and how to control for them. In truth, we feel that aspects of this discussion will be more interesting and useful to our audience than the description of our changes to the NFS server.

Note that when we refer to NFS, we are referring only to versions 2 (RFC 1094) and 3 (RFC 1813) of the NFS protocol. We do not discuss NFS version 4 (RFC 3010) in this paper, although we believe that its performance will be influenced by many of the same issues.

The rest of this paper is organized as follows: In section 2 we discuss related work, and in Section 3, we give an overview of file system benchmarking. We describe our benchmark and our testbed in Section 4. In Section 5, we discuss some of the properties of modern disks, disk scheduling algorithms, and network transport protocols that can disrupt NFS benchmarks. We return to the topic of optimizing NFS read performance via improved read-ahead, and define and benchmark the *SlowDown* heuristic in Section 6. Section 7 gives a new cursor-based method for improving the performance of stride access patterns and measures its effectiveness. In Section 8, we discuss plans for future work and then conclude in Section 9.

2 Related Work

NFS is ubiquitous in the UNIX world, and therefore has been the subject of much research.

Dube *et al.* discuss the problems with NFS over wireless networks, which typically suffer from packet loss and reordering at much higher rates than our switched Ethernet testbed [6]. We believe that our *SlowDown* heuristic would be effective in this environment.

Much research on increasing the read performance of NFS has centered on increasing the effectiveness of client-side caching. Dahlin *et al.* provide a survey of several approaches to cooperative client-side caching and analyze their benefits and tradeoffs [4]. The NQNFS (Not Quite NFS) system grants short-term leases for cached objects, which increases performance by reducing both the quantity of data copied from the server to the client and the number of NFS calls that the client makes to check the consistency of their cached copies [10]. Unfortunately, adding such constructs to NFS version 2 or NFS version 3 requires non-standard changes to the protocol. The NFS version 4 protocol does include a standard protocol for read leases, but has not achieved widespread deployment.

Another approach to optimizing NFS read performance is reducing the number of times the data buffers are copied. The zero-copy NFS server shows that this technique can double the effective read throughput for data blocks that are already in the server cache [11].

In contrast to most previous work in NFS read performance, we focus on improving read performance for uncached data that must be fetched from disk. In this sense, our work is more closely related to the studies of read-ahead [23] and the heuristics used by FFS [13].

3 Benchmarking File Systems and I/O Performance

There has been much work in the development of accurate workload and micro benchmarks for file systems [22, 24]. There exists a bewildering variety of benchmarks, and there have been calls for still more [16, 19, 27].

Nearly all benchmarks can be loosely grouped into one of two categories: *micro* benchmarks, such as lmbench [14] or `bonnie` [1], which measure specific low-level aspects of system performance such as the time required to execute a particular system call, and *macro* or *workload* benchmarks, such as the SPEC SFS [26] or the Andrew [9] benchmarks, which estimate the performance of the system running a particular workload. Workload benchmarks are used more frequently than micro benchmarks in the OS research literature because there are several that have become "standard" and have been used in many analyses, and therefore provide a convenient baseline with which to compare contemporary systems and new techniques. There are two principal drawbacks to this approach, however – first, the benchmark workloads may have little or no relevance to the workload current systems actually run, and second, the results of these holistic benchmarks can be heavily influenced by obscure and seemingly tangential factors.

For our analysis, we have created a pair of simple micro benchmarks to analyze read performance. A micro benchmark is appropriate for our research because our goal is to isolate and examine the effects of changes to the read-ahead heuristics on read performance. The benchmark used in most of the discussions of this paper is defined in Section 4.2. The second benchmark is defined in Section 7. Neither of these benchmarks resembles a general workload – their only purpose is to measure the raw read performance of large files.

We do not attempt to age the file system at all before we run our benchmarks. Aging a file system has been shown to make benchmark results more realistic [24]. For most benchmarks, fresh file systems represent the best possible case. For our enhancements, however, fresh file systems are one of the worst cases. We are attempting to measure the impact of various read-ahead heuristics, and we believe that read-ahead heuristics increase in importance as file systems age. Therefore, any benefit we see for a fresh file system should be even more pronounced on an aged file system.

4 The Testbed

In this section, we describe our testbed, including the hardware, our benchmark, and our method for defeating the effects of caching.

4.1 The Hardware

The server used for all of the benchmarks described in this paper is a Pentium III system running at 1 GHz, with 256 MB of RAM. The system disk is an IBM DDYS-T36950N S96H SCSI-3 hard drive controlled by an Adaptec 29160 Ultra160 SCSI adapter. The benchmarks are run on two separate disks: a second IBM DDYS-T3690N S96H drive (attached to the Adaptec card), and a Western Digital WD200BB-75CAA0 drive (attached to a VIA 82C686 ATA66 controller on the motherboard). The server has two network interfaces: an Intel PRO/1000 XT Server card running at 1 Gb/s and an 3Com 3c905B-TX Fast Etherlink XL card running at 100 Mb/s.

The clients are Pentium III systems running at 1 GHz, with 1 Gigabyte of RAM, and two network interfaces: an Intel PRO/1000 XT Server card running at 1 Gb/s, and an Intel Pro 10/100B/100+ Ethernet card, running at 100 Mb/s. The 100Mb/s interfaces are for general use, while the 1Gb/s interfaces are only used by the benchmarks.

The gigabit Ethernet cards are connected via a NetGear GSM712 copper gigabit switch. The switch and the Ethernet cards use 802.3x flow control, and the standard Ethernet MTU of 1500 bytes. The raw network bandwidth achievable by the server via TCP over the gigabit network is 49 MB/s. This falls far short of the theoretical maximum, but approaches the DMA speed of the PCI bus of the server motherboard, which we measured at 54 MB/s using the `gm_debug` utility provided by Myrinet to test their network cards. (A Myrinet card was installed in the server long enough to run this benchmark, but was not in the system for the other tests.) Only the benchmark machines are attached to the gigabit switch.

All systems under test run FreeBSD 4.6.2. The FreeBSD kernel was configured to remove support for pre-686 CPUs and support for hardware devices not present in our configuration, but we made no other customizations or optimizations beyond what is explicitly described in later sections of the paper. For all tests, the server runs eight `nfsds` instead of the default four, and the clients run eight `nfsiods` instead of the default four.

4.2 The Benchmark

The aspect of system performance that we wish to measure is the sustained bandwidth of concurrent read operations; the sustained bandwidth we can obtain *from disk* via the file system when several processes concurrently read sequentially through large files. In this section we describe the simple benchmark that we have designed for this purpose. Although it is simple, our benchmark illustrates the complexity of tuning even simple behaviors of the system.

4.3 Running the Benchmark

Before running the benchmark, we create a testing directory and populate it with a number of files: one 256 MB file, two 128 MB files, four 64 MB files, eight 32 MB files, sixteen 16 MB files, and thirty-two 8 MB files. We fill every block in these files with non-zero data, to prevent the file system from optimizing them as sparse files.

The benchmark loops through several different numbers of concurrent readers:

For each n (1, 2, 4, 8, 16, 32)

- For each file of size $256/n$ MB, create a reader process to read that file. This process opens the file, reads through it from start to end, and then closes the file. Start all these processes running concurrently.
- Wait until the last reader process has finished. Record the time taken by each reader. The number of MB read divided by the time required for the last reader to finish gives the effective throughput of the file system.

During the first iteration, a single reading process will be created, which will read through the 256 MB file. In the second iteration, two reading processing will concurrently read through different 128 MB files. In the final iteration, 32 reading processes will concurrently read different 8 MB files.

For all of the timed benchmark results shown in this paper, each point represents the average of at least ten separate runs. Unless otherwise mentioned, the standard deviation for each set of runs is less than 5% of the mean (and typically much less).

4.3.1 Defeating the Cache

We wish to benchmark the speed that the file system can pull data from the disk (either explicitly or via read-ahead). To ensure that we measure this (instead of memory bandwidth), we must make sure that the data we read are not already in the cache.

For our particular setup, every set of files contains 256 MB, so a complete iteration of the benchmark requires reading 1.5 GB. Because our clients have 1 GB of RAM and the server has only 256 MB of RAM and files are not re-read until 1.25 GB of other data has been read, none of the data will survive in the cache long enough to have an effect, at least with our current OS. If our clients and servers engaged in cooperative caching, or used a caching mechanism intelligent enough to recognize the cyclic nature of our benchmarks, we would have to take more care to ensure that none of the data are read from cache. Other techniques for ensuring that the data are flushed from the cache include rebooting each client and server between each iteration of the benchmark, unmounting and remounting the file systems used by the benchmark, and reading large amounts of unrelated data until they fill the cache. We experimented with each of these and found that they made no difference to the benchmark results, so we are confident that caching did not influence our benchmarking.

5 Benchmarking Traps

Our initial attempt to measure the effect of changes to the FreeBSD NFS server and to experiment with heuristics designed to address some of the behaviors we observed in our earlier NFS trace study was frustrating. Our algorithms were easy to implement, and the diagnostic instrumentation we added to monitor our algorithms confirmed that they were working as intended. However, the results from our benchmarks were confusing and sometimes contradictory. Different runs on the same hardware could give very different results, and the results on different hardware were often puzzling. We had anticipated that the effect of our changes would be relatively small, but we had not expected it to be overwhelmed by unexpected effects. Therefore, before proceeding with the benchmarks, we decided that a more interesting course of action would be to investigate and expose the causes of the variation in our benchmark and see if we could design our experiments to control for them.

5.1 ZCAV Effects

Modern disk drives, with few exceptions, use a technique known as zoned constant angular velocity coding (ZCAV) to increase the total disk capacity and average transfer rate [15]. ZCAV can be thought of as an approximation of constant linear density coding, modified to allow the disk to spin at a constant rate and provide an integral number of disk sectors per track. The innermost cylinders of the disk drive contain fewer sectors than the outermost tracks (typically by a factor of 2:3, but for some drives as much as 1:2). The transfer rate between the disk and its buffer varies proportionally; in the time it takes to perform a single revolution, the amount of data read or written to disk can vary by a factor of almost 2 depending on whether the head is positioned at the innermost or outermost track. If the transfer rate between the disk drive and the host memory is greater or equal to the internal transfer rate of the disk at the innermost track, then the effect of the differences in transfer rate will be visible to the host. For contemporary SCSI and IDE drives and controllers, which have host interface bandwidth exceeding their maximum read/write speeds, this difference is entirely exposed. If the proportion between the number of sectors in the innermost and outermost cylinders is 2:3, then reading a large file from the outermost cylinders will take only two-thirds of the time (and two-thirds of the number of seeks, since each track at the outside of the disk contains more sectors).

This effect has been measured and analyzed and is the basis of some well-studied techniques in file system layout tuning [15, 28]. Beyond papers that explicitly discuss methods for measuring and exploiting disk properties, however, mention of this effect is rare.

The ZCAV effect can skew benchmark results enormously, depending on the number of files created and accessed during the benchmark. If two independent runs of the same benchmark use two different sets of files, then the physical location of these files on disk can have a substantial impact on the benchmark. If the effect that the benchmark is attempting to measure is subtle, then it may be completely overwhelmed by the ZCAV effect. It may require a daunting number of runs to statistically separate the effect being measured from the noise intro-

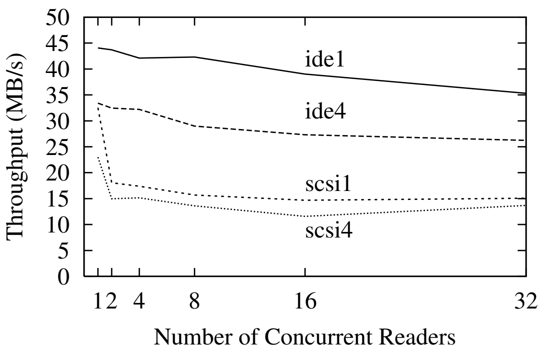

Figure 1: The ZCAV effect. The `scsi1` and `ide1` partitions use the outermost cylinders of the disk, while the `scsi4` and `ide4` use the innermost. As a result, the transfer rates for the `scsi1` and `ide1` partitions are higher than `scsi4` and `ide4`.

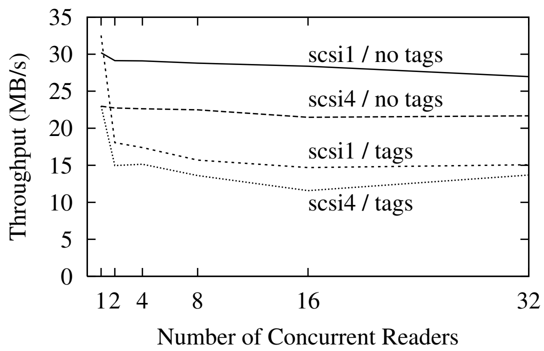

Figure 2: The effect of using tagged queues on SCSI performance. On our test system, disabling tagged queues improves transfer rates substantially for concurrent processes performing long sequential reads.

duced by using blocks from different areas of the disk.

The ZCAV effect is illustrated in Figure 1, which shows the results of running the same benchmark on different areas of a disk. For our benchmarks, we have divided each of our test disks into four partitions of approximately equal size, numbered 1 through 4. The files in tests `scsi1` and `ide1` are positioned in the outer cylinders of the SCSI and IDE drives, respectively, while the `scsi4` and `ide4` test files are placed in the inner cylinders. For both disks, it is clear that ZCAV has a strong effect. The effect is clearly pronounced for the IDE drive. The SCSI drive shows a weaker effect – but as we will see in Section 5.2, this is because there is another effect that obscures the ZCAV effect for simple benchmarks on our SCSI disk. For both drives the ZCAV effect is more than enough to obscure the impact of any small change to the performance of the file system.

The best method to control ZCAV effects is to use the largest disk available and run your benchmark in the smallest possible partition (preferably the outermost partition). This will minimize the ZCAV effect by minimizing the difference in capacity and transfer rate between the longest and shortest tracks used in your benchmark.

5.2 Tagged Command Queues

One of the features touted by SCSI advocates is the availability of *tagged command queues* (also known as *tagged queues*). This feature permits the host to send several disk operation requests to the disk and let the disk execute them asynchronously and in whatever order it deems appropriate. Modern SCSI disks typically have an internal command queue with as many as 256 entries. Some recent IDE drives support a feature conceptually identical to tagged command queues, but our IDE drive does not.

The fact that the tagged command queue allows the disk to reorder requests is both a boon for ordinary users and a source of headaches for system researchers. With tagged queues enabled, the disk may service its requests in a different order than they arrive at the disk, and its heuristics for reordering requests may be different from what the system researcher desires (or expects). For example, many disks will reorder or reschedule requests in order to reduce the total power required to position the disk head. Some disks even employ heuristics to reduce the amount of audible noise they generate – many users would prefer to have a quiet computer than one that utilizes the full positioning speed of the disk. The same model of disk drive may exhibit different performance characteristics depending on the firmware version, and on whether it is intended for a desktop or a server.

The SCSI disk in our system supports tagged queues, and the default FreeBSD kernel detects and uses them. We instrumented the FreeBSD kernel to compare the order that disk requests are sent to the disk to the order in which they are serviced, and found that when tagged queues are disabled, the two orders are the same, but when tagged queues are enabled, the disk does reorder requests. Note that this instrumentation was disabled during our timed benchmarks.

To explore the interaction between the FreeBSD disk scheduler and the disk's scheduler, we ran a benchmark with the tagged queues disabled. The results are shown in Figure 2. For our benchmark, the performance is significantly increased when tagged queues are disabled.

When tagged queues are enabled, the performance for the default configuration has a dramatic spike for the single-reader case, but then quickly falls away for multiple readers. With the tagged command queue disabled, however, the throughput for multiple concurrent readers decreases slowly as the number of readers increases, and is almost equal to the spike for the single reader in the default configuration. For example, the throughput for `scsi1` levels off just above 15 MB/s in the default configuration, but barely dips below 27 MB/s when tagged command queues are disabled.

There is no question that tagged command queues are effective in many situations. For our benchmark, however, the kernel disk scheduler makes better use of the disk than the on-disk scheduler. This is undoubtedly due in part to the fact that the geometry the disk advertises to the kernel does, in fact, closely resemble its actual geometry. This is not necessarily the case – for example, it is not the case for many RAID devices or similar systems that use several physical disks or other hardware (perhaps distributed over a network) to implement one logical disk. In a hardware implementation of RAID, an access to a single logical block may require accessing several physical blocks whose addresses are completely hidden from the kernel. In the case of SAN devices or storage devices employing a dynamic or adaptive configuration the situation is even more complex; in such devices the relationship between logical and physical block addresses may be arbitrary, or even change from one moment to the next [29]. For these situations it is better to let the device schedule the requests because it has more knowledge than the kernel.

Even for ordinary single-spindle disks, there is a small amount of re-mapping due to bad block substitution. This almost always constitutes a very small fraction of the total number of disk blocks and it is usually done in such a manner that it has a negligible effect on performance.

5.3 Disk Scheduling Algorithms

The FreeBSD disk scheduling algorithm, implemented in the `bufqdisksort` function, is based on a cyclical variant of the SCAN or elevator scan algorithm, as described in the BSD 4.4 documentation [13]. This algorithm can achieve high sustained throughput, and is particularly well suited to the access patterns created by the FFS read-ahead heuristics. Unfortunately, if the CPU can process data faster than the I/O subsystem can deliver it, then this algorithm can create unfair scheduling. In the worst case, imagine that the disk head is positioned at the outermost cylinder, ready to begin a scan inward, and the disk request queue contains two requests: one for a block on the outermost cylinder, requested by process α, and another for a block on the innermost cylinder, requested by process β. The request for the first block is satisfied, and α immediately requests another block in the same cylinder. If α requests a sequence of blocks that are laid out sequentially on disk, and does so faster than the disk can reposition, its requests will continue to be placed in the disk request queue before the request made by β. In the worst case, β may have to wait until α has scanned the entire disk. Somewhat perversely, an optimal file system layout greatly increases the probability of long sequential disk accesses, with a corresponding increase in the probability of unfair scheduling of this kind.

This problem can be reduced by the use of tagged command queues, depending on how the on-disk scheduler is implemented. In our test machine, the on-board disk scheduler of the SCSI disks is in effect more fair than the FreeBSD scheduler. In this example, it will process β's request before much time passes.

The unfairness of the elevator scan algorithm is also somewhat reduced by the natural fragmentation of file systems that occurs over time as files are added, change size, or are deleted. Although FFS does a good job of reducing the impact of fragmentation, this effect is difficult to avoid entirely.

The primary symptom of this problem is a large variation in time required by concurrent readers, and therefore this behavior is easily visible in the variance of the run times of each subprocess in our simple benchmark. Each subprocess starts at the same time, and reads the same amount of data, so intuition suggests that they will all finish at approximately the same time. This intuition is profoundly wrong when the default scheduler is used. Figure 3 illustrates the distribution of the individual process times for runs for the benchmark that runs eight concurrent processes, each reading a different 32 MB file. Note that the cache is flushed after each run. The plot of the time required to complete 1 through 8 processes using the elevator scan scheduler on the `ide1` partition shows that the average time required to complete the first process is 1.04 seconds, while the second finishes in 1.98 seconds, the third in 2.94, and so on until the last job finishes after an average of 5.97 seconds. With tagged queues disabled, a similar distribution holds for the `scsi1` partition, ranging from 1.18 through 8.54 seconds, although the plot for `scsi1` is not as straight as that for `ide1`. The difference between the time required by the fastest and slowest jobs is almost a factor 6 for `ide1`, and even higher for `scsi1`.

N-step CSCAN (N-CSCAN) is a fair variation of the Elevator scheduler that prohibits changes to the schedule for the current scan – in effect, it is always planning

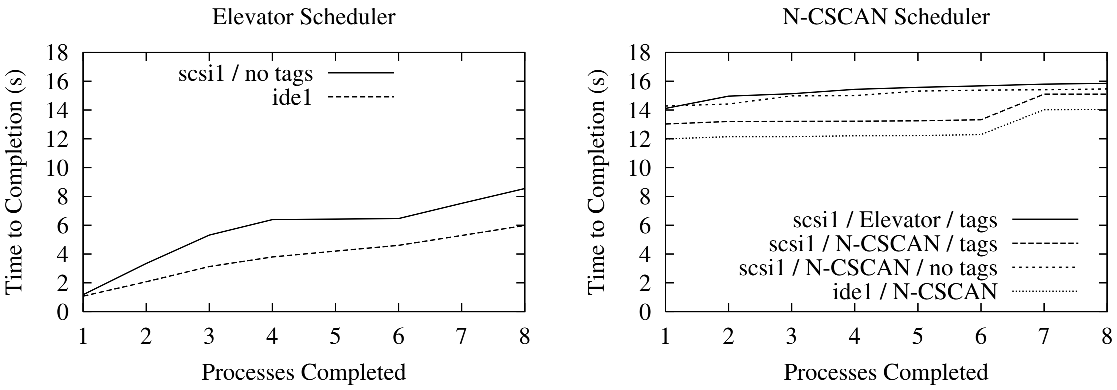

Figure 3: The interaction between tagged queues, the disk scheduling algorithm, and the distribution of average time required to complete a given number of processes. In each run, eight processes are started at the same time, and each reads the same amount of data. Each point on these plots represents the average of 34 runs. Processes run more quickly with the Elevator scan, but the last process takes 6-7 times longer to complete than the first. With the N-CSCAN scheduler, there is less variation in the distribution of running times, but all of the jobs are much slower.

the schedule for the *next* scan [5]. The resulting scheduler is fair in the sense that the expected latency of each disk operation is proportional to the length of the request queue at the time the disk begins its next sweep. Only a small patch is needed to change the current FreeBSD disk scheduler to N-CSCAN. We have implemented this change, along with a switch that can be used to toggle at runtime which disk scheduling algorithm is in use. As illustrated again in Figure 3, this dramatically reduces the variation in the run times for each reader process: for both `ide1` and `scsi1`, the difference in elapsed time between the slowest and the fastest readers is less than 20%.

Unfortunately, fairness comes at a high price: although all of the reading processes make progress at nearly the same rate, the overall average throughput achieved is less than half the bandwidth delivered by the unfair elevator algorithm. In fact, for these two cases, the *slowest* reading process for the elevator scan algorithm requires approximately 50% less time to run than the *fastest* reading process using the N-step CSCAN algorithm. For this particular case, it is hard to argue convincingly in favor of fairness. In the most extreme case, however, it is possible to construct a light workload that causes a process to wait for several minutes for the read of a single block to complete. As a rule of thumb, it is unwise to allow a single read to take longer than it takes for a user to call the help desk to complain that their machine is hung. At some point human factors can make a fair division of file system bandwidth as important as overall throughput.

Also shown in Figure 3 is the impact of these disk scheduling algorithms on `scsi1` when the tagged command queue is enabled. As described in Section 5.2, the on-disk tagged command queue can override many of the scheduling decisions made by the host disk scheduler. In this measurement, the on-disk scheduling algorithm appears to be fairer than N-step CSCAN (in terms of the difference in elapsed time between the slowest and fastest processes), but even worse in terms of overall throughput.

Although the plots in Figure 3 are relatively flat, they still exhibit an interesting quirk – there is a notable jump between the mean run time of the sixth and seventh processes to finish for N-CSCAN for `ide1` and `scsi1` with tagged queues disabled. We did not investigate this phenomenon.

The tradeoffs between throughput, latency, fairness and other factors in disk scheduling algorithms have been well studied and are still the subject of research [3, 20]. Despite this research, choosing the most appropriate algorithm for a particular workload is a complex decision, and apparently a lost art. We find it disappointing that modern operating systems generally do not acknowledge these tradeoffs by giving their administrators the opportunity to experiment and choose the algorithm most appropriate to their workload.

5.4 TCP vs UDP

SUN RPC, upon which the first implementation of NFS was constructed, used UDP for its transport layer, in part because of the simplicity and efficiency of the UDP protocol. Beginning with NFS version 3, however, many vendors began offering TCP-based RPC, including NFS

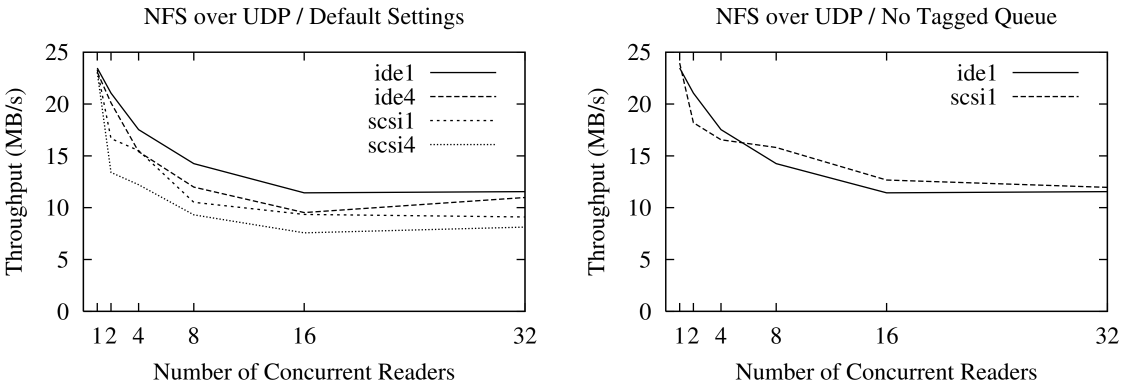

Figure 4: The speed of NFS over UDP, with and without tagged queues. Performance drops quickly as the number of concurrent readers increases. With tagged queues disabled, `scsi1` performance improves relative to `ide1` as the number of concurrent readers increases. Note that the ZCAV effect is still visible.

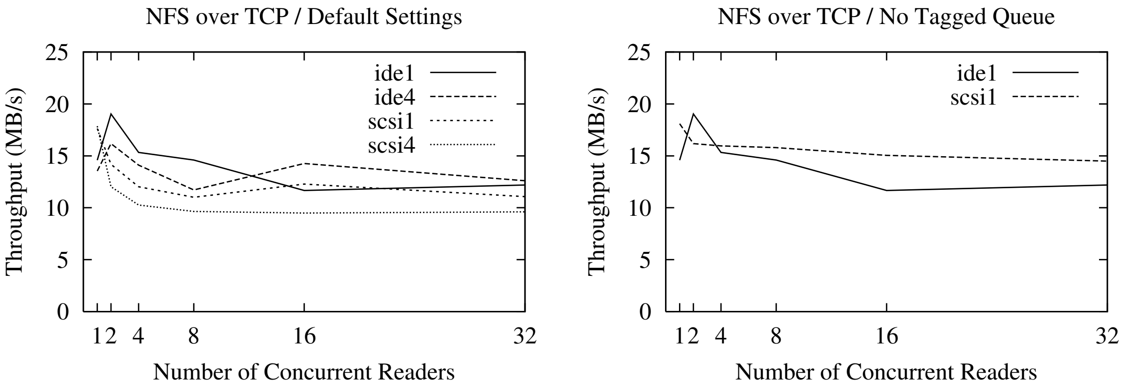

Figure 5: The speed of NFS over TCP, with and without tagged queues. Compared to UDP, the throughput is relatively constant as the number of concurrent readers increases, especially when tagged queues are disabled on the `scsi1`. We do not know why the IDE partitions show a performance spike at two concurrent readers, nor why `ide4` is faster than `ide1` for 16 readers.

over TCP. This has advantages in some environments, particularly WAN systems, due to the different characteristics of TCP versus UDP. UDP is a lightweight and connectionless datagram protocol. A UDP datagram may require several lower-level packets (such as Ethernet frames) to transmit, and the loss of any one of these packets will cause the entire datagram to be lost. In contrast, TCP provides a reliable connection-based mechanism for communication and can, in many cases, detect and deal with packet corruption, loss, or reordering more efficiently than UDP. TCP provides mechanisms for intelligent flow control that are appropriate for WANs.

On a wide-area network, or a local network with frequent packet loss or collision, TCP connections can provide better performance than UDP. Modern LANs are nearly always fully switched, and have very low packet loss rates, so the worst-case behavior of UDP is rarely observed. However, mixed-speed LANs do experience frequent packet loss at the junctions between fast and slow segments, and in this case the benefits of TCP are also worth considering. In our testbed, we have only a single switch, so we do not observe these effects in our benchmarks.

The RPC transport protocol used by each file system mounted via NFS is chosen when the file system is mounted. The default transport protocol used by `mount_nfs` is UDP. Many system administrators use `amd` instead of `mount_nfs`, however, and `amd` uses a different implementation of the mount protocol. On FreeBSD, NetBSD, and many distributions of GNU/Linux, `amd` uses TCP by default, but on other systems, such as OpenBSD, `amd` uses UDP. This choice can be overridden, but often goes unnoticed.

A comparison of the raw throughput of NFS for large reads over TCP and UDP is given in Figures 4 and 5. Compared to the performance of the local file system, shown in Figure 1, the throughput of NFS is disappointing; for concurrent readers the performance is about half that of the local file system and only a fraction of the potential bandwidth of the gigabit Ethernet. The throughput for small numbers of readers is substantially better for UDP than TCP, but the advantage of UDP is attenuated as the number of concurrent readers increases until it has no advantage over TCP (and in some cases is actually slower). In contrast, the throughput of accesses to the local disk slightly increases as the number of readers increases. We postulate that this is due to a combination of the queuing model used by the disk drive, and the tendency of the OS to perform read-ahead when it perceives that the access pattern is sequential, but also to throttle the read-ahead to a fixed limit. When reading a single file, a fixed amount of buffer space is set aside for read-ahead, and only a fixed number of disk requests are made at a time. As the number of open files increases, the total amount of memory set aside for read-ahead increases proportionally (until another fixed limit is reached) and the number of disk accesses queued up for the disk to process also grows. This allows the disk to be kept busier, and thus the total throughput can increase.

Unlike UDP, the throughput of NFS over TCP roughly parallels the throughput of the local file system, although it is always significantly slower, even over gigabit Ethernet. This leads to the question of why UDP and TCP implementations of NFS have such different performance characteristics as the number of readers increases – and whether it is possible to improve the performance of UDP for multiple readers.

5.5 Discussion

As illustrated in Figures 4 and 5, the effects of ZCAV and tagged queues are clearly visible in the NFS benchmarks. Even though network latency and the extra overhead of RPC typically reduces the bandwidth available to NFS to half the local bandwidth, these effects must be considered because they can easily obscure more subtle effects.

In addition to the effects we have uncovered here, there is a new mystery – the anomalous slowness of the `ide1` and `ide4` partitions when accessed via NFS over TCP by one reader. We suspect that this is a symptom of TCP flow control.

6 Improving Read-Ahead for NFS

Having now detailed some of the idiosyncrasies that we encountered with our simple benchmark, let us return to the task at hand, and evaluate the benefit of a more flexible sequentiality metric to trigger read-ahead in NFS.

The heuristics employed by NFS and FFS begin to break down when used on UDP-based NFS workloads because many NFS client implementations permit requests to be reordered between the time that they are made by client applications and the time they are delivered to the server. This reordering is due most frequently to queuing issues in the client `nfsiod` daemon, which marshals and controls the communication between the client and the server. This reordering can also occur due to network effects, but in our system the reorderings are attributable to `nfsiod`.

It must be noted that because this problem is due entirely to the implementation of the NFS client, a direct and pragmatic approach would be to fix the client to prevent request reordering. This is contrary to our research agenda, however, which focuses on servers. We are more interested in studying how servers can handle arbitrary and suboptimal client request streams than optimizing clients to generate request streams that are easier for servers to handle.

The fact that NFS requests are reordered means that access patterns that are in fact entirely sequential from the perspective of the client may appear, to the server, to contain some element of randomness. When this happens, the default heuristic causes read-ahead to be disabled (or diminished significantly), causing considerable performance degradation for sequential reads.

The frequency at which request reordering takes place increases as the number of concurrent readers, the number of `nfsiod`s, and the total CPU utilization on the client increases. By using a slow client and a fast server on a congested network, we have been able to create systems that reorder more than 10% of their requests for long periods of time, and during our analysis of traces from production systems we have seen similar percentages during periods of peak traffic. On our benchmark system, however, we were unable to exceed 6% request reordering on UDP and 2% on TCP on our gigabit network with anything less than pathological measures. This was slightly disappointing, because lower probabilities of request reordering translate into less potential for improvement by our algorithm, but we decided to press ahead and see whether our algorithm is useful in our situation and thus might be even more useful to users on less well-mannered networks.

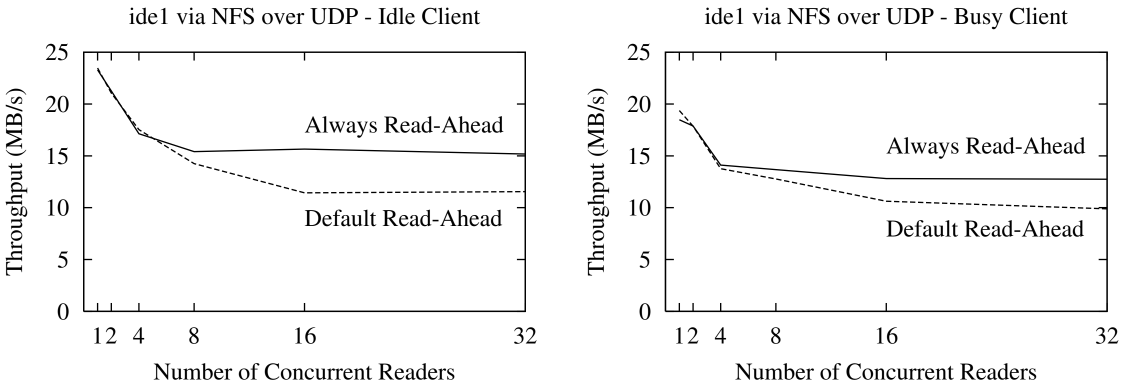

Figure 6: A comparison of the NFS throughput for the default read-ahead heuristic and a hard-wired "always do read-ahead" heuristic. All tests are run using file system `ide1` via UDP. The idle client is running only the benchmark, while the busy client is also running four infinite loop processes.

6.1 Estimating the Potential Improvement

Figure 6 shows the NFS throughput for the default implementation compared to the throughput when we hard-wire the sequentiality metric to always force read-ahead to occur. The difference between the "Always Read-ahead" and "Default Read-ahead" lines shows the potential improvement. In theory, for large sequential reads (such as our benchmarks) the NFS server should detect the sequential access pattern and perform read-ahead. As shown in Figure 6, however, for more than four concurrent readers the default and optimal lines diverge. This is due in part to the increased number of packet reorderings that occur when the number of concurrent readers increases, but it is also due to contention for system resources, as we will discuss in Section 6.3.

In our own experiments, we noticed that the frequency of packet reordering increases in tandem with the number of active processes on the client (whether those processes are doing any I/O or not), so Figure 6 also shows throughput when the client is running four "infinite loop" processes during the benchmark. Not surprisingly, the throughput of NFS decreases when there is contention for the client CPU (because NFS does have a significant processing overhead). Counter to our intuition, however, the gap between the "Always Read-ahead" line and the "Default Read-ahead" lines is actually smaller when the CPU is loaded, even though we see more packet reordering.

6.2 The *SlowDown* Sequentiality Heuristic

There are many ways that the underlying sequentiality of an access pattern may be measured, such as the metrics developed in our earlier studies of NFS traces. For our preliminary implementation, however, we wish to find a simple heuristic that does well in the expected case and not very badly in the worst case, and that requires a minimum of bookkeeping and computational overhead.

Our current heuristic is named *SlowDown* and is based on the idea of allowing the sequentiality index to rise in the same manner as the ordinary heuristic, but fall less rapidly. Unlike the default behavior (where a single out-of-order request can drop the sequentiality score to zero), the *SlowDown* heuristic is resilient to "slightly" out-of-order requests. At the same time, however, it does not waste read-ahead on access patterns that do not have a strongly sequential component – if the access pattern is truly random, it will quickly disable read-ahead. The default metric for computing the heuristic, as implemented in FreeBSD 4.x, is essentially the following: when a new file is accessed, it is given an initial sequentiality metric *seqCount* $= 1$ (or sometimes a different constant, depending on the context). Whenever the file is accessed, if the current offset *currOffset* is the same as the offset after the last operation (*prevOffset*), then increment *seqCount*. Otherwise, reset *seqCount* to a low value.

The *seqCount* is used by the file system to decide how much read-ahead to perform – the higher *seqCount* rises, the more aggressive the file system becomes. Note that in both algorithms, *seqCount* is never allowed to grow higher than 127, due to the implementation of the lower levels of the operating system.

The *SlowDown* heuristic is nearly identical in concept to the additive-increase/multiplicative-decrease used by TCP/IP to implement congestion control, although its application is very different. The initialization is the same as for the default algorithm, and when *prevOffset* matches *currOffset*, *seqCount* is incremented as before. When *prevOffset* differs from *currOffset*, however, the response of *SlowDown* is different:

- If *currOffset* is within 64k (eight 8k NFS blocks) of *prevOffset* then *seqCount* is unchanged.
- If *currOffset* is more than 64k from *prevOffset*, then divide *seqCount* by 2.

In the first case, we do not know whether the access pattern is becoming random, or whether we are simply seeing jitter in the request order, so we leave *seqCount* alone. In the second case, we want to start to cut back on the amount of read-ahead, and so we reduce *seqCount*, but not all the way to zero. If the non-sequential trend continues, however, repeatedly dividing *seqCount* in half will quickly chop it down to zero.

It is possible to invent access patterns that cause *SlowDown* to erroneously trigger read-ahead of blocks that will never be accessed. To counter this, more intelligence (requiring more state, and more computation) could be added to the algorithm. However, in our trace analysis we did not encounter any access patterns that would trick *SlowDown* to perform excessive read-ahead. The only goal of *SlowDown* is to help cope with small reorderings in the request stream. An analysis of the values of *seqCount* show that *SlowDown* accomplishes this goal.

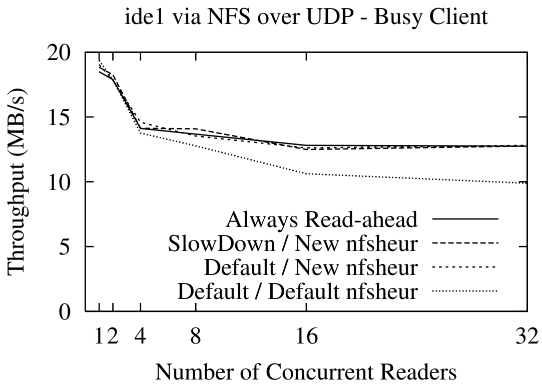

Figure 7: The effect of *SlowDown* and the new *nfsheur* table for sequential reads for disk `ide1` over UDP. The new *nfsheur* increases throughput for more than four concurrent readers, and gives performance identical to the Always Read-ahead case. *SlowDown* makes no further improvement.

6.3 Improving the *nfsheur* Table

NFS versions 2 and 3 are stateless protocols, and do not contain any primitives analogous to the `open` and `close` system calls of a local file system. Because of the stateless nature of NFS, most NFS server implementations do not maintain a table of the open file descriptors corresponding to the files that are active at any given moment. Instead, servers typically maintain a cache of information about files that have been accessed recently and therefore are believed likely to be accessed again in the near future. In FreeBSD, the information used to compute and update the sequentiality metric for each active file is cached in a small table named *nfsheur*.

Our benchmarks of the *SlowDown* heuristic showed no improvement over the default algorithm, even though instrumentation of the kernel showed that the algorithm was behaving correctly and updated the sequentiality metric properly even when many requests were reordered. We discovered that our efforts to calculate the sequentiality metric correctly were rendered futile because the *nfsheur* table was too small.

The *nfsheur* table is implemented as a hash table, using open hashing with a small and limited number of probes. If the number of probes necessary to find a file handle is larger than this limit, the least recently used file handle from among those probed is ejected and the new file handle is added in its place. This means that entries can be ejected from the table even when it is less than full, and in the worst case a small number of active files can thrash *nfsheur*. Even in the best case, if the number of active files exceeds the size of the table, active file handles will constantly be ejected from the table. When a file is ejected from the table, all of the information used to compute its sequentiality metric is lost.

The default hash table scheme works well when a relatively small number of files are accessed concurrently, but for contemporary NFS servers with many concurrently active files the default hash table parameters are simply too small. This is not particularly surprising, because network bandwidth, file system size, and NFS traffic have increased by two orders of magnitude since the parameters of the *nfsheur* hash table were chosen. For our *SlowDown* experiment, it is clear that there is no benefit to properly updating the sequentiality score for a file if the sequentiality score for that file is immediately ejected from the cache.

To address this problem, we enlarged the *nfsheur* table, and improved the hash table parameters to make ejections less likely when the table is not full. As shown in Figure 7, with the new table implementation *SlowDown* matches the "Always Read-ahead" heuristic. We were also surprised to discover that with the new table, the default heuristic *also* performs as well as "Always Read-Ahead". It is apparently more important to have an entry in *nfsheur* for each active file than it is for those entries to be completely accurate.

7 Improving Stride Read Performance

The conventional implementation of the sequentiality metric in the FreeBSD implementation of NFS (and in fact, in many implementations of FFS) uses a single descriptor structure to encapsulate all information about the observed read access patterns of a file. This can cause suboptimal read-ahead when there are several readers of the same file, or a single reader that reads a file in a regular but non-sequential pattern. For example, imagine a process that strides through a file, reading blocks $0,\ x,\ 1,\ x{+}1,\ 2,\ x{+}2,\ 3,\ x{+}3\ \cdots$. This pattern is the composition of two completely sequential read access patterns ($0,\ 1,\ 2,\ 3,\cdots$ and $x,\ x+1,\ x+2,\ x+3,\cdots$), each of which can benefit from read-ahead. Unfortunately, neither the default sequentiality metric nor *SlowDown* recognizes this pattern, and this access pattern will be treated as non-sequential, with no read-ahead. Variations on the stride pattern are common in engineering and out-of-core workloads, and optimizing them has been the subject of considerable research, although it is usually attacked at the application level or as a virtual memory issue [2, 17].

In the ordinary implementation, the *nfsheur* contains a single offset and sequentiality count for each file handle. In order to handle stride read patterns, we add the concept of *cursors* to the *nfsheur*. Each active file handle may have several cursors, and each cursor contains its own offset and sequentiality count. When a read occurs, the sequentiality metric searches the *nfsheur* for a matching cursor (using the same approximate match as *SlowDown* to match offsets). If it finds a matching cursor, the cursor is updated and its sequentiality count is used to compute the effective *seqCount* for the rest of the operation, using the *SlowDown* heuristic. If there is no cursor matching a given read, then a new cursor is allocated and added to the *nfsheur*. There is a limit to the number of active cursors per file, and when this limit is exceeded the least recently used cursor for that file is recycled.

If the access pattern is truly random, then many cursors are created, but their sequentiality counts do not grow and no extra read-ahead is performed. In the worst case, a carefully crafted access pattern can trick the algorithm into maximizing the sequentiality count for a particular cursor just before that cursor dies (and therefore potentially performing read-ahead for many blocks that are never requested), but the cost of reading the extraneous blocks can be amortized over the increased efficiency of reading the blocks that *were* requested (and caused the sequentiality count to increase in the first place).

The performance of this method for a small set of stride read patterns is shown in Figure 8 and Table 1. For the "2" stride for a file of length n, there are two sequential subcomponents, beginning at offsets 0 and $n/2$, for the "4" stride there are four beginning at offsets 0, $n/4$, $n/2$, and $3n/4$, and for the "8" stride there are eight sequential subcomponents beginning at offsets 0, $n/8$, $n/4$, $3n/8$, $n/2$, $5n/8$, $3n/4$, and $7n/8$. To the ordinary sequentiality or *SlowDown* heuristics, these appear to be completely random access patterns, but our cursor-based algorithm detects them and induces the proper amount of read-ahead for each cursor. As shown in this figure, the time required to read the test files using the cursor-based method is at least 50% faster than using the default method. In the most extreme case, the cursor-based method is 140% faster for the 8-stride reader on `ide1`.

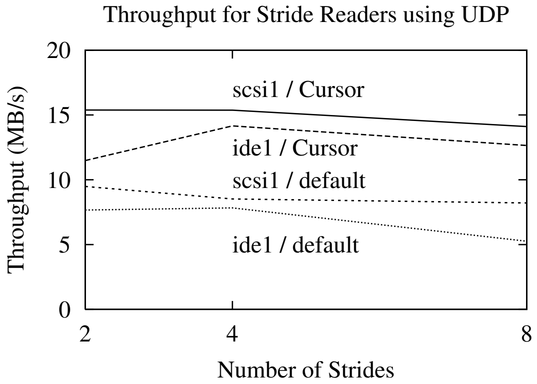

Figure 8: Throughput using the default NFS read-ahead compared to the cursor read-ahead, for reading a 256 MB file using 2, 4, and 8-stride patterns. `scsi1` runs 60-70% faster for all tests when cursors are enabled. `ide1` is only 50% faster for the 2-stride test with cursors enabled, but 140% faster for the 8-stride test.

8 Future Work

We plan to investigate the effect *SlowDown* and the cursor-based read-ahead heuristics on a more complex and realistic workload (for example, adding a large number of metadata and write requests to the workload).

In our implementation of *nfsheur* cursors, no file handle may have more than a small and constant number of cursors open at any given moment. Access patterns such as those generated by Grid or MPI-like cluster workloads can benefit from an arbitrary number of cursors, and therefore would not fully benefit from our implementation.

In our simplistic architecture, it is inefficient to increase the number of cursors, because every file handle

File System		s = 2	s = 4	s = 8
`ide1`	UDP/Default	7.66 (0.02)	7.83 (0.02)	5.26 (0.02)
	UDP/Cursor	11.49 (0.29)	14.15 (0.14)	12.66 (0.43)
`scsi1`	UDP/Default	9.49 (0.03)	8.52 (0.04)	8.21 (0.03)
	UDP/Cursor	15.39 (0.20)	15.38 (0.15)	14.12 (0.46)

Table 1: Mean throughput (in MB/s) of ten reads of a single 256 MB file using a stride read, comparing the default read-ahead heuristic to the cursor-based heuristic. The cache is flushed before each run. The numbers in parenthesis give the standard deviation for each sample.

will reserve space for this number of cursors (whether they are ever used or not). It would be better to share a common pool of cursors among all file handles.

It would be interesting to see if the cursor heuristics are beneficial to file-based database systems such as MySQL [7] or Berkeley DB [25].

9 Conclusions

We have shown the effect of two new algorithms for computing the sequentiality count used by the read-ahead heuristic in the FreeBSD NFS server.

We have shown that improving the read-ahead heuristic by itself does not improve performance very much unless the *nfsheur* table is also made larger, and making *nfsheur* larger by itself is enough to achieve optimal performance for our benchmark. In addition, our changes to *nfsheur* are very minor and add no complexity to the NFS server.

We have also shown that a cursor-based algorithm for computing the read-ahead metric can dramatically improve the performance of stride-pattern readers.

Perhaps more importantly, we have discussed several important causes of variance or hidden effects in file system benchmarks, including the ZCAV effect, the interaction between tagged command queues and the disk scheduling algorithm, and the effect of using TCP vs UDP, and demonstrated the effects they may have on benchmarks.

9.1 Benchmarking Lessons

- *Do not overlook ZCAV effects.* To reduce the interference of ZCAV effects on your benchmarks, confine your benchmarks to a small section of the disk, and use the largest possible disk in order to minimize the difference in transfer speed between the innermost and outermost tracks. If the goal of your benchmark is to minimize the impact of seeks or rotational latency, use the innermost tracks. For the best possible performance, use the outermost tracks.
- *Know your hardware.* Typical desktop workstations use PCI implementations that have a peak transfer speed slower than typical disk drives, and cannot drive a gigabit Ethernet card at full speed.
- *Check for Unexpected Variation.* The disk scheduler can order I/O in a different order than you expect – and tagged command queues can reorder them yet again. This can cause unexpected effects; watch for them.
- *Know your protocols.* Are you using TCP or UDP? Does it make a difference for your test? Would it make a difference in other situations?

Acknowledgments

The paper benefited enormously from the thoughtful comments from our reviewers and Chuck Lever, our paper shepherd. This work was funded in part by IBM.

Obtaining Our Software

The source code and documentation for the changes to the NFS server and disk scheduler described in this paper, relative to FreeBSD 4.6 (or later), are available at http://www.eecs.harvard.edu/~ellard/NFS.

References

[1] Timothy Bray. The Bonnie Disk Benchmark, 1990. http://www.textuality.com/bonnie/.

[2] Angela Demke Brown and Todd C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *The Fourth Symposium on Operating Design and Implementation OSDI*, October 2000.

[3] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, Florence, Italy, June 1999.

[4] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Monterey, CA, 1994.

[5] Harvey M. Deitel. *Operating Systems, 2nd Edition*. Addison-Wesley Publishing Company, 1990.

[6] Rohit Dube, Cynthia D. Rais, and Satish K. Tripathi. Improving NFS Performance Over Wireless Links. *IEEE Transactions on Computers*, 46(3):290–298, 1997.

[7] Paul DuBois. *MySQL, 2nd Edition*. New Riders Publishing, 2003.

[8] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[9] J. Howard, M. Kazar, S. Menees, S. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[10] Rick Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 261–278, San Fransisco, CA, USA, 17–21 1994.

[11] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[12] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[13] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[14] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[15] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix Technical Conference*, January 1997.

[16] J. Mogul. Brittle Metrics in Operating Systems Research. In *The Seventh Workshop on Hot Topics in Operating Systems: [HotOS-VII]: 29–30 March 1999, Rio Rico, Arizona*, pages 90–95, 1999.

[17] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.

[18] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[19] Thomas M. Ruwart. File System Performance Benchmarks, Then, Now, and Tomorrow. In *18th IEEE Symposium on Mass Storage Systems*, San Diego, CA, April 2001.

[20] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990.

[21] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX Annual Technical Conference*, pages 18–23, June 2000.

[22] Margo I. Seltzer, David Krinsky, Keith A. Smith, and Xiaolan Zhang. The Case for Application-Specific Benchmarking. In *Workshop on Hot Topics in Operating Systems*, pages 102–107, 1999.

[23] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why Does File System Prefetching Work? In *USENIX Annual Technical Conference*, pages 71–84, June 1999.

[24] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.

[25] Sleepycat Software. *Berkeley DB*. New Riders Publishing, 2001.

[26] SPEC SFS (System File Server) Benchmark, 1997. http://www.spec.org/osg/sfs97r1/.

[27] Diane Tang. Benchmarking Filesystems. Technical Report TR-19-95, Harvard University, Cambridge, MA, USA, October 1995.

[28] Peter Triantafillou, Stavros Christodoulakis, and Costas Georgiadis. A Comprehensive Analytical Performance Model for Disk Devices Under Random Workloads. *Knowledge and Data Engineering*, 14(1):140–155, 2002.

[29] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 90–106. IEEE Computer Society Press and Wiley, 2001.