Moving Network Server Latency Off the Disk Speed Curve

Yaoping Ruan and Vivek Pai
Department of Computer Science
Princeton University
{yruan,vivek}@cs.princeton.edu

Abstract

Using three generations of hardware and two server software packages, we demonstrate that while network server capacity has been improving, the server-induced latency is not keeping pace with improvements in processor speed. We trace the roots of this problem to head-of-line blocking within filesystem-related kernel queues, which causes degradation of existing service/fairness policies. The net result of this problem is that requests that could have been served in memory, with low latency, are forced to unnecessarily wait on disk-bound requests. While this batching behavior has relatively little impact on throughput, its effects on latency are severe. This problem manifests itself in a phenomenon we call *service inversion*, which we show is responsible for most of the latency increase under load.

We modify servers to avoid these problems, and using them, we demonstrate a qualitatively different change in the latency profiles, generating more than an order of magnitude reduction in latency. In conjunction, we show that the resulting systems are able to serve most requests without being tied to disk performance. We show that the results are not dependent on server software architecture, and can be applied with benefits to the Flash Web Server, a high-performance event-driven Web server, as well as the widely-deployed multiple-process Apache Web Server.

1 Introduction

Much of the performance-related research in network servers has focused on improving throughput, with less attention paid to latency [8, 9, 16]. In an environment with large numbers of users accessing the Web over slow links, the focus on throughput was understandable since perceived latency was dominated by wide area network (WAN) delays. Additionally, early servers were often unable to handle high request rates, so throughput research had an easily measurable effect on service availability. The development of popular benchmarks in this area, such as SpecWeb [21] and WebStone [14], also focused on throughput, giving developers extra incentive to improve throughput.

Several trends are reducing the non-server latencies, thereby increasing the relative contribution of server-induced latency. Improvements in server-side network connectivity reduce server-side network delays, while growing broadband usage reduces client-side network delays. Content distribution networks, which replicate content on geographically-dispersed servers, reduce the distance between the client and the desired data, reducing round-trip latency. Some recent work has begun to address the issue of measuring end-user latency [4, 18], with optimization approaches mostly focusing on scheduling [7, 12, 22, 23].

However, little is understood about the trends in network server latencies, or even how the system components affect them. Current research generally makes the assumption that server latency is largely caused by queuing delays, that it is inherent to the system, and that scheduling techniques are the preferred method to address the problem. Unfortunately, these assumptions are not explicitly tested, complicating attempts to systematically address the origins of latency. Based on these observations, we are interested in understanding the causes of network server latency, so that the scalability of server latency can be addressed. To obtain broadly-applicable results, we compare two software packages, the event-driven Flash Web Server [16] and the multiple-process Apache Web Server [1], across three generations of processors.

We find that both servers scale well on in-memory workloads, but that their results on a workload involving disk access show less impressive behavior. Though server throughput improves with increases in processor speed, server latency drops only slightly. These observation are troublesome, since in-memory workloads will become less common as disk capacities continue to grow exponentially and greatly exceed physical memory sizes.

By instrumenting the kernel, we find that these servers waste much time blocking in filesystem-related system calls, even when the needed data is in physical memory. As a result, requests that could have been served from main memory are forced to wait unnecessarily for disk-bound requests. While this batching behavior has little impact on throughput, its effects on latency are severe. This head-of-line blocking causes other problems, such as a degradation of the kernel's service policies that are designed to ensure fairness. By examining individual request latencies, we find that this blocking gives rise to a phenomenon we call *service inversion*, where short requests are often served with much higher latencies than much larger requests. We also find that this phenomenon increases with load, and that it is responsible for most of the growth in server latency.

By addressing these issues both in the application and the kernel, we demonstrate a qualitatively different change in the latency profiles, exhibiting much lower *service inversion* and generating more than an order of magnitude reduction in server-induced latency. We also show that these latency profiles generally scale with processor speed, where cached requests are no longer bound by disk-related issues.

The rest of the paper is organized as follow: In Section 2, we present the test environment, workloads, methodology, and servers used throughout this paper. In Section 3, we characterize server latency across a range of workloads. We identify specific problems and their effects in Section 4, and introduce a new metric to quantify the effects in Section 5. We describe how we address these problems and the resulting servers in Section 6. We present the results of experiments on the new servers in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2 Background

Since we begin our analysis by experimentally measuring the observed latency characteristics of different servers, we first provide some context explaining our methodology, experimental setup, servers tested, and workloads. This experimental setup and workload are used through out this paper unless otherwise noted.

Processor	Pentium-II	Pentium-III	P4 Xeon
Speed	300 MHz	933 Mhz	3 GHz
Bcopy bandwidth	93 MB/s	265 MB/s	624 MB/s
Read bandwidth	213 MB/s	555 MB/s	1972 MB/s
Memory latency	245 ns	101 ns	116 ns

Table 1: Server hardware information

2.1 Testbed configuration

We use three hardware platforms to span three processor generations and an order of magnitude increase in raw clock speed. To equalize as many factors as possible, all machines use the same hard drive (5600 RPM Maxtor IDE 2F030L0), network adaptor (Netgear GA621 Gigabit Ethernet), and physical memory size (1 GB). The details of our server machines are shown in Table 1, with measured values provided by lmbench [13]. We use six 1.3 GHz AMD Duron machines as clients, with 256 MB of memory per machine. The network is a Netgear FS518 Gigabit Ethernet switch. All machines are configured to use the default (1500 byte) MTU. We use the FreeBSD 4.6 operating system, with all tunable parameters set for high performance – 128K max sockets, 16K file descriptors per process, 64KB socket buffers, 80K mbufs, 40K mbuf clusters, and 16K inode cache entries.

2.2 Server Software

To test the common scenario as well as a more aggressive case, we use two different servers with different software architectures and design goals. Our standard case uses the multi-process Apache server [1], version 1.3.27, since it is widely deployed and general purpose. To represent higher-performance servers, we use the event-driven Flash Web Server [16], a high-performance research system with aggressive optimizations. It uses a single main process that uses non-blocking sockets to multiplex all client connections. All disk-related operations are performed by a small set of helper processes to avoid blocking the main process. To increase performance, it aggressively caches open files, memory-mapped data, and application-level metadata. In contrast, Apache dedicates one process per connection, and performs very little caching in order to reduce resource consumption. Both servers are configured for maximum performance. In Flash, the file cache size is set to 80% of physical memory, with remaining parameters automatically adjusted. We also aggressively configure Apache – periodic process shutdown is disabled, reverse lookups are disabled, and the maximum number of processes is raised to 2048 by recompiling with an increased HARD_SERVER_LIMIT. Since Apache's logging causes a noticeable performance loss, we disable access logging in both servers.

2.3 Workloads

In order to use a widely-understood workload while still maintaining tractability in the analysis, we focus on a static content workload modeled on the SpecWeb96 and SpecWeb99 [21] benchmarks. These workloads are modeled after the access patterns of multiple Web sites, with file sizes ranging from 100 bytes to 900 KB, and are the *de facto* standards in industry, with more than 190 published results. File popularity is explicitly modeled – half of all accesses are for files in the 1KB-9KB range, with 35% in the 100-900 byte range, 14% in the 10KB-90KB range, and 1% in the 100KB-900KB range, yielding an average dynamic response size of roughly 14 KB. Each directory in the system contains 36 files (roughly 5 MB total), and the directories are chosen using a Zipf distribution with an alpha value of 1.

SpecWeb99 normally self-scales, increasing both data set size and number of simultaneous connections with the target throughput. However, this approach complicates comparisons between different servers, so we use fixed values for both parameters. To facilitate comparisons with previous work such as Haboob [23] and Knot [22], we use their parameters of a 3 GB data set and 1024 simultaneous connections. We also adopt the persistent connection model from these tests, with clients issuing 5 requests per connection before closing it. With these parameters, we maintain per-client throughput levels comparable SpebWeb99's quality-of-service requirements.

2.4 Latency Measurement Methodology

To understand how load affects response latency, we measure latencies at various requests rates. Each server's maximum capacity is determined by having all clients issue requests in an infinite-demand model, and then relative rates are reported as load fractions *relative to the infinite demand capacity of each server*. This process simplifies comparisons across servers, though it may bias toward servers with low capacity. Latency is measured by recording the wall-clock time between the client starting the HTTP request and receiving the last byte of the response. We normally report mean response time, but we note that it can hide the details of the latency profiles, especially under workloads with widely-varying request sizes. So, in addition to mean response time, we also present the 5^{th}, 50^{th} (median) and 95^{th} percentiles of the latency. Where appropriate, we also provide the cumulative distribution function (CDF) of the client-perceived latencies.

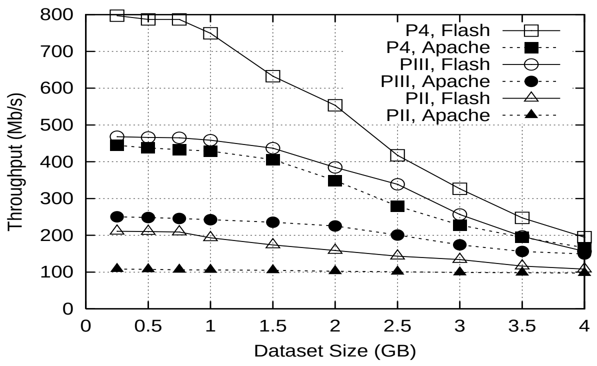

Figure 1: Flash and Apache server capacities with various data set sizes on the three processor generations

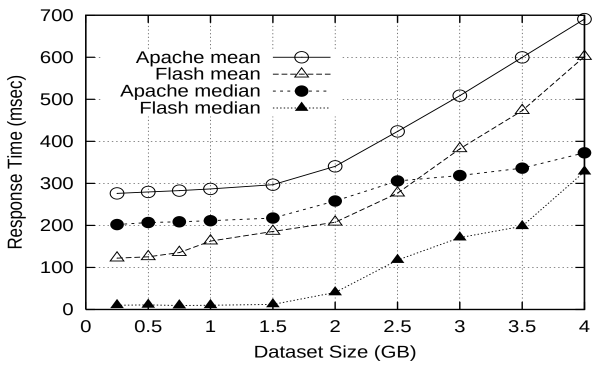

Figure 4: Median and mean latencies of Apache and Flash with various data set sizes

3 Latency Characteristics

We begin our analysis by measuring the infinite-demand performance of the various configurations while adjusting the data set size. These results, shown in Figure 1, exhibit several interesting properties. The in-memory performance of both Apache and Flash scale well with processor speed – the Pentium-III performance is roughly twice that of the Pentium-II, and the Pentium 4 is twice as fast as the Pentium-III. Flash shows noticeably higher performance on this portion of the workload, largely due to its aggressive optimizations. Once the data set size exceeds physical memory, performance degrades, especially for the two faster processors. In these cases, the performance is disk-limited, and both machines have idle CPU, especially when used with the Flash server. The Pentium-II Apache result shows less relative degradation, largely because its in-memory performance worse than the disk-bound performance of most other configurations. The capacity results are not qualitatively surprising since one would expect performance degradation as the data set size exceeds physical memory.

To understand how latencies are affected by processor speed, we conduct a more detailed look at server latency, shown in Figures 2 and 3. These two graphs represent an in-memory workload and a disk-bound workload, respectively, and show the mean latencies for both server packages across all three processors. Measurements are taken at various load levels, and show a remarkable consistency – at the same *relative* load levels, both Apache and Flash exhibit similar latencies, the in-memory latencies are much lower than the disk-bound latencies, and the latencies show only minor improvement with processor speed. These measurement shows that processor improvements have resulted in improved server capacity, but have not significantly affected latency.

A deeper investigation of the effect of data set size on server latency provides more insight into the underlying problems as well as a surprising result. Figures 4 shows mean and median latencies as a function of data set size, where all tests are run at a relative load level of 0.95 on the Pentium-4. The mean latency remains relatively flat for the in-memory workload, but begins to grow when the data set size exceeds the physical memory of the machine, 1 GB. This increase in mean latency is expected, since these filesystem cache misses require disk access, and the disk latency will raise the mean.

The *increase in median latency* is quite surprising for this workload – the request popularity is such that most requests should be comfortably served out of the filesystem cache. SpecWeb strongly biases toward small files within each directory, and uses a Zipf distribution in selecting directories. The net result of these factors is that the most popular files consume very little aggregate space. Table 2 illustrates this point well – even the top 95% of all requests would consume less memory than the physical memories of our machines. If we make the reasonable assumption that the OS is capable of

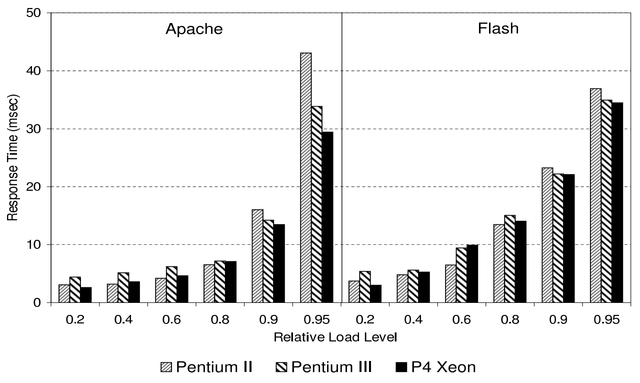

Figure 2: In-memory workload (0.5 GB) latency profiles of Apache and Flash across three processor generations

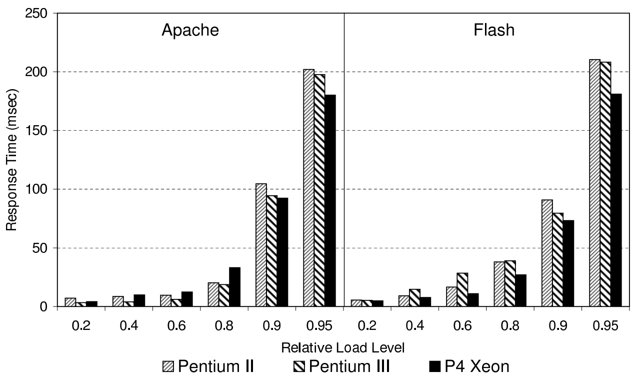

Figure 3: Disk-bound workload (3.0 GB) latency profiles of Apache and Flash across three processor generations

caching the most active 4.4 MB (the 50% working set size at 3 GB) of data on a system with 1 GB of memory, then we are left to conclude that the small amount of cache miss activity is interfering with the accesses for documents that should be cache hits.

This observation is problematic, because it implies that, for non-trivial workloads, server latency is tied to disk performance, even for cached requests. Without server or operating system modification, latency scalability is therefore tied to mechanical improvements, rather than faster improvements in electronic components. The *expected* latency behavior would have been precisely the opposite – that as the number of disk accesses increased, and the overall throughput decreased, the median latency would actually *decrease* since fewer requests would be contending for the CPU at any time. Queuing delays related to CPU scheduling would be mitigated, as would any network contention effects.

Based on the above observation, we focus on server latency characteristics on disk-bound workloads and the fastest processor. We use a 3 GB data set for measurements in the coming sections. Our initial latency measurements show the two servers have seemingly similar mean response time profiles, despite their different software architectures. Using the infinite-demand throughputs, we run these servers with request rates of 20%, 40%, 60%, 80%, 90%, and 95% of the infinite-demand rate, with the results shown in Figures 5 and 6. While the general shape of the mean response curves is not surprising, some important differences emerge when examining the others. Apache's median latency curve is much flatter, but rises slightly at the 0.95 load level. The mean latency for Apache becomes noticeably worse at that level, with a value comparable to that of Flash, while Apache's latency for the 95^{th} percentile grows sharply.

Data Set Size (GB)	Top 50% (MB)	Top 90% (MB)	Top 95% (MB)	Top 99% (MB)
1	2.1	39.5	64.6	138.3
2	3.0	72.9	123.6	262.8
3	4.4	101.8	181.2	385.7
4	4.9	131.8	235.0	505.0

Table 2: SpecWeb's popularity distributions yield relatively small working set for the most popular files. Sizes do not scale linearly with data set size due to the Zipf-based popularity distribution of directories

Given the different growth patterns for the different latency percentiles, we would expect the complete latency CDF plots to show different curves for the two servers, and this belief is confirmed in Figures 7 and 8, where latency CDFs are shown for three load levels in addition to infinite demand. Both servers exhibit latency degradation as the server load approaches infinite demand, with the median value rising over one hundred times.

Two features which appear to be related to the server architecture are immediately apparent – the relative smoothness of the Flash curves, and the seemingly lower degradation for Apache at or below load levels of 0.90. By multiplexing all client connections through a single process, the Flash server introduces some batching effects, particularly through the use of the `select()` system call. This batching causes even the fastest responses to be affected under load. As a result, Flash returns very few responses in less than 10ms when the load exceeds 90%, whereas Apache still delivers over 60% of its responses within that time. We believe that this is because Apache's multiple processes operate independently, and in-memory requests are often being serviced very quickly without interference from other requests.

However, this portion of the CDF does not explain Apache's worse mean response times, for which the explanation can be seen in the tail of the CDFs. Though Apache is generally better in producing quick responses under load, latencies beyond the 95^{th} percentile grow sharply, and these values are responsible for Apache's worse mean response times. Given the slow speed of disk access, these tails seem to be disk-related rather than purely queuing effects. Given the high cost of disk access versus memory speeds, these tails dominate the mean response time calculations.

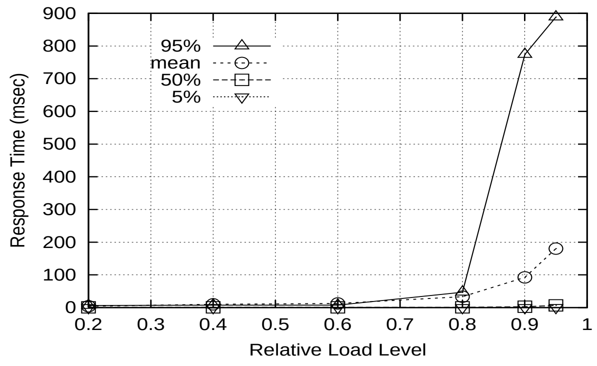

Figure 5: Apache Latency Profile on the Pentium-4 and a 3.0 GB data set. The relative load of 1.0 equals 241 Mb/s

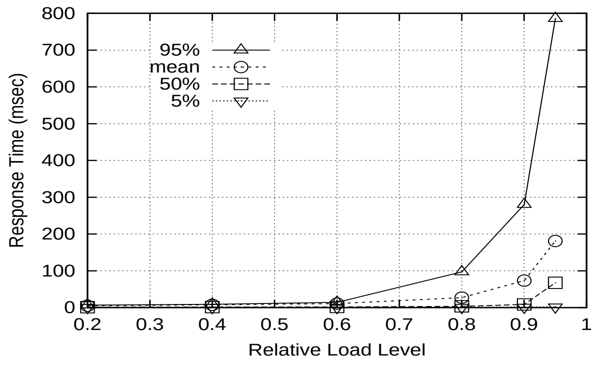

Figure 6: Flash Latency Profile on the Pentium-4 and a 3.0 GB data set. The relative load of 1.0 equals 336 Mb/s

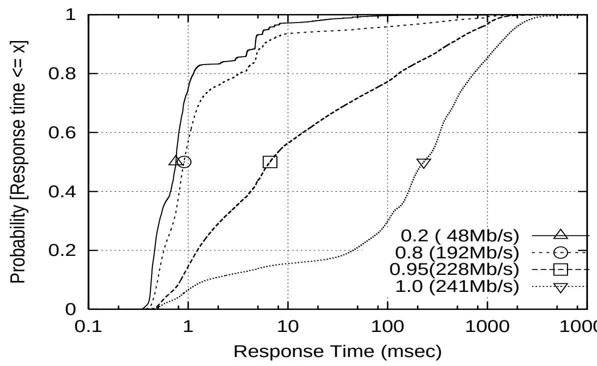

Figure 7: Apache latency CDFs for various load levels on the Pentium-4 and a 3.0 GB data set

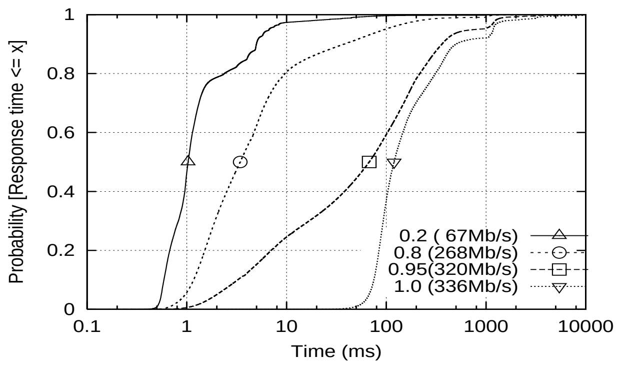

Figure 8: Flash latency CDF for various load levels on the Pentium-4 and a 3.0 GB data set

4 The Origins of High Latency

In this section we investigate the origins of the high latency we saw on the earlier tests. By instrumenting the kernel, we trace much of the root cause to blocking in filesystem-related system calls. This blocking affects the queuing model for the services, causing a policy degradation when head-of-line blocking occurs. We present evidence that this behavior is occurring in both Flash and Apache, although via different mechanisms.

Using an infinite-demand workload on a 3 GB data set, we find that even when we increase the number of clients, only the Pentium-II processor becomes saturated, using either Apache or Flash. Using the "top" tool, which shows process activity, we find that the main Flash process is blocking inside the kernel on operations other than the `select()` system call. We are not surprised that the faster processors are capable of saturating the disk, but we are surprised to see Flash's main process blocking, since it should be avoiding all blocking outside of `select()` via non-blocking network sockets and disk-related helper processes. Directly observing similar slowdowns in Apache is more difficult, since its multiple-process design exploits the fact that the operating system will schedule another process when the current process blocks.

4.1 Observing Blocking in Flash

Further evidence that Flash is unexpectedly blocking can be obtained by observing the return values of the `select()` system call. This call takes a list of file descriptors as input, and returns a count of how many of them are ready for activity. This call is known to not be work-conserving since all descriptors are polled at each call invocation – when free CPU cycles are scarce, each call will return with many ready sockets, but at low load, the call will return as soon as a single descriptor becomes ready [2]. For the disk-bound workload on the Pentium 4, we show a CDF of the return values from `select()` in Figure 9.

Clearly, most activity is being reported in batches – the median number of ready descriptors is **12**, the mean is **61** and the maximum length is more than **600**. More than 25% of the invocations return over 100 ready descriptors. This kind of behavior would be understandable if the CPU were taxed, but this workload shows the CPU still has idle time. Given the observation from "top" that the main process is blocking, we can see that the blocking is causing both the CPU idle time and the batching. Even though descriptors are ready for servicing and idle CPU exists, the blocking system calls are artificially limiting overall performance.

This measurement also shows why *median* latency is be-

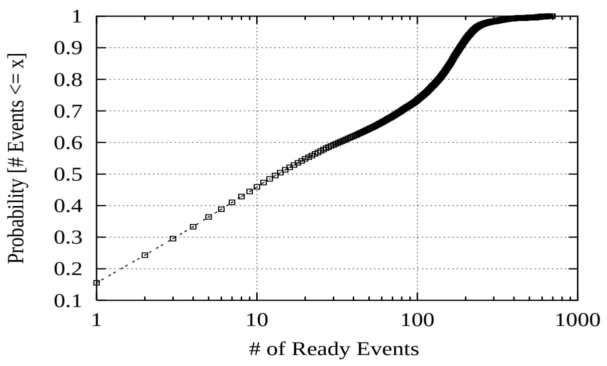

Figure 9: CDF of number of ready events (the return values from `select()`) in Flash

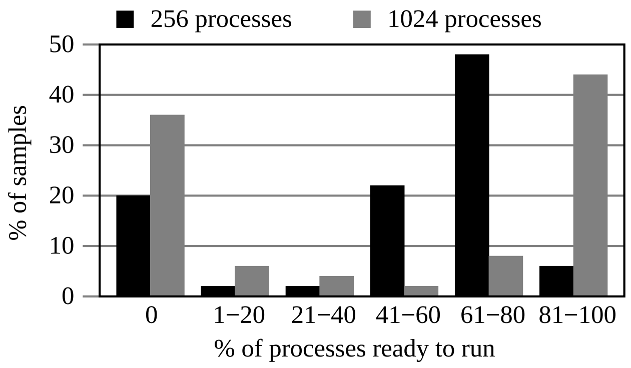

Figure 10: Scheduler burstiness (via the instantaneous run queue lengths) in Apache for 256 and 1024 processes

ing affected in Flash and why this trend hinders latency scalability – since all connections are multiplexed and handled within a single process, any disk blocking caused by a relatively unpopular file can prevent the servicing of cache hits during that time. With faster processors, this problem is likely to get *worse*, since the extra capacity means that more simultaneous connections can be supported. When any of these connections causes blocking, more connections are affected.

4.2 Inferring Blocking in Apache

While blocking in the Flash server is unexpected because it utilizes non-blocking system calls, we expect blocking in the multiple-process Apache server. Each process handles only one connection, so if it blocks in the processing of a request, conventional wisdom holds that such blocking is necessary and affects only the request being handled. However, since the Apache server also shows high latency for memory cache hits, we suspect that it is similarly affected by some form of head-of-line blocking.

Since Apache does not have easily-testable invariant regarding blocking such as Flash does, we must use another mechanism to infer it. We can use the observation that blocking in Flash increases the burstiness of system activity to find a similar behavior in Apache. In particular, we note that if resource contention occurs in Apache, it would involve multiple processes, and the release of a resource would involve several processes becoming runnable at the same time. We would expect that the more processes involved, the higher the burstiness, and the more variability in the behavior of the run queue.

We instrument the OS scheduler to report the number of runnable Apache processes each second, and we test Apache in two configurations with a different number of simultaneous processes. In particular, we use 256 and 1024 server processes, and we use an infinite-demand workload with 1024 clients. Both configurations show roughly the same capacity, due to the infinite-demand model and local-area clients. We note how many processes are runnable at any given time, and report this as a percentage of the total number of processes in Figure 10.

From this data, we can see evidence that the behavior of the scheduler is very different in the 256 and 1024 process cases, suggesting that unexpected blocking is affecting Apache in a similar way. The 1024 process case exhibiting much more bursty behavior than the 256 process case, in line with our expectation. At this level, the scheduler is very bimodal – all processes are blocked roughly one-third of the time, and over 80% of the processes are in the runnable queue over 40% of the time. The 256 process case is more evenly distributed, with the run queue generally containing 60-80% of the total processes. Note that all processes being blocked does not imply the entire system is idle – disk and interrupt-driven network activity is still being performed.

5 Service Inversion

The most significant effect of this blocking behavior is unnecessary delays in serving queued requests. In particular, cached requests that could have been served in memory and with low latency are forced to wait on disk-bound requests. We term this phenomenon "service inversion" since the resulting latencies would be inverted compared to the ideal latencies. In this section, we study this phenomenon using a disk-bound workload and propose an approach to quantify the service inversion value.

Due to the fact that certain request processing steps operate independently of the main Flash process, any blocking that occurs early in the processing of a request can affect the fairness policies of the system. Specifically, the networking code is split in the kernel, with the sockets-related operations occurring in the "top half", which is invoked by the application. The "bottom half" code is driven by interrupts, and performs the actual sending of data. So, when an application is blocked, any data that has already been sent to the networking code can still operate in the "bottom half" of the kernel. Likewise, since the disk helpers in Flash operate as separate processes, they can continue to operate on their current request even when the main process is blocked.

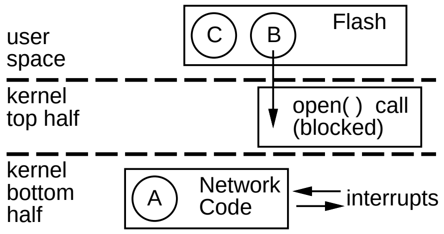

Figure 11: Service inversion example – Assume three requests (A, B, and C) arrive at the same time, and A is processed first. If it is cached and is sent to the networking code in the kernel bottom half, interrupt-based processing for it can continue even if the the process gets blocked. In this case, even if A is large, it may get finished before processing on C even starts.

To understand how this unfairness can lead to head-of-line blocking, consider the scenario in Figure 11, where three requests arrive at virtually the same time, with the middle request causing the process to block. For example, assume it is blocked by an `open()` call, which takes place relatively early in the processing of an HTTP request, shown in Figure 12. Specifically, it occurs in the "find file" step, before the data reads occurs (if needed) and before any data is sent to the networking code. If the first and third requests are cached, they would normally be served at nearly the same time. However, the first request may get sent to the networking code, and the third request would then have to wait until the process is unblocked. The net effect is that the third request suffers from head-of-line blocking. The fairness policies of the system, particularly the scheduling of network packets, is not given a chance to operate since the three requests do not reach the networking code at the same time.

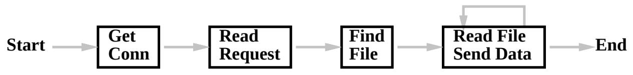

Figure 12: HTTP request processing steps

If the requests before the blocked requests are large, and the requests after the blocked request are small, we label the resulting phenomenon *service inversion*. The occurrence of this behavior is relatively simple to detect at the client – the latencies for small requests would be higher than the latencies for larger requests.

5.1 Identifying Service Inversion

As a qualitative approach to understanding the prevalence of service inversion, we take the latency CDFs discussed in section 3 and split it by decile. Given the fact that more than 95% of the requests could fit into physical memory, as shown in Table 2, then the ideal response time would be roughly proportional to its transfer size. By examining the different response sizes within each decile, we can estimate the extent of reordering. Using all 36 file sizes present in the workload would cause clutter and complicate interpretation, so we instead group the responses into four series by size such that their dynamic frequencies are roughly equal. These details of this categorization are shown in Table 3.

series	size range	percentage
1	0.1 - 0.5 KB	25.06%
2	0.6 - 4 KB	28.05%
3	5 - 6 KB	23.55%
4	7 - 900KB	23.34%

Table 3: Workload categories for latency breakdowns

The graphs in Figures 13 and 14 show the composition of responses by decile for the two servers, with the leftmost bar corresponding to the fastest 10% of the responses and the rightmost representing the slowest 10%. These graphs are taken from the latency CDFs at a load level of 0.95 on the Pentium-4.

In a perfect scenario with no service inversion, the first 2.5 bars would consist solely of responses in Series 1, followed by 2.5 bars from Series 2, etc. However, as we can see, both graphs show responses from the different series spread across all deciles, suggesting that the service inversion problem is common to both servers. One surprising aspect of these plots is that the Series 1 values are spread fairly evenly across all deciles, indicating that even the smallest files are often taking as long as some of the largest files.

Some amount of inversion is to be expected from the characteristics of the workload itself, since directories are weighted according to a Zipf-1 distribution. With roughly 600 directories in our data set, the last directory receives 600 times fewer requests than the first. So, even though files 100KB or greater account for only 1% of the requests (35 times fewer than the smallest files), the effect of the directory preference causes the largest files in the first directory to be requested about 17 times as frequently as the smallest files in the final directory. While the large files still require much more space, an LRU-style replacement in the filesystem cache could cause these large files to be in memory more often. In practice, this effect seems to be relatively minor, as we will show later in the paper.

5.2 Quantifying Service Inversion

While the latency breakdowns by decile provide a qualitative feel of the unfairness of the system, a more quantitative evaluation of service inversion can be derived from the CDF. We construct the formula based on the following observation: Given responses A, B, C, D, E with sizes $A < B < C < D < E$. If the observed response times have the same order as the response sizes, we say that no service inversion has occurred, and the corresponding value should be zero. On the contrary, if the response times are in the reverse order of their sizes, then we say that the server is completely inverted, and give it a value of 1.

The insight into calculating the inversion is the following: we want to determine how perturbed a measured order is,

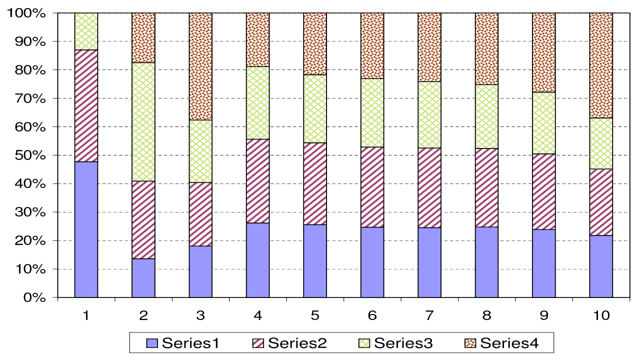

Figure 13: Apache CDF breakdown by decile at load 0.95 on 3.0 GB data set

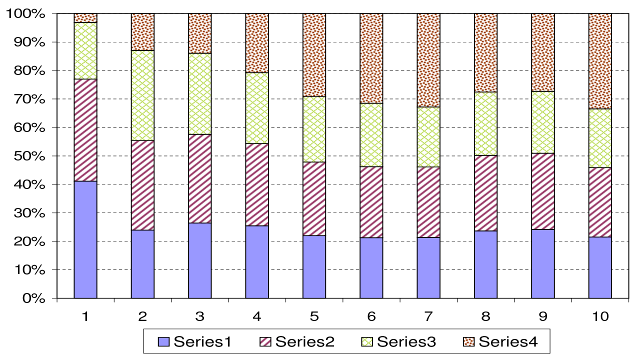

Figure 14: Flash CDF breakdown by decile at load 0.95 on 3.0 GB data set

compared with the order of the response sizes. The perturbation is merely the difference in position of a response in the ordered list of response times versus its position in a list ordered by size, where this distance is calculated for each response and summed for the entire list. We then normalize this versus the maximum perturbation possible. A particular service inversion value is given by:

$$\sum_{i=1}^{n} Distance(i)/\lfloor n^2/2 \rfloor \qquad (1)$$

where distance is absolute value of how far the request is from the ideal scenario, and $\lfloor n^2/2 \rfloor$ is the total distance of requests in the reverse order of their sizes, which is the maximum perturbation possible. In the above example, assume the observed latency order is B, C, A, D, E. By comparing with the ideal order, A, B, C, D, E, we see the distance of file B is 1, C is 1, A is 2, and D, E are 0. The inversion value is $4/12 = 0.33$. Since this measurement requires only the response sizes and latencies, as long as the distribution of sizes is the same, it can be used to compare two different servers or the same server at multiple load levels. To handle the case of multiple requests with the same response size, we calculate distance by comparing the N^{th} observed position with the N^{th} ideal position for each response of the same size.

By measuring service inversion as a function of load level, we discover that this effect is a major contributor to the latency increase under load. Table 4 shows the quantified inversion values for both servers, and demonstrates that while inversion is relatively small at low loads, it exceeds half of the worst-case value as the load level increases. The latencies at the higher load levels therefore not only suffer from queuing delays, but also service inversion delays from blocking. We will show in the next section that the delays stemming from blocking and service inversion are in fact the dominant source of delay.

6 The New Servers

In this section we describe how we modify both servers to reduce blocking, and we analyze the effects on capacity, latency, and service inversion. To the extent that space permits, we repeat all previous experiments to demonstrate that the modified servers overcome the latency and blocking problems previously observed.

	Relative load level					
	0.20	0.40	0.60	0.80	0.90	0.95
Apache	0.14	0.23	0.28	0.51	0.54	0.58
Flash	0.25	0.35	0.45	0.52	0.56	0.58

Table 4: Service inversion versus load level for Apache and Flash on a Pentium-III and 3.0 GB data set

6.1 New-Flash

The previous experiments suggest that the source of the problem lies with the operating system, and is not tied to the server's software architecture. Diagnosing the source of the problem is relatively easy using the Flash server, since the main process should maintain the invariant that it never blocks. By instrumenting the operating system to note where the main process of Flash blocks, we can identify the associated system calls and call sites.

The primary source of blocking in Flash is the `open()` system call, which is used to get access to files on cache misses. When the main process needs to open a new file, it first invokes a helper process to open the file, in order to bring the associated metadata into the filesystem cache. Then, when the helper process is done, it notifies the main process of its completion, and the main process opens the file, assuming that no blocking will occur because the metadata is already cached. We find that the problem with this assumption is directory-level inode locking, in which case lock contention can occur between the main process and the helper. One seemingly plausible candidate, filesystem access time modification, does not seem to be responsible, because disabling it has no effect on the blocking.

We address this issue by moving the `open()` call entirely out of the main process, and having the helper processes return file descriptors to the main process using the `sendmsg()` system call. We also take the opportunity to address some of the other more CPU-intensive system calls.

In recent years, FreeBSD has added support for a zero-copy data sending system call, `sendfile()`. We replace the use of memory-mapped files with this system call, which not only reduces memory bandwidth consumption, but also the number of mapped regions that the virtual memory manager handles. Likewise, FreeBSD added a scalable, work-conserving event notification facility, `kevent()`, and we use this in place of `select()`. We refer to this server as New-Flash.

6.2 Flashpache

Due to the differences in software architecture, we cannot directly employ the same technique that we used in New-Flash to improve Apache. However, if we can assume that the filesystem-related calls are the likely culprits, we can leverage the lessons from Flash. Since Apache does not perform file descriptor caching, each process calls `open()` on every request, and this behavior results in a much higher rate of these calls.

We modify Apache to offload the URL-to-file translation process, in which the `open()` system call occurs. This step is handled by a new "backend" process, to which all of the Apache processes connect via persistent Unix-domain sockets. The backend employs a Flash-like architecture, with a main process and a small number of helpers. The main process keeps a filename cache like the one in the Flash server, and schedules helpers to perform cache miss operations. The backend takes the responsibility of finding the requested file, opening the file, and sending the file descriptor and metadata information back to the Apache processes. Upon receiving a valid open file descriptor from the backend, the Apache process can return the associated data to the client. Since the backend handles URL lookup for all Apache processes, it is possible to combine duplicated requests and even preload data blocks into the filesystem cache before passing the control back to Apache processes, thus reducing the chances of more blocking. We call this new server Flashpache, to reflect its hybrid architecture.

7 Results

We begin our analysis by repeating the infinite-demand measurements for the 3 GB data set, with the results shown in Table 5. Included are the figures for the original Flash, as well as the intermediate steps of file descriptor passing (fd pass) and removing memory-mapped files (no mmap). We can see that the overall capacity of Flash has increased by 34% for this workload, while Apache's capacity increases by 13%.

	Latency (ms)			Capacities
	median	mean	90%	(Mb/s)
Flash	67.4	181.0	362.0	336.0
fd pass	11.5	50.0	71.2	395.0
no mmap	1.8	93.5	92.9	437.5
New-Flash	1.6	29.3	6.6	450.0
Apache	6.6	180.2	414.7	241.1
Flashpache	1.1	12.0	5.7	272.9

Table 5: Latencies & capacities for original and modified servers

The more impressive result, however, is the drastic reduction in latency, even when run at these higher throughputs. Flash sees improvements of **40x median, 6x mean, and 54x in 90th percentile latency**. Apache sees improvements of **6x median, 15x mean, and 72x in 90th percentile latency**. The one seemingly odd result, an increase in mean latency from fd-pass to no-mmap, is due to an increase in blocking, since the removal of `mmap()` also results in losing the `mincore()` function, which could precisely determine memory residency of pages. The New-Flash server obtains this residency information via a flag in `sendfile()`, which again eliminates blocking.

The burstiness induced by blocking has also been reduced or eliminated in both servers, as seen in Figures 15 and 16. In New-Flash, the mean number of **events per call has dropped from 61 to 1.6**, and the median has dropped from 12 to 2. Likewise, Flashpache no longer exhibits bimodal behavior at the scheduler level, instead showing roughly 20% of all processes ready at any given time. Figure 17 confirms our scalability across processors – even with much lower Pentium-II latencies, **improvements in processor speed now reduce latency** on both servers.

Not only do the new servers have lower latencies, but they also show *qualitatively* different latency characteristics. Figure 18 shows that **median latency no longer grows with data set size**, despite the increase in mean latencies. Mean latency still increases due to cache misses, but the median request is a cache hit in all cases. Figures 19 and 20 show the latency CDFs for 5^{th} percentile, mean, median, and 95^{th} percentile with varying load. Though the mean latency and 95^{th} percentile increase, the 95^{th} percentile shows less than a tripling versus its minimum values, which is much less growth than the two orders of magnitude observed originally. The other values are very flat, indicating that most of the requests are served with the same quality at different load levels. More importantly, the 95^{th} percentile CDF values are lower than the mean latency. The reason for this is that the time spent on the largest requests is much higher compared to time spent on other requests. This result conforms to the workload expectations stated in Section 3.

In summary, both new servers demonstrate lower initial latencies, slower growth in latency, and better decrease of latency with processor speed. These servers are no longer dominated by disk access times, and should scale with improvements in processors. That these changes eliminate over 80% of the latency answers the question about latency origins – **these latencies were dominated by blocking**, rather than request ordering.

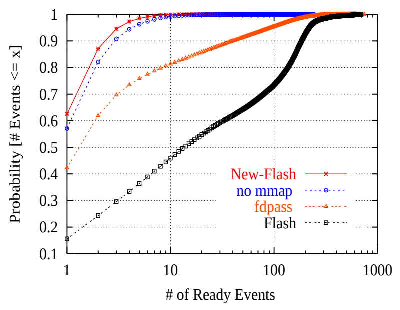

Figure 15: CDFs of # of ready events for Flash variants

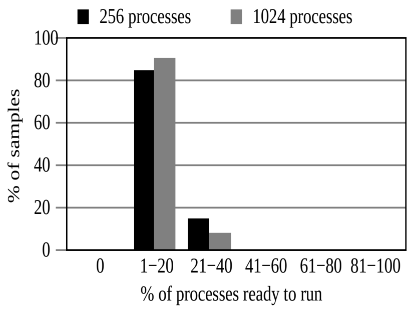

Figure 16: Scheduler burstiness in Flashpache for 256 and 1024 processes

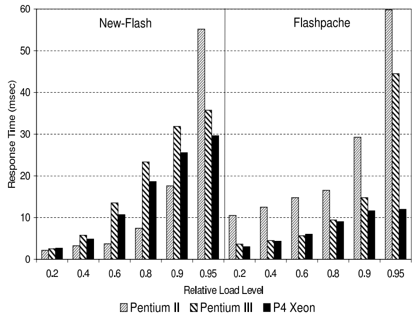

Figure 17: Latency profile of New-Flash and Flashpache on three processor generations

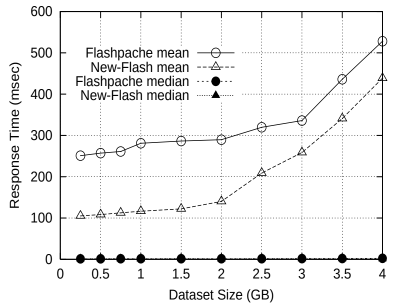

Figure 18: Median and mean latency of New-Flash and Flashpache with different data set sizes

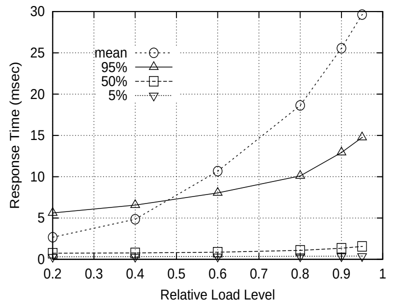

Figure 19: Latency profile of New-Flash

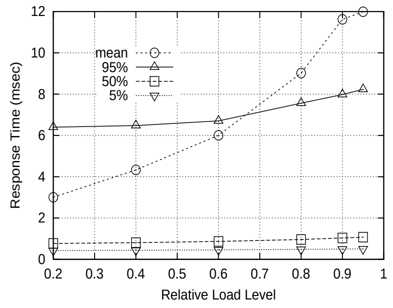

Figure 20: Latency profile of Flashpache

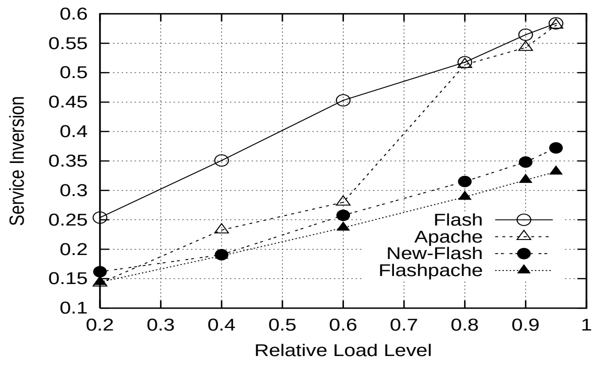

Figure 24: Service inversion of original and modified servers

7.1 Service Inversion Improvements

In order to verify the unfairness of the new servers, we further examine the latency breakdown by decile for the 0.95 relative load level and the service inversion at different load levels. Figure 21 shows the percentage of each file series in each decile for New-Flash, and we observe some interesting changes compared to the original server. The smallest files (series 1) dominates the first two deciles, the largest files (series 4) dominate the last two deciles, and the series 3 responses are clustered around the fifth decile. This behavior is much closer to the ideal than what had seen earlier. Some small responses still appear in the last column, but these may because these files have low popularity and incur cache misses. Also complicating matters is that the absolute latency value is still below 10ms for 98% of the requests, so the first nine deciles only differ by a very small amount. This observation is verified by calculating the service inversion value.

Figure 24 shows the change of the inversion value with the load level. Compared to the old system, we reduce the inversion by over 40%, suggesting requests are treated more fairly in the new system. The fact that the inversion value still increases with the load is a matter for further investigation. However, this may be a limitation of our service inversion calculation itself.

By comparing service inversion for this workload with that of a completely in-memory workload, we can see how far we are from a nearly “ideal” scenario. In particular, we are still concerned whether filesystem cache misses are responsible for the service inversion. In Figure 22, we see the latency breakdown for a workload with a 500MB data set. The difference between it and the New-Flash breakdown are visible only after careful examination. The numerical value for the in-memory case is 0.33, while the New-Flash result is 0.35, suggesting that if any inversion is due to cache misses, its measured effects are minimal. The Flashpache breakdown, shown in Figure 23, is similar. The values for the Flashpache server and its original counterpart are also shown in Figure 24, and we can see that our modifications have almost halved the inversion under high load.

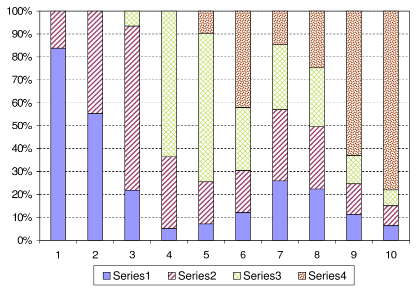

Figure 21: CDF breakdown for New-Flash on 3.0 GB data set, load 0.95

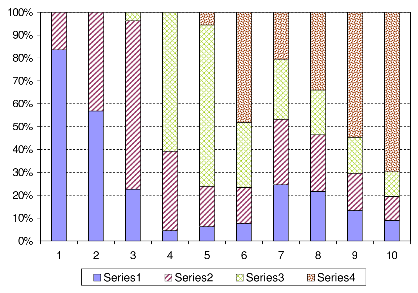

Figure 22: In-memory workload CDF breakdown, New-Flash, load 0.95

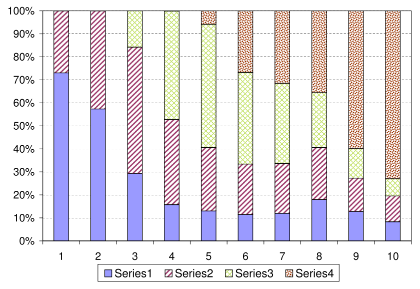

Figure 23: CDF breakdown for Flash-pache on 3.0 GB data set, load 0.95

8 Related Work

Performance optimization of network servers has been an important research area, with much work focused on improving throughput. Performance studies of the Harvest Cache [5] established the suitability of event-driven designs for high-performance network servers, and the Flash server demonstrated how to avoid some disk-related blocking [16]. Schmidt and Hu [20] performed much of the early work in studying various server architectures for improving server performance. We have demonstrated that these servers can benefit from latency-improving techniques designed to eliminate blocking within the operating system.

More recently, researchers have focused more attention to latency measurement and improvement. Rajamony & Elnozahy [18] measure the client-perceived response time by instrumenting the documents being measured. Bent and Voelker explore similar measurements, but focusing on how optimization techniques affect download times [4]. Olshefski et al. [15] propose a way of inferring client response time by measuring server-side TCP behaviors. Improvement techniques have been largely limited to connection scheduling, with most of the attention focused on SRPT policy [6], including server modification and kernel instrumentation for network stack scheduling [7]. Cohort scheduling [12] focuses on gaining performance by batching similar requests but does not examine why queuing occurs. Our work examines the root cause of the blocking, which yields the scheduling opportunities in these other studies. Our new servers rely on the existing scheduling within the operating system, and the results suggest that eliminating the existing obstacles yields automatic improvement on current service/fairness policies.

The negative impact of locking and blocking has been a major concern in parallel programming research. Rajwar et al. [19] proposed a transactional lock-free support for multi-threaded systems. While head-of-line blocking is a well-known phenomenon in the network scheduling context, we demonstrate that this phenomenon also exists in network server applications and has severe effects on user-perceived latency. Puente et al. [17] and Jurczyk et al. [11] have studied various blocking issues in the networks.

Our approach of fairness evaluation may be more suitable for network servers than the Jain fairness index [10] used in other work [23], since we focus more on the latencies of individual requests rather than coarse-grained characteristics of clients. Bansal & Harchol-Balter [3] investigate the unfairness of SRPT scheduling policy under heavy tailed workloads and draw the conclusion that the unfairness of their approach barely noticeable. Our approach does not have this concern, since we address the latency issues directly rather than try to schedule around them.

9 Conclusion

In this paper, we have examined server scalability of two server software packages on three generations of hardware. We find that faster processors improve server capacity, but have little effect on latency. By experimenting with workloads of various sizes, we determine that when disk accesses occur, both mean and median latencies increase, though the median should be unaffected. We trace the roots of this problem to head-of-line blocking within filesystem-related kernel queues. This behavior, in turn, causes batching and burstiness, which has little impact on throughput, but severely degrades latency. By examining individual request latencies, we find that this blocking gives rise to a phenomenon we call *service inversion*, where requests are served unfairly.

By addressing the blocking issues both in the Apache and the Flash server, we improve latency by more than an order of magnitude, and demonstrate a qualitatively different change in the latency profiles. The resulting servers also exhibit higher capacity, lower burstiness, and more fair request handling across a wide range of workloads. Their latency values also scale well with improvements in processor speed, making them better candidates for future improvements. Finally, our results show that most server-induced latency is tied to blocking effects, rather than queuing.

References

[1] Apache Software Foundation. The Apache Web server. http://www. apache.org/.

[2] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX 1999*

Annual Technical Conference, pages 253–265, Monterey, CA, June 1999.

[3] N. Bansal and M. Harchol-Balter. Analysis of srpt scheduling: Investigating unfairness. In *Proc. of the SIGMETRICS '01 Conference*, Cambridge, MA, June 2001.

[4] L. Bent, Geoffrey, and M. Voelker. Whole page performance. In *In Proceedings of the Seventh International Workshop on Web Content Caching and Distribution (WCW'02), Boulder, CO, August 2002.*, 2002.

[5] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125, 1995.

[6] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems (USITS'97)*, Boulder, CO, Oct. 1999.

[7] M. Harchol-Balter, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2), May 2003.

[8] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*, Phoenix, AZ, Nov. 1997.

[9] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the apache web server. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99)*, February 1999.

[10] R. Jain. Congestion control and traffic management in ATM networks: Recent advances and A survey. *Computer Networks and ISDN Systems*, 28(13):1723–1738, 1996.

[11] M. Jurczyk and T. Schwederski. Phenomenon of higher order head-of-line blocking in multistage interconnection networks under nonuniform traffic patterns. E79-D(8):1124–1129, August 1996.

[12] J. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *USENIX 2002 Annual Technical Conference*, pages 103–114, Monterey, CA, June 2002.

[13] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, pages 279–294, San Diego, CA, June 1996.

[14] Mindcraft, Inc. WebStone Benchmark. http://www.mindcraft.com/webstone.

[15] D. Olshefski, J. Nieh, and D. Agrawal. Inferring client response time at the web server. In *Proc. of the SIGMETRICS '02 Conference*, Marina Del Rey, CA, June 2002.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[17] V. Puente, J. A. Gregorio, C. Izu, and R. Beivide. Impact of the head-of-line blocking on parallel computer networks: Hardware to applications. In *European Conference on Parallel Processing*, pages 1222–1230, 1999.

[18] R. Rajamony and M. Elnozahy. Measuring client-perceived response times on the www. In *Proc. of the 3rd USENIX Symp. on Internet Technologies and Systems (USITS'97)*, San Francisco, CA, March 2001.

[19] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[20] D. C. Schmidt and J. C. Hu. Developing flexible and high-performance Web servers with frameworks and patterns. *ACM Computing Surveys*, 32(1):39, 2000.

[21] Standard Performance Evaluation Corporation. SPEC Web 96 & 99 Benchmarks. http://www.spec.org/osg/ web96/ and http://www.spec.org/osg/web99/.

[22] R. von Behren, J. Condit, F. Zhou, G. C. Necula, , and E. Brewer. Capriccio: Scalable threads for internet services. In *Proc. of the 18th ACM Symp. on Operating System Principles*, Bolton Landing, NY, Oct. 2003.

[23] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, Oct. 2001.