The Origins of Network Server Latency & the Myth of Connection Scheduling *

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

Abstract

This paper presents an investigation into the origin and breakdown of server-induced latency with the goal of understanding how latency-optimization techniques can be made more effective. Using two servers with different architectures, we analyze the latency behavior under various loads and find that a phenomenon we call *service inversion* is responsible for much of the latency increase under load. We trace the roots of this problem to negative interactions between the server application and the locking and blocking mechanisms in the kernel. Using a modified server that avoids these problems, we demonstrate a qualitatively different change in the latency profiles, generating more than an order of magnitude reduction in latency.

We also find that locking and blocking in the operating system artificially increases the burstiness of the load presented to the application. By eliminating such delays, not only do we observe a much smoother level of activity at the server, but we also eliminate most of the motivation for connection scheduling. In effect, we show that connection scheduling opportunities at the application level appear to be largely artifact of the server implementation, rather than a broadly-applicable technique. Our measurements on the modified server show much lower service inversion, suggesting that when the coarse-grained blocking is removed, the existing scheduling inside the networking stack adequately performs fine-grained connection scheduling.

1 Introduction

Much of the performance-related research in network servers has focused on improving throughput, with less attention paid to latency [13, 14, 19]. 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 and WebBench, also focused on throughput, giving developers extra incentive to improve throughput.

*This work has been partially supported by an NSF CAREER award

With the improvements in server-side connectivity, the increasing broadband penetration rate, and the popularity of Web proxy cache servers, server-induced latency can become a noticeable fraction of the end user's response latency. Some recent work has begun to address the issue of measuring end-user latency [6, 21], with optimization approaches mostly focusing on scheduling [11, 16, 24, 25]. These approaches generally make the assumption that queuing delays are inherent to the system, and that existing operating systems do not handle the request queue optimally. Unfortunately, the existing research does not address *why* these delays exist, complicating attempts to systematically address their origins. Based on these observations, we are interested in understanding the causes of network server latency and investigating how latency optimization techniques can be made more effectively.

This paper investigates the latency characteristics of two servers under various load conditions in order to understand how latency arises and what steps can be taken to reduce it. We use the event-driven Flash Web Server [19] and the multiple-process Apache Web Server [1] to represent the two dominant approaches to server software architecture. Despite their very different architectures and implementations, we find that both show similar latency increases under load, and also exhibit behavior we call *service inversion*, which is responsible for most of the latency increase under load. We trace the roots of this problem to negative interactions between the server application and the locking and blocking mechanisms in the operating system. 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 further discover that locking and blocking in the operating system artificially increase the burstiness of the load presented to the application. This burstiness in event delivery has often been the motivating condition for application-level connection scheduling. We observe that eliminating such delays in the Flash server not only increases the performance of the system, but also reduces the burstiness of event delivery. With a smaller number of ready events at any time, the opportunities for connection scheduling largely disappear, suggesting that connection scheduling is unnecessary and only a result of other

implementation artifacts. By monitoring activity at the OS scheduler, we confirm that similar problems arise in the multiple-process Apache server, confirming that these locking/blocking problems, and the associated artificial burstiness in load, are not dependent on server architecture.

Finally, we address the remaining issues of latency and fairness by examining the networking stack, and demonstrate a new approach to fairness that adaptively adjusts the TCP congestion window. We show that it achieves effects similar to scheduling policies like SRPT, but with much less effort. The results from our latency studies demonstrate that when the coarse-grained blocking is avoided, the existing scheduling inside the networking stack adequately performs fine-grained connection scheduling, that application level connection scheduling is largely an artifact of the server implementation, and can be rendered unnecessary.

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 present our measurements of server latency and introduce the server inversion metric. We describe our investigation of some of these latency origins in Section 4 and also describe how we address the problems identified. In Section 5, we show the latency measurements using our new server. We discuss how these changes affect the burstiness of event delivery in the server and its implications on connection scheduling in Section 6. Finally, we discuss some related work in Section 7 and conclude in Section 8.

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.

2.1 Latency Measurement Methodology

We measure latencies at various requests rates, both to understand how the latency profile changes with load, and also to avoid overloading the server. We control the load by adjusting the client request rate and we measure the client-perceived latency by recording the wall-clock time between starting the HTTP request and receiving the last byte of the response. In practice, we first measure the server's upper limit on capacity by having all of the clients issue requests in an infinite demand model, and then we measure the response time at various percentiles of the capacity by changing the request rate. To simplify comparisons of different servers, we generally report all rates as load fractions *relative to the infinite demand capacity of each server*.

While mean response time is a common metric in latency-related measurements, it can hide the details of the latency profiles, especially under workloads with widely-varying request sizes, such as Web workloads. So, in addition to mean response time, we also present the 5^{th} and 95^{th} percentiles of the latency, as well as the median (50^{th} percentile). For some measurements, we also provide the cumulative distribution function (CDF) of the client-perceived latencies.

2.2 Experimental Setup

All experiments are run on a uniprocessor 3.0GHz Pentium 4 Xeon server with 1GB of physical memory, one 7200 RPM IDE disk, and a single Netgear GA621 gigabit ethernet network adaptor. The clients consist of ten Pentium II machines running at 300MHz, with 128 MB of memory per machine. All clients are connected to an Netgear FS518 switch and communicate with the server through a gigabit uplink. 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. The number sockets in the box is increased to 128K, the number of file descriptors per process is increased to 16K, default socket buffer sizes are increased to 64KB, mbufs and mbuf clusters are 80K and 40K respectively, and and the filesystem inode cache is increased to 16K entries.

2.3 Servers

To obtain diversity in measurements, we focus on the event-driven Flash Web server [19] and the widely used multi-process Apache server [1], version 1.3.27, because they represent the two main designs common in server architectures. The Flash server represents the event-driven approach, using a single main process that multiplexes all client connections via the use of non-blocking sockets. The main process employs a set of helper processes to perform disk-related operations to load data into the filesystem cache. Aggressive caching of open files, memory-mapped data, and application-level metadata are designed to increase its performance. Apache, in contrast, dedicates one process per in-progress connection, and performs very little caching of results in order to reduce the resource consumption of each process. Both servers are configured for maximum performance. In Flash, the cache size is set to 800MB, and the remaining parameters are automatically adjusted based on the cache size. In Apache, we increase the HARD_SERVER_LIMIT in the source file to support 2048 maximum processes. We disable the periodic shutdown of processes, in order to reduce the performance loss associated with that cleanup. Since Apache logging causes a noticeable performance loss, we disable access logging in both servers.

2.4 Workloads

Our choice of workloads is designed to maintain as much realism as possible while still confining the number of free parameters in order to make the analysis more tractable. We focus on a static content workload modeled on the SpecWeb96 and SpecWeb99 [23] benchmarks, which are the *de facto* standards in industry, with more than 150 published results. These workloads are in turn modeled on the access patterns from multiple Web sites, with file sizes ranging from 100 bytes to 900KB. 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 response size of roughly 14 KB. Each directory in the system contains 36 files (roughly 5.5 MB total), and the directories are chosen using a Zipf distribution with an alpha value of 1.

These workloads are normally self-scaling, where both the data set size and the number of simultaneous connections increase with the throughput level of the system. However, this approach would render analytical comparisons between servers of different capacities more difficult, so we choose to fix both the data set size and the number of simultaneous connections. To facilitate comparisons with other researchers, we choose a data set size of 3.3 GB, and 1020 simultaneous connections, which is a scenario used to evaluate the Haboob [25] and Knot [24] servers. With these parameters, we maintain per-client throughput levels comparable to the ones specified in the quality-of-service requirements in the SpecWeb99 benchmark. To simulate connection behavior in the real world, persistent connections are used with clients issuing 5 requests per connection before closing it.

3 Server Latency Characteristics

In this section, we show latency characteristics of our two servers under various loads and analyze how the latency profiles differ at various levels of detail. In doing so, we also propose a new index for measuring the fairness of servers with respect to response latency.

3.1 Latency versus Load

We first use an infinite demand workload to measure the capacity of our servers, and then use the measurement methodology described previously to drive the servers at different load levels. We are interested in how these two servers with very different architectures behave at the macroscopic level, and how their latency profiles change under load. On this workload, in infinite demand mode, Apache is able to achieve 305 Mb/s, while Flash achieves 352 Mb/s. The slight advantage for Flash is not surprising due to its aggressive caching optimizations, but these benefits are tempered by the disk access in this workload.

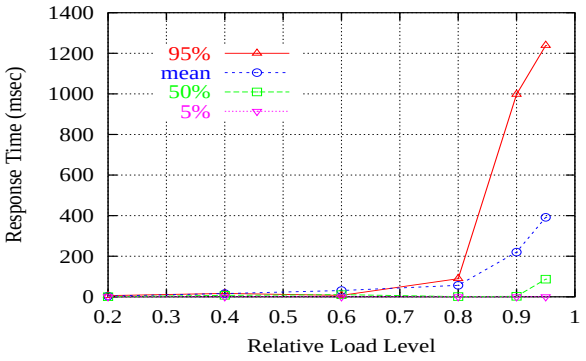

Figure 1: **Apache Latency Profile** – max load of 305 Mb/s

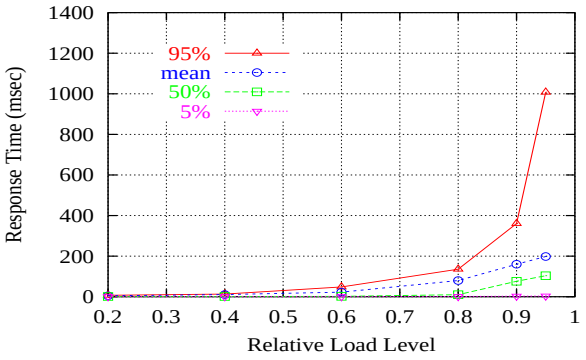

Figure 2: **Flash Latency Profile** – max load of 352 Mb/s

Our initial latency measurements show the two servers to have seemingly similar mean response time profiles, despite their different 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 1 and 2. While the general shape of the mean response curves is not surprising, some important differences emerge when examining the others. The Apache median latency curve shows much flatter behavior, and matches the Flash value at the 0.95 load level. The mean latency for Apache becomes noticeably worse at that level, with a value roughly double that of Flash, while Apache's latency for the 95^{th} percentile grows sharply.

Given the different growth patterns for the different latency percentiles, we would expect that complete latency CDF plots to show different curves for the two servers, and this belief is confirmed in Figures 3 and 4, where latency CDFs are shown for three load levels in addition to infinite demand. Both servers exhibit much latency degradation as the server load approaches infinite demand, with the median value growing by over one hundred times.

Two features of these CDFs, 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

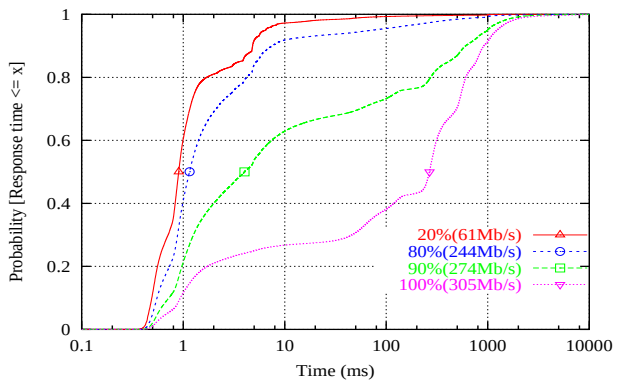

Figure 3: **Apache Latency CDF**

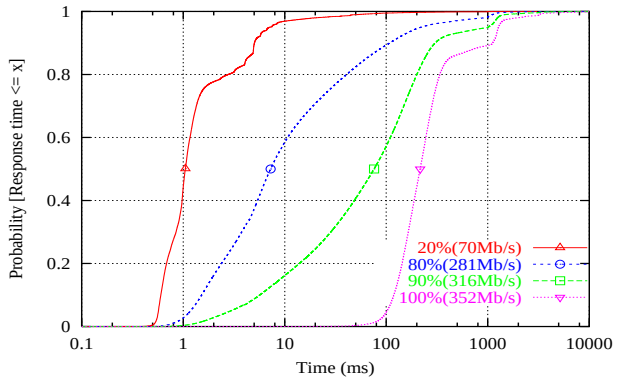

Figure 4: **Flash Latency CDF**

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 relatively 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.

3.2 Identifying Service Inversion

The seemingly contradictory results for Flash, having generally "worse" CDF curves but a better mean latency, leads us to further investigate the underlying reasons. Also puzzling is why these curves show delays in the tens or hundreds of milliseconds before the 80^{th} percentile – only 1% of the workload involves requests larger than 100KB, so a rough estimate would suggest that 99% of all requests could be serviced using only one-tenth of the data set size. For our 3.3GB data set, this figure is 330MB, which is less than the main memory of the machine.

series	size range	percentage
0	0.1 - 0.5 KB	25.06%
1	0.6 - 4 KB	28.055%
2	5 - 6 KB	23.55%
3	7 - 900KB	23.335%

Table 1: **Workload Categories** for latency breakdowns

The above analysis implies that requests are being processed unfairly, with small responses sometimes being delayed for hundreds of milliseconds. In Flash, the batching effects and previously-observed overheads [3] of the `select()` call would be the apparent culprit, but the counterpart in Apache is not obvious. Intuitively, the response time of a multi-process server is largely controlled by the OS scheduler, which applies a processor sharing policy at a fine-grained level. Runnable processes are not affected by blocked processes, and the large responses are split into multiple service instances due to the CPU quantization and the socket buffer size of the network. The second factor is likely to dominate, so no request is likely to receive more than 64KB (the socket buffer size) of service at once, regardless of the total response size.

This unfairness, where smaller requests can be queued behind (portions of) larger requests, can increase the overall latency of the system, and can contribute to some of the latency growth. We term the unfairness of the system "service inversion", representing the disordering of the requests served. To begin investigating how prevalent service inversion is in these workloads, we begin by visualizing its prevalence.

As a qualitative approach to understanding the prevalence of service inversion, we split the latency CDF by decile, and then show the occurrence of different response sizes within each decile. 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 split are shown in Table 1.

The graphs in Figures 5 and 6 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 0.90.

In a perfect scenario with no service inversion, the first two and one-half bars would consist solely of responses in Series 1, followed by two and one-half 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.

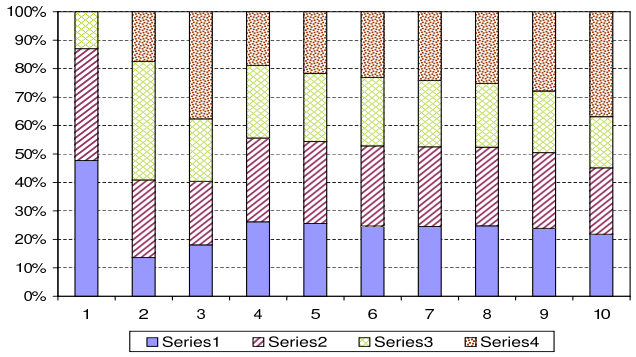

Figure 5: **Apache CDF breakdown by decile at load 0.90**

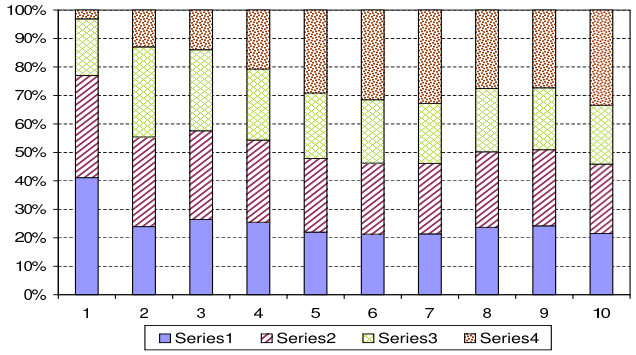

Figure 6: **Flash CDF breakdown by decile at load 0.90**

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 700 directories in our data set, the last directory receives 700 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.

3.3 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 a series of files a, b, c, d, e with file sizes $a < b < c < d < e$, if the response time of each file is in the same order of their sizes, i.e.

$$a, b, c, d, e \tag{1}$$

we define it as an ideal server with no service inversion, and a corresponding value of 0. On the contrary, if the response time is in the reverse order as their sizes, i.e.

$$e, d, c, b, a \tag{2}$$

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 the order of responses in a given CDF is from its ideal based only on 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:

$$\sum_{i=0}^{n} Distance/[n^2/2] \tag{3}$$

where distance is how far the request is from the ideal scenario, and $[n^2/2]$ is the total distance of requests in the reverse order as their sizes, which is the maximum perturbation possible. In the above example, if we receive the file in the following order:

$$b, c, a, d, e \tag{4}$$

By comparing with the order from (1), the distance of file b is 1, c is 1, and a is 2, 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.

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. Figure 7 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. Put simply, not only can requests expect queuing delays as servers get busier, but the mechanisms in the scheduler and networking stack that normally provide some degree of fairness appear to become less effective under these conditions.

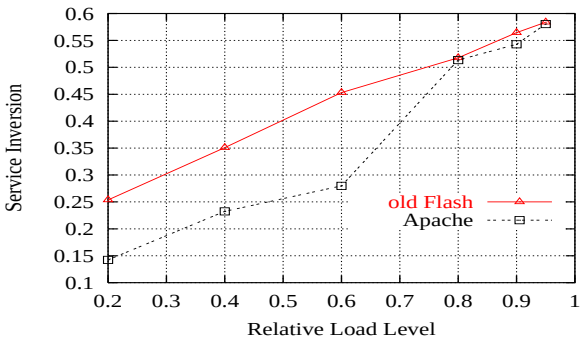

Figure 7: **Service Inversion** for Apache and Flash

4 The Origin of Excess Latency

In this section we present the origins of the service inversion phenomenon we observed in the previous sections. Specifically, we investigate factors which affect server latency both at the OS and application levels. We trace much of it to the interactions among the operating system, application, and workload, which lead to various forms of locking and blocking within the operating system. We also briefly describe the changes we make to the Flash Server to avoid these problems.

4.1 Finding Kernel Delays

To understand the load-induced behavior of the operating system and the server, we use the Flash server with an infinite-demand workload since the application-level multiplexing reduces the number of possible user/kernel interactions occurring at any time. Using the "top" tool, which shows process activity and processor consumption, we find that even with a backlog of requests, we are unable to saturate the processor, and we notice that the main Flash process is blocking inside the kernel on operations other than `select()`. We would expect that on a disk-bound workload, the disk bottleneck would prevent saturating the CPU, but since the main Flash process was designed to be non-blocking, the only conditions where it should block would be waiting for any activity inside of `select()`.

We confirm this unexpected behavior directly by using kernel profiling and indirectly by examining the burstiness of the number of events returned from each call to `select()`. With the server process blocked or waiting on locks inside the kernel, ready events are being unnecessarily delayed, leading to lower throughput and increased latency. This factor would also be a major contributor to the service inversion under load. We begin to investigate why this behavior was occurring.

In order to identify where applications spend their time in the kernel and remove unnecessary delays, we instrument the system call entry/exit points, the trap handler, and the scheduler to record unexpected activity. Among the information recorded is the time spent in the kernel, the time spent blocked, the location in the kernel where blocking occurred, and the resource contention severity. Using this approach on the Flash Web server, we find several negative interactions between the server application and the kernel which leads to the locking and blocking.

4.2 Locking and Blocking Origins

Using this system call monitoring, we find that two system calls in the Flash server are responsible for most of the blocking of the main process, and we remedy these problems by modifying the application or the kernel.

The first common problem occurs in an `open()` system call in the cache miss path of request handling, and is avoided by changing the communication mechanism Flash uses for interaction between the main process and the helpers. Flash uses main-memory caching of commonly-used files, and avoids disk access in the main process by using blocking helper processes to perform actions like converting from URLs to pathnames in the filesystem. Since the helpers and main process share the same OS, file data and metadata accessed by the helper is loaded into the filesystem cache, and is assumed to be available to the main process without blocking. The problem we observe arises when the main process attempts to open a file that has just been fetched by a helper process. While the metadata may be in memory, if the helper process has started a new request in the same directory, inode locking within the OS causes the main process's `open()` call to block while the helper finishes its disk activity. We avoid this problem by having the helper processes return open file descriptors using `sendmsg()`, eliminating duplicated work in the main process.

We discover the other major source of blocking is that the `sendfile()` system call, which is used to transfer files without copying, can exhaust its allocation of kernel virtual address space, and blocks waiting on existing transfers to free their buffers. We adopt optimizations from previous zero-copy systems [9, 20] that cache kernel virtual address buffers, thus avoiding multiple mappings for many requests to the same file. Thus the duplicated virtual memory and associated physical map (pmap) operations are eliminated.

4.3 Other Optimizations

In the process of finding these problems, we also discover the management of memory-mapped files via the `mmap/munmap/mincore` system calls consumes significant processing time, and due to the large number of mapped regions, incurs delays of several milliseconds per operation. The use of `sendfile()` also increases the number of open file descriptors, causing more work in `select()`. We address this problem by using a more efficient event-deliver call, `kevent()` [17]. With these

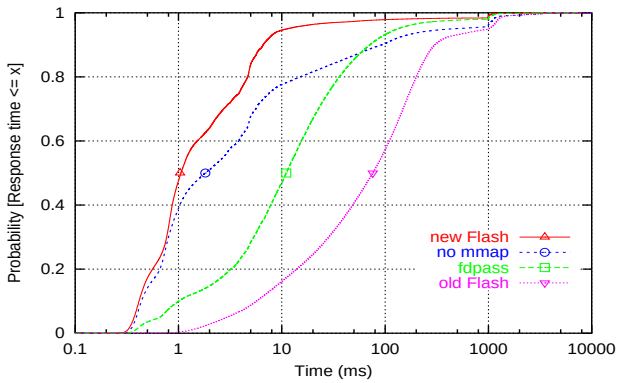

Figure 8: **CDFs for Flash Server Improvements**

changes, the memory mapped files are not strictly needed, but since these operations were being performed to test memory residency of pages to avoid blocking in the main process, eliminating them may increase the number of page faults due to pages missing from the cache. We address this problem by adding a flag to `sendfile()` which causes the call to return with an error if the pages it is trying to send are not resident and would require blocking. We modify the Flash server to use this variant of `sendfile()`. These observations regarding `sendfile()` have been communicated to the FreeBSD developer community, and work has begun to integrate our changes into a future release of FreeBSD.

5 A Low Latency Server

In this section we present the performance evaluation of the steps we made on the Flash server using the same workload as in the previous sections. We examine the latency profiles of each optimization, compare the "service inversion" with the original system, and analyze the latency characteristics of the new server.

5.1 New Latency Profiles

Using the same load level (0.90) that we used for our previous latency profiles, we evaluate the performance gains of the three optimizations described in the previous section. Figure 8 shows the latency CDF of the old Flash server (old Flash), the effects of passing file descriptors between the helpers and the main process (fdpass), the benefits of eliminating the mapped file cache (no mmap), and the new Flash server with optimized `sendfile()` (new Flash). The results exhibit more than an order of magnitude reduction in latency from the original server, with the median of the new server around 1 millisecond.

Table 2 shows the mean latency (at load level 0.90) and the throughput of the servers measured using the infinite demand model. We observe that the new Flash has a factor of 8 in mean latency improvement over the original system. One interesting observation in this table is the increase in latency (and throughput) when the memory-mapping of files is eliminated. Recall that the purpose of `mmap()` is to allow memory residency checking of files, and our workload involves significant disk access. Though we save the file mapping overhead when we eliminate `mmap()`, the system actually blocks on cache misses because of the implementation of `sendfile()`. This is also shown in Figure 8 as the lower 90% of the "nommap" latency curve beats "fdpass," but the remaining 10% takes longer because of blocking. This problem is fixed in our final system by the new flag in `sendfile()`. We also observe that our overall throughput gain is only 28%, indicating that the latency improvement do not stem purely from extra capacity.

	mean latency (ms)	Throughput (Mb/s)
old Flash	180.21	352.0
fd pass	50.01	395.0
no mmap	93.53	437.5
new Flash	23.30	450.0

Table 2: **Latencies & Throughputs for improvements**

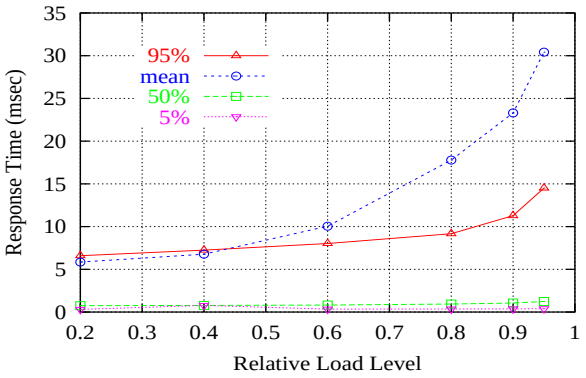

Figure 9: **Latency Profile of the New Flash**

Not only does the new server have a lower latency, but it also shows *qualitatively* different latency characteristics. Figure 9 demonstrates 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 the server serves most of the requests 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 heavy-tailed feature is common in Web workloads, and our latency now more cleanly separates small requests from large requests.

Given the fact that the mean latency gains are dominated by the tail of the response curve, it is reasonable to be concerned that the largest requests are being penalized in order to improve the latency of most requests. however, when we compare the maximum latency of the new Flash with the original, we find that this concern is not founded. Previous work on differentiated scheduling policies like SRPT [4] has drawn the conclusion that even with active scheduling, there is no unfairness to requests in the tail of these Web workloads.

5.2 Service Inversion of the New Server

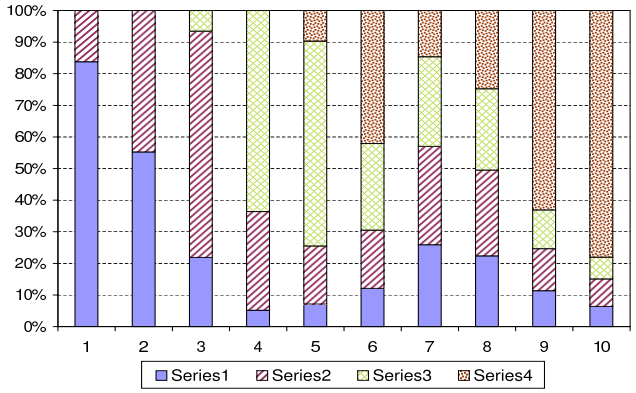

Figure 10: **CDF breakdown for New Flash, load 0.90**

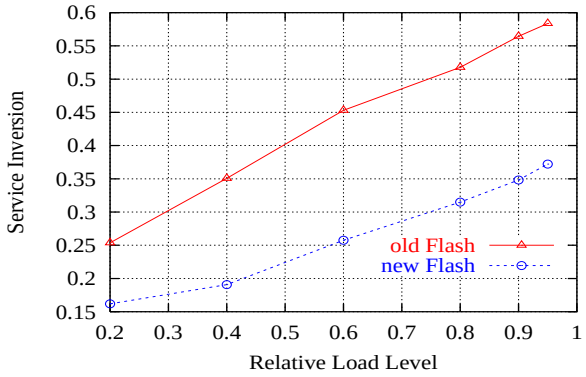

Figure 11: **Service Inversion of new and old Flash**

In order to verify the unfairness of the new server, we further examine the latency breakdown by decile for the 0.90 load level and the "service inversion" at different load levels. Figure 10 shows the percentage of each file series in each decile, and we observe some interesting changes compared to the original server. The smallest files (series 1) dominates the first two decile, 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 been seen

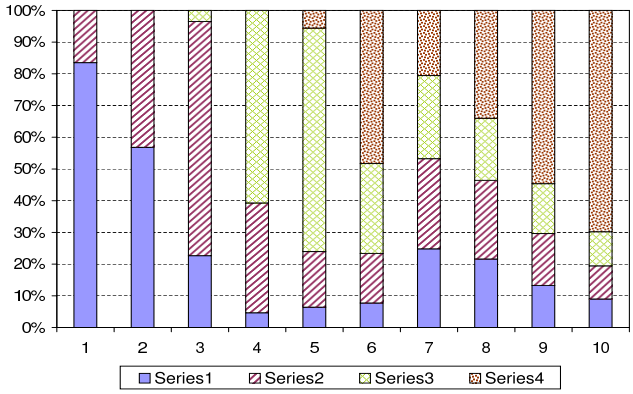

Figure 12: **In-memory workload CDF breakdown, New Flash, load 0.90**

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 11 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. One of the reasons could be "inversion" in the network queue, which we discuss in the next section.

Since we use a workload with dataset size larger than the physical memory, we should confirm that the ordering in Figure 10 is not caused by disk access of the large files. Research [2, 5] has shown that Web workloads have popular small files and a few large unpopular files, and our workload has a similar distribution. Thus, it is very likely that all of the popular small files hit in the memory cache and result in shorter response time. In order to test this hypothesis, we use a workload with a 500MB data set that fits entirely in memory. The latency breakdown shown in Figure 12, and we see virtually no difference between Figure 10 and Figure 12. The small workload has an inversion value of 0.33 and the large one has 0.35, which implies that the right ordering is not caused by cache but the elimination of locking and blocking issues.

6 The Myth of Connection Scheduling

We have demonstrated that locking and blocking in the operating system are the main origins of server-induced latency. By eliminating these issues, we show improvement in latency and service quality. We demonstrate that they are also responsible for burstiness of the load presented in the server, which is the foundation for application-level

connection scheduling. In this section we present measurements of event queue lengths in both the Flash server and the Apache server, and discuss the fact that connection scheduling is unnecessary in nonblocking systems and largely a result of other implementation artifacts.

6.1 Event Batching in the Flash Server

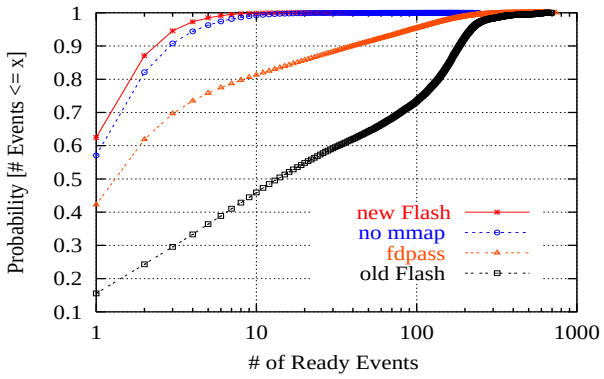

Figure 13: **Event Queue Lengths for Flash Variants**

The main prerequisite for connection scheduling is that the server receives enough requests at once in order to perform any sort of meaningful scheduling. We did observe this kind of burstiness in the original Flash server, due to the combination of batching in `select()` and the impact of delays caused by locking. Since both effects have been removed in the new Flash, we expect that events will be delivered to the application as soon as possible, resulting in fewer large bursts of requests.

We measure the number of ready events returned by the event delivery system calls `select()/kevent()`. Figure 13 shows the CDF of ready events measured in half an hour on four different server configurations with the same workload we used in previous sections, while Figure 14 shows the same data weighted by the number of events returned. We have all of the clients issuing requests in an infinite demand model to fully load the server. The original Flash server exhibits longer event queue lengths with a mean of 61 event per dispatch and a maximum length of more than 600. In contrast, the new server only returns an average of 1.6 events per call, with 99.9% of the dispatches returning fewer than 10 events. One would expect that with fewer events returned with each call, the new server dispatches ready events more frequently. Checking the number of dispatches confirms this analysis, with the new server completing almost four hundred times more calls than the original server. By eliminating the unnecessary locking and blocking in the OS, requests are being served as soon as they are ready. As long as the CPU is not overloaded, we expect that there are always fewer batched events at any time.

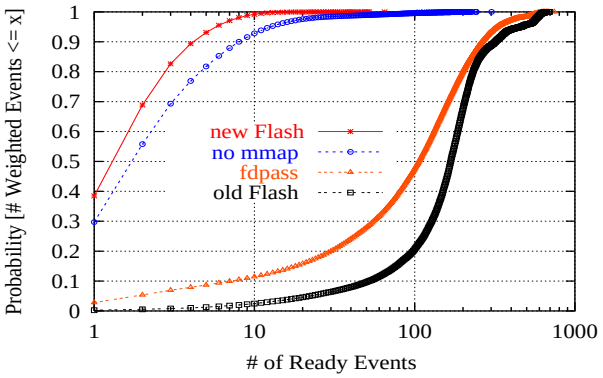

Figure 14: **Weighted Queue Lengths for Flash Variants**

Though the new server does run a bit faster than the old one, both servers are fully loaded with the clients issuing requests as fast as possible. Even with this workload, the new server exhibits only minimal burstiness. In an open-system model such as would be found outside of the lab, this workload pattern would only result from overload. In that scenario, admission control and load shedding would be mandatory for good performance. So, even in the event that this workload presents any opportunities for application-level connection scheduling, the likelihood of it having any meaningful impact outside the lab is minimal. With a properly-designed server, lower load levels would not generate enough burstiness, and higher load levels would have to be handled by overload control mechanisms. These results convince us that the opportunities for connection scheduling are largely nonexistent with a better server, and that what has been observed previously is the artifact of server implementation.

6.2 Burstiness in the Apache Server

Since we optimized only the Flash server to remove the locking and blocking issues, we are interested in knowing if the burstiness of the multi-process servers is also affected by these problems. This question motivates us to investigate the burstiness of the Apache server. We instrument the OS scheduler to export the length of runnable processes queue every second. Because each Apache process handles requests with a blocking manner, blocking is expected in the system. But locking caused by resource contention may also artificially increase the burstiness of the load, resulting in longer runnable process queue. To test our theory, we configure the Apache server with different number of maximum processes, and have all the clients fully load both of the systems.

We configure the Apache server with 256 and 1024 maximum processes and measure the number of runnable processes in each case. We show the sampling of runnable processes as dots scattered over 500 seconds. Figure 15 shows the scattering with the server configured with 256

maximum processes, and Figure 16 is the result for 1024 maximum processes. Each dot represents the number of runnable processes at that particular time. We notice that the dots in Figure 15 are more dispersed than those in Figure 16, and the number of runnable processes is always close to the maximum number when it is configured with 1024 processes. We do not see similar behavior in Figure 15 where the number of runnable processes is generally much lower than the maximum number.

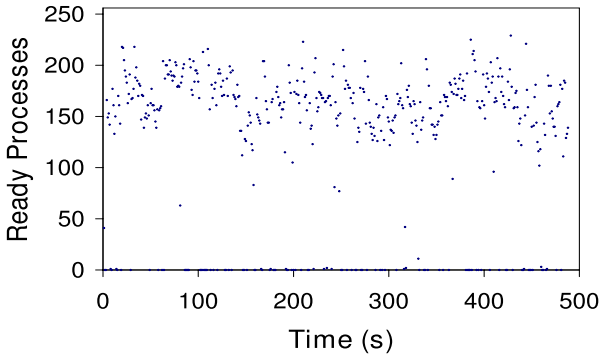

Figure 15: **Run queue length for 256 Apache processes**

Because all the clients issue requests in an infinite demand model, both systems should be fully loaded, and we confirm that both of them achieve almost the same capacity. The existence of the larger process queue in the system with 1024 processes suggests that the system may have more burstiness, and spend more time on blocking, with no processes runnable at certain times.

In order to confirm the percentage of time which is actually idle because of blocking, we normalize the queue length and generate a system load level as follows:

$$loadlevel = \#RunnableProcesses/\#Processes \quad (5)$$

Figure 17 shows the PDF of the normalized load levels over the time period. Our hypothesis that the system actually spends some time blocking is confirmed by seeing 40% of the time is in idle on the system with 1024 processes and 20% is idle on the one with 256 processes. We also observe that the Apache server with 1024 processes exhibits heavier bursts of load (but for shorter periods) than the 256 process configuration. When the server is configured with 256 maximum processes, the system has a lower load level but is more evenly distributed over the whole period.

These results indicate that the problems of locking and blocking, and the associated artificial burstiness in load, are not dependent on server architecture, but are a phenomenon caused by the OS. Though eliminating these problems in multi-process servers may be more difficult, our experience with Flash suggests the benefits may be worthwhile.

6.3 Congestion Window Effects

Previous work shows that under heavy network congestion, server latency can be reduced by giving preference to small files or requests based on the SRPT (Shortest Remaining Processing Time) and controlling the sending order of packets in the network buffer [11]. Though our new system does reduce latency significantly, we believe there is still space for improvement and these issues affect the remaining service inversion shown in Figure 10. This observation draws our attention to investigate the effect of network stack, which we address in the remainder of this section.

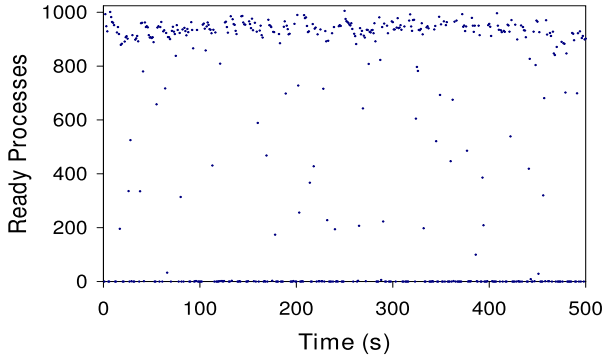

Figure 16: **Run queue length for 1024 Apache processes**

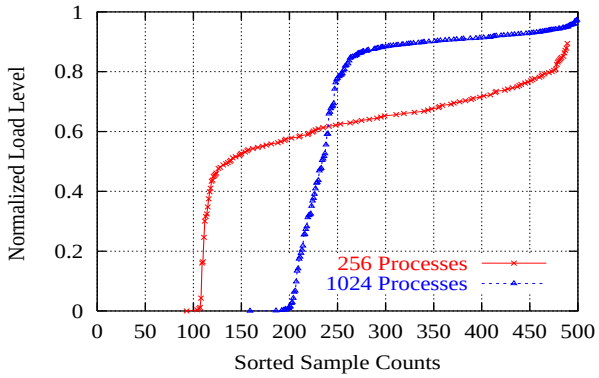

Figure 17: **Normalized load levels for Apache**

The “mice and elephants” behavior of TCP flows arises from the observation that most flows are short-lived (mice), but that the majority of bytes transferred take place in large, long-lived flows (elephants) [10]. With a median transfer size of less than 10KB, many Web transfers are complete before the TCP connection has exited the slow-start phase. However, longer transfers not only exit slow start, but can open large congestion windows over time. Persistent connections can cause a similar effect [12], since multiple transfers over a single connection will appear to TCP as a single large transfer.

The "problem" with large congestion windows is twofold – large transfers can use bandwidth unfairly, penalizing smaller transfers, and new responses sent over a connection with an enlarged congestion window will complete more quickly than if they had started with a new TCP connection. These transfers may complete more quickly than other responses, yielding queuing delays and more unfairness.

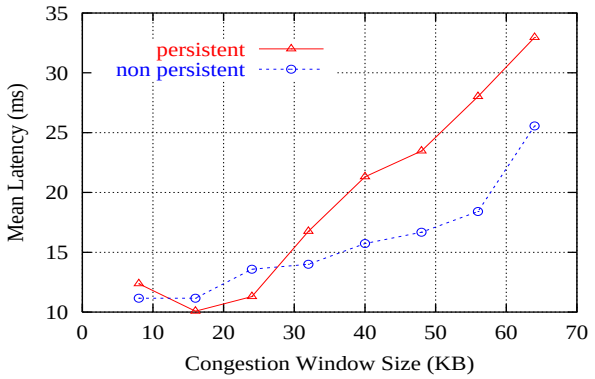

Figure 18: **Mean latency vs max congestion window size**

To experiment with this behavior, we limit the size of the TCP congestion window in our networking stack and repeat the latency experiments at the 0.90 load level. This change can be made entirely on the server side, and does not have to be visible to the application, since the socket buffer size can remain the same. The resulting mean latencies for persistent and nonpersistent connections are shown in Figure 18, and are quite surprising.

This simple approach reduces the mean latency from an already low 33ms to as little as 10ms. The results indicate that the "sweet spot" for the maximum congestion window size is roughly the average transfer size, with larger congestion windows penalizing small responses and increasing the mean response time. The "unfair" aspect of persistent connections are also visible, with the nonpersistent connections showing lower mean response time.

This experiment is merely a starting point for further investigations, since the lab conditions used to produce it are not representative of wide-area network conditions. In WAN environments, large congestion windows are needed for high bandwidth-delay pipes to be fully utilized. Likewise, persistent connections reduce the number of round-trip delays associated with connection setup and teardown. However, this experiment illustrates two important points: some of our remaining latency is due to unfairness in the network, and we can achieve benefits similar to SRPT scheduling, but with a much simpler mechanism. Interestingly, the service inversion value remains 0.33, which we believe is attributable to the benefits taking place within series 4, and therefore not being counted in our metric.

7 Related Work

Performance optimization on the network servers has been an important research area, with much work focused on improving throughput. Performance studies of the Harvest Cache [7] established the suitability of event-driven designs for network servers, and the Flash server demonstrated how to avoid some disk-related blocking [19]. Schmidt and Hu [22] performed much of the early work in studying threaded 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 [21] 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 [6]. Olshefski et al. [18] propose a way of inferring client response time by measuring server-side TCP behaviors.

Connection scheduling as a technique to reduce server latency has been proposed in several works. Most of the attention in this area has been focused on SRPT policy [8], both including server modification and kernel instrumentation for network stack scheduling [11]. Two server platforms, Haboob [25] and Knot [24], implement this approach in user space. Cohort scheduling [16] focuses on gaining performance by batching similar requests. Our work examines the assumption of these approaches, particularly the opportunities for scheduling, and find that the underlying assumptions may not exist in well-implemented servers under realistic load scenarios.

Our approach of fairness evaluation may be more suitable for network servers than the Jain fairness index [15] used in other works, since we focus more on the latencies of individual requests rather than coarse-grained characteristics of clients. Bansal & Harchol-Balter [4] investigate the unfairness of SRPT scheduling policy under heavy tailed workloads and draw the conclusion that the unfairness is barely noticeable. Our new server relies on the existing scheduling disciplines within the operating system and networking code to give small requests faster treatment, and we find that eliminating the existing obstacles yields automatic performance improvement.

8 Discussion & Conclusion

In this paper, we have examined the causes of server-induced latency in network servers, and have found that locking and blocking mechanisms not only cause high latency and burstiness, but are the major reasons behind the unfairness of existing servers. We have shown that eliminating these problems can be simple, and can yield dramatic latency reductions. The burstiness of these systems

had been previously harnessed for application-level connection scheduling, and we demonstrate that once the implementation issues causing the problems are addressed, the queue lengths diminish to the point where any extra scheduling is not only unnecessary, but also difficult to apply.

We have proposed a new metric for quantifying unfairness, the service inversion index, and we have shown how this metric can be used to understand the root causes of latency degradation under load. Through the use of this metric, we have seen that the queuing delays, long assumed to be responsible for latency growth, are only secondary factors under Web-like workloads. Our resulting latency profile looks both qualitatively and quantitatively different from other servers. Not only do we show latency reductions ranging from one to two orders of magnitude, but we also show very little degradation for most requests as load increases.

We believe that as broadband penetration increases and wide-area round-trip times decrease, more attention will be paid to server-induced latency. In this context, we advocate seriously re-examining many of the widely-held assumptions regarding the origins of latency, because as we have shown, much of the conventional wisdom seems to not be rooted in fact.

We expect to conduct further research in this area, not only to understand how our service inversion metric can be made more useful, but also to understand the limits of latency in these systems. Even with our optimizations, we did not achieve perfect fairness for all requests, and we are curious as to whether this goal is achievable without compromising overall performance.

References

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

[2] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, pages 126–137, Philadelphia, PA, Apr. 1996.

[3] 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.

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

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[6] 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.

[7] 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.

[8] 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.

[9] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 189–202, Asheville, NC, Dec. 1993.

[10] L. Guo and I. Matta. The war between mice and elephants. Technical Report 2001-005, Boston University, June 2001.

[11] 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.

[12] J. Heidemann. Performance interactions between p-http and tcp implementations. In *ACM Computer Communication Review, 27(2):65–73*, April 1997.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] J. Lemon. Kqueue: A generic and scalable event notification facility. In *FREENIX Track: USENIX 2001 Annual Technical Conference*, pages 141–154, Boston, MA, June 2001.

[18] 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.

[19] 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.

[20] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.