Understanding and Addressing Blocking-Induced Network Server Latency

Yaoping Ruan
IBM T.J. Watson Research Center
Yorktown Heights, NY
{yaoping.ruan}@us.ibm.com

Vivek Pai
Department of Computer Science
Princeton University
{vivek}@cs.princeton.edu

Abstract

We investigate the origin and components of network server latency under various loads and find that filesystem-related kernel queues exhibit head-of-line blocking, which leads to bursty behavior in event delivery and process scheduling. In turn, these problems degrade the existing fairness and scheduling policies in the operating system, causing requests that could have been served in memory, with low latency, to unnecessarily wait on disk-bound requests. While this batching behavior only mildly affects throughput, it severely degrades latency. This problem manifests itself in fairness and service quality degradation, a phenomenon we call *service inversion*.

We show a portable solution that avoids these problems without kernel or filesystem modifications, We modify two different Web servers to use this approach, and demonstrate a qualitatively different change in their latency profiles, generating more than an order of magnitude reduction in latency. The resulting systems are able to serve most requests without being tied to disk performance, and they scale better with improvements in processor speed. These results are not dependent on server software architecture, and can be profitably applied to experimental and production servers.

1 Introduction

Much of the performance-related research in network servers has focused on improving throughput, with less attention paid to latency [6, 13]. 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 directly affected service availability. The development of popular throughput-centric benchmarks, such as SPECWeb96 [19] and WebStone [12], also gave 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 geographically, reduce the distance between the client and the desired data, reducing round-trip latency. With latencies between most major cities in the mainland US on the order of tens of milliseconds, server induced latency could be a significant portion of end-user perceived latency. Some recent work addresses the issue of measuring end-user latency [3, 15], with optimization approaches mostly focusing on scheduling [5, 9, 20, 21].

However, comparatively little is understood about trends in network server latencies, or how system components affect them. Current research generally assumes that server latency is largely caused by queuing delays, that it is inherent to the system, and that scheduling techniques are the preferred solution to address them. Unfortunately, these assumptions are not explicitly tested, complicating attempts to systematically address issues of latency. Based on these observations, our goal is to understand the root causes of network server latency and address them, so that server latency can be improved. A better understanding of latency's origins can also help other research, such as improving Quality-of-Service (QoS) or scheduling policies.

By instrumenting the kernel, we find that Web servers can incur latency blocked 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. This batching behavior may have little impact on throughput, it can significantly affect latency. It causes head-of-line blocking in the OS and manifests itself as 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 reduces the fairness of response orders, 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 under load.

By addressing the blocking issues both in the application and the kernel, we improve response time by more than an order of magnitude, and demonstrate qualitatively different change in the latency profiles. The resulting servers also exhibit much lower service inversion and better fairness. These latency profiles in our resulting servers generally scale with processor speed, where cached requests are no longer bound by disk-related issues. In comparison, experiments using the original servers only show that server throughput improves with increases in processor speed, but not server latency. We believe that our solution is more portable than redesigning kernel locking, and that our findings also apply to Web proxies, where more disk activity is required and the working sets generally exceed physical memory.

The rest of the paper is organized as follow: In Section 2, we present the servers used throughout this paper, test environment, workloads, and methodology. In Section 3 we identify the latency problems and explain their causes. We introduce a new metric to quantify the effects in Section 4. In Section 5, we discuss how we address these problems, describe the resulting servers, present the experimental results on the new servers, and examine latency scalability with processor speeds. We discuss related work in Section 6 and conclude in Section 7.

2 Background

In this section we provide some background on our previous work, and describe the network servers, experimental setup, workloads and methodology since we begin our analysis with experimental measurements of the servers. Our earlier work on performance debugging tools [17] examined blocking in servers, but did not specifically try to understand the origins of latency, our main contribution in this work.

2.1 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. To represent widely-deployed general-purpose servers, we use the multi-process Apache server [1], version 1.3.27. To test high-performance servers, we use the event-driven Flash Web Server [13], a research system with aggressive optimizations. Where appropriate, we test two versions of Flash – one using the standard `select()` system call for event delivery, as well as one that uses the more scalable `kevent()` event-delivery mechanism coupled with the zero-copy `sendfile()` system call.

The Apache server utilizes blocking system calls and relies on the operating system's scheduling policy to provide parallelism, while Flash uses event delivery mechanism to multiplex all client connections. Flash consists of a single main process using non-blocking sockets, and a small set of helper processes performing disk-related operations. 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.

Processor	P-II	P-III	P4 Xeon
Speed (MHz)	300	933	3000
Bcopy bandwidth (MB/s)	93	265	624
Read bandwidth (MB/s)	213	555	1972
Memory latency (ns)	245	101	116

Table 1: Server hardware information – hardware characteristics of three generations of the Intel Pentium processor, with some values measured by lmbench [11]

In our experiments, both servers are configured for maximum performance. In Flash, the file cache size is set to 80% of the physical memory, with remaining parameters automatically adjusted. We also aggressively configure Apache – periodic process shutdown is disabled, reverse lookups are disabled, the maximum number of processes is raised to 2048, and access logging is disabled in both servers.

2.2 Experimental Setup

Our main test platform is a uniprocessor 3.06GHz Pentium-4 with 1GB physical memory, one 5600 RPM Maxtor IDE disk, and a single Netgear GA621 gigabit Ethernet network adaptor. 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, 64K file descriptors per process, 64KB socket buffers, 80K mbufs, 40K mbuf clusters, and 16K inode cache entries. We also investigate latency scalability using three hardware platforms which 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 disk and network interface. The details of our server machines are shown in Table 1, with measured values provided by lmbench [11].

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 [19] benchmarks. These workloads are modeled after the access patterns of multiple Web sites,

Data Set Size	Top 50 %	Top 90 %	Top 95 %	Top 99 %
1024	2.1	39.5	64.6	138.3
2048	3.0	72.9	123.6	262.8
3072	4.4	101.8	181.2	385.7
4096	4.9	131.8	235.0	505.0

Table 2: SPECWeb's popularity distributions. All sizes shown in MB. Sizes do not scale linearly with the total data set size because directories are weighted using a Zipf popularity distribution

with file sizes ranging from 100 bytes to 900 KB, and are the *de facto* standards in industry, with more than 200 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. The strong bias toward small files leads to the result that the most popular files consume very little aggregate space. Table 2 illustrates this heavy-tail feature well – the most popular 99% of the requests occupy at most 14% of the size of data set.

SPECWeb 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 [21] and Knot [20], we use their parameters of a 3GB data set and 1024 simultaneous connections. With this data set size, most requests can be served from memory while a small portion will cause disk access. 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 to SPECWeb99's quality-of-service requirements.

2.4 Measurement Methodology

To understand how load affects response time, we measure latencies at various requests rates. Each server's maximum capacity is determined by having all clients issue requests in an infinite-demand (saturation) model, which is defined as load level of 1, and then relative rates are reported as load fractions relative to the infinite demand capacity of each server. This process simplifies comparison across servers, though it may bias toward servers with low capacity. Response time is measured by recording the wall-clock time between the client starting

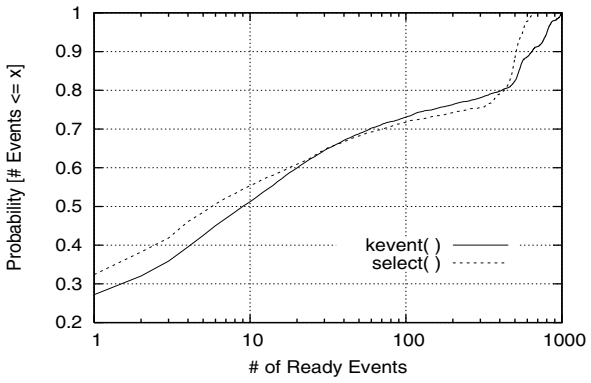

Figure 1: CDF of number of ready events (the return values from `select()`) in Flash

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 distribution. Where appropriate, we also provide the cumulative distribution function (CDF) of the client-perceived latencies.

3 Blocking in Web Servers

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.

3.1 Observing Blocking in Flash

Using our workloads, we find that the main Flash process is blocking inside the kernel on operations other than the `select()` or `kevent()` and the system shows idle CPU time. While CPU idle time is not surprising for a workload that accesses disk, the main process in Flash should never block – all the disk activity should be channeled to the helpers.

Examining the number of ready file descriptors returned per invocation of `select()` or `kevent()` provides more evidence of blocking. These calls form the main loop of an event-driven server, and are invoked as many times as needed as long as the system is active. Event handlers take corresponding actions based on the returned file descriptor value and action indicator. The number of ready descriptors returned by the `select()`

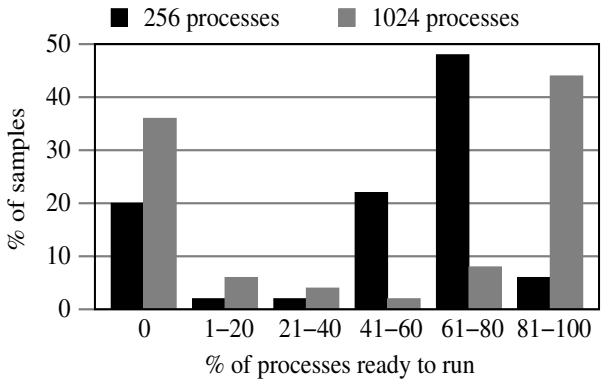

Figure 2: Scheduler burstiness (via the instantaneous run queue lengths) in Apache for 256 and 1024 server processes

or `kevent()` call reflects the queue length which will be processed by event handlers. The CDF of the number of ready descriptors is shown in Figure 1, and indicates that these calls typically return a large number of ready events per call. For `select()`, 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. The distribution for `kevent()` is similar.

In this workload, the CPU should never be idle – even if the amount of work available decreases, the main loop should call `select()` or `kevent()` more often, decreasing the number of ready descriptors per call. Only when one ready descriptor is returned per call should the CPU exhibit any idle time. However, given the idle time and the observed 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 performance and increasing latency.

3.2 Inferring Blocking in Apache

Directly observing a similar problem in Apache is more difficult because any of its processes may block on disk activity, and its multiple-process design exploits the fact that the OS will schedule another process when the running process blocks. While conventional wisdom holds that such blocking is necessary and affects only the request being handled, excess blocking may hinder parallelism and cause high latency.

Since Apache does not have easily-testable invariants regarding blocking such as Flash does, we 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,

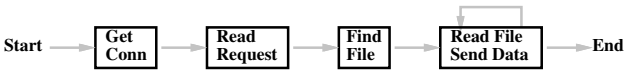

Figure 3: HTTP request processing steps

it would block other processes requesting the same resource, and the release of a resource would involve several processes becoming runnable at the same time. We expect that as more processes are involved, burstiness increases as does run queue variability.

We instrument the OS scheduler to report the number of runnable Apache processes, and test in two configurations. We use 256 and 1024 maximum server processes, an infinite-demand workload, and 1024 clients. Both configurations show roughly the same throughput, due to the infinite-demand model and LAN clients. In Figure 2, we show what percentage of the Apache processes are runnable at any given time.

In both cases, the distribution is very bimodal – most of the time, either no Apache processes are runnable or most of them are. The burstiness, when many processes suddenly become runnable at once, is more evident in the 1024 process case – 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 only slightly less bursty, 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 in the kernel's "bottom-half."

3.3 Causes of Blocking

Our earlier work on developing the DeBox tool [17] identified the call sites in Flash where blocking occurred, but did not investigate the mechanisms by which it occurred. Among the problems, we identified that the Flash server would sometimes block in the "find file" step of the HTTP processing pipeline shown in Figure 3. This step involves performing a series of `open()` and `stat()` calls to traverse the URL's components in the filesystem. This blocking was unexpected because of the way Flash opens files – it invokes a helper process to perform the steps first, and then the helper notifies the main process, which repeats the process. In this case, the helper had presumably just finished this process, so all of the necessary metadata should have been memory-resident when the main process performed the same actions. This blocking occurs even if the filesystem is mounted asynchronous or read-only, ruling out synchronous metadata writes.

Further investigation reveals that the metadata locking problem is due to lock contention during disk access. In particular, we find that one of the problems is lock con-

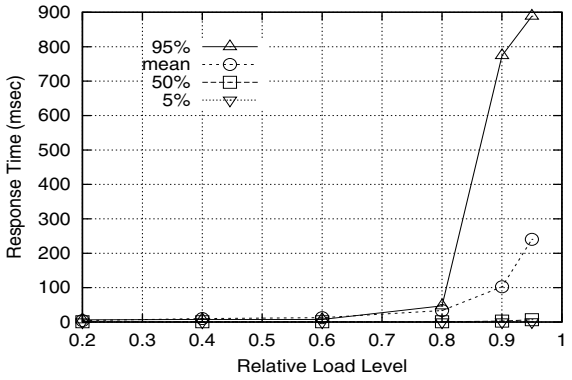

Figure 4: Apache Latency Profile. The relative load of 1.0 equals 241 Mb/s

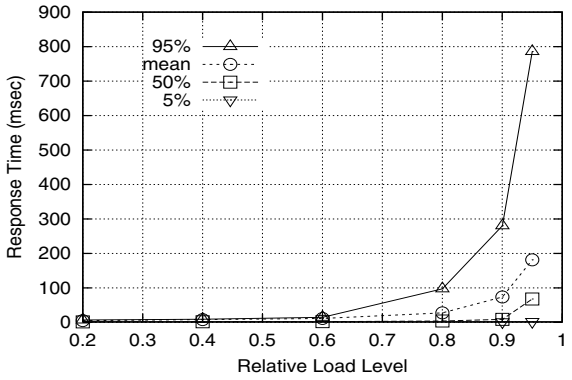

Figure 5: Flash Latency Profile. The relative load of 1.0 equals 336 Mb/s

tention when the main process and the helper access a shared file path. When this happens, the helper usually is doing disk I/O but still holding the vnode name lock to ensure the consistency of the corresponding entry. The decision to make this lock exclusive instead of read-only appears to be a design decision to simplify the associated code – in most types of code, the probability of lock contention would be low, so making this lock exclusive simplifies the code. We further validate this theory by confirming that the blocking occurs even when access time modifications are disabled and even when the filesystem is mounted read-only.

The problem of metadata handling is not FreeBSD-specific. We observe little lock contention in Linux but have observed metadata cache misses commonly occurring when the data set exceeds the physical memory size, causing blocking in otherwise cached requests.

The degree of this problem is significant in FreeBSD due to an interaction between a number of implementation choices. The choices and possible motivation for each are as follows:

- **Exclusive vnode locks** – FreeBSD often uses exclusive vnode locks, presumably to reduce complexity and also to avoid possible deadlock scenarios related to lock promotion.
- **Directory walk locks** – When walking a directory to find a file, the OS acquires the child directory's lock before releasing the parent's lock.
- **Locks during disk access** – If getting the child's inode requires a disk access, the parent's lock is held during the disk access. Releasing locks and re-acquiring them after the disk access causes extra work, and is subject to problems if a higher-level path component changes during the disk access.

These choice are independently reasonable, but their combination leads to the unintended blocking. In particular, if multiple processes are trying to resolve similar paths, and one blocks on an inode access, the others can block waiting on an exclusive lock for the shared parent. If more processes try to resolve the same path, they can block higher in the file tree waiting on other readers to release lower-level exclusive locks. A single inode read can then cause many readers to become unblocked, leading to a burst of activity in the form of ready processes.

The metadata locking problem also explains what occurs in Apache and why it has gone unnoticed for so long. Since Apache does not cache open file descriptors, every request processed must perform this same set of steps. The design relies on the OS's own metadata caching to avoid these steps requiring excessive disk access, but without any information about which accesses should be cached, Apache developers can not determine when blocking during an `open()` call is unexpected.

3.4 Response Time Effects

To measure server latency characteristics on disk-bound workloads and show the impact of the underlying blocking problems, we run the servers with request rates of 20%, 40%, 60%, 80%, 90%, and 95% of their respective infinite-demand rates. The results, shown in Figures 4 and 5, show some interesting trends. 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 (95% of the infinite-demand rate). 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.

Some insight into the latency degradation for these servers can be gained by examining the spread of request latencies at the various load levels, shown in Figures 6 and 7. Both servers exhibit latency degradation as

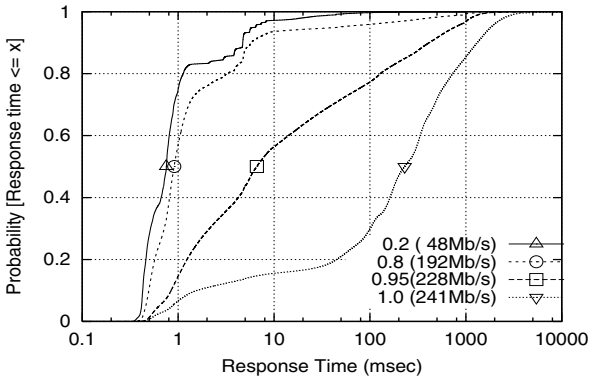

Figure 6: Apache latency CDFs for various load levels

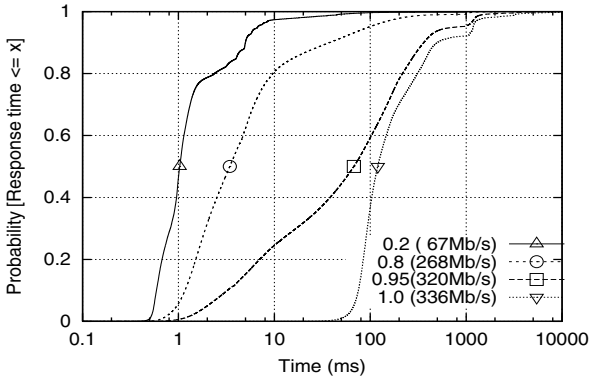

Figure 7: Flash latency CDF for various load levels

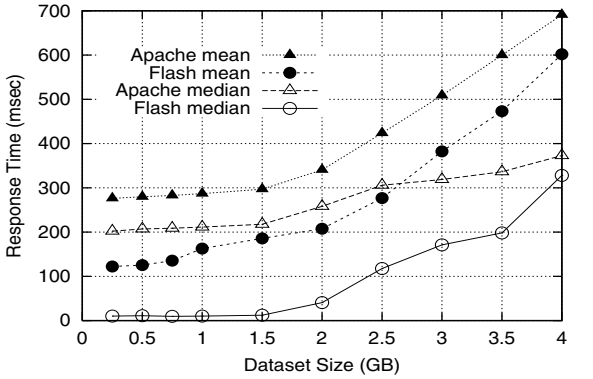

Figure 8: Median and mean latencies of Apache and Flash with various data set sizes

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 and blocking effects are immediately apparent – the relative smoothness of the Flash curves, and the seemingly lower degradation for Apache at or below load levels of 0.95. By multiplexing all client connections through a single process, the Flash server introduces some batching effects, particularly when blocking occurs. This batching causes even the fastest responses to be delayed. As a result, Flash returns very few responses in less than 10ms when the load exceeds 95%, whereas Apache still delivers over 60% of its responses within that time. We believe that under low lock contention, Apache's multiple processes allow in-memory requests to be serviced very quickly without interference from other requests. At higher loads, locking becomes more significant, and only 18% of requests can be served within 10ms.

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.

3.5 Response Time vs. Data Set Size

A deeper investigation of the effect of data set size on server latency provides more insight into the blocking problems as well as a surprising result. Figures 8 shows mean and median latencies as functions of data set size. 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, 1GB. 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 measured cache hit rate is more than 99%, suggesting that most requests should be comfortably served out of the filesystem cache. The cache hit rate is in line with what we showed in Table 2. These tests confirm that the small amount of cache miss activity is interfering with accesses 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.

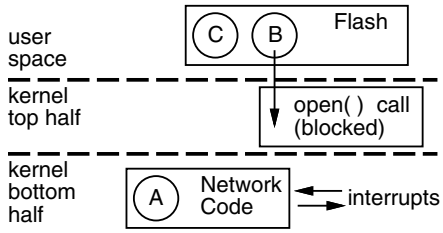

Figure 9: Service inversion example – Assume three requests (A, B, and C) arrive at the same time, and A is processed first. If it is cached and is sent to the networking code in the kernel bottom half, interrupt-based processing for it can continue even if the the process gets blocked. In this case, even if A is large, it may get finished before processing on C even starts.

4 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, similar to the priority inversion problem in scheduling. 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 and propose an approach to quantify the service inversion value.

Since certain request processing steps operate independently of the server process, any blocking that occurs early in request processing can affect the system's fairness policies. 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 kernel's "bottom half." 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.

Head-of-line blocking in the literature is usually studied in the network scheduling context. To understand the blocking scenario in the OS and how it causes service inversion, consider the scenario in Figure 9, where three requests arrive simultaneously, with the middle request causing the process to block. Assume it is blocked by an `open()` call, which takes place 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 system's fairness policies, particularly the scheduling of network packets, are not given a chance to operate since the three requests do not reach the networking code at the same time.

If the requests before the blocked requests are larger than the ones that follow, 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.

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. Percentages shown are the dynamic request frequencies for the given file sizes in the SPECWeb99 workload.

4.1 Identifying Service Inversion

To qualitatively understand the prevalence of service inversion, we take the latency CDFs from Figures 6 and 7 and split them by decile. Since SPECWeb biases toward small files and more than 95% of the requests could fit into physical memory, ideal response times would be roughly proportional to transfer sizes. By examining the different response sizes within each decile, we can estimate the extent of reordering. To simplify the visualization, we group the responses by sizes into four series such that their dynamic frequencies are roughly equal. The details of this categorization are shown in Table 3.

The graphs in Figures 10 and 11 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.

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, both graphs show responses from the different series spread across all deciles, suggesting both servers exhibit service inversion. 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 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

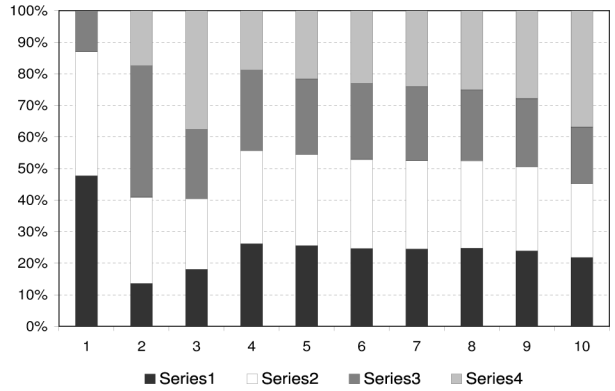

Figure 10: Apache CDF breakdown by decile at load level 0.95

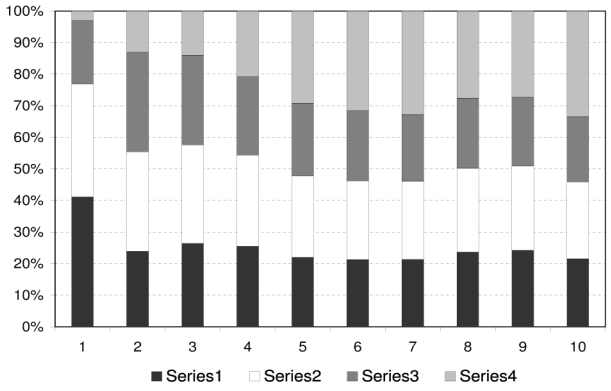

Figure 11: Flash CDF breakdown by decile at load level 0.95

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 directory bias 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 is relatively minor, as we will show later in the paper.

4.2 Quantifying Service Inversion

While the latency breakdowns by decile qualitatively show the system's unfairness, 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 as follows: we want to determine how perturbed a measured order is, compared with the order of the response sizes. Perturbation is the difference in position of a response in the ordered list of response times versus its position in a list ordered by size, where the per-response distances are 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.

	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

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.

5 The New Servers & Results

In this section we describe our solution and evaluate the resulting systems. We analyze the effects on capacity, latency, and service inversion, and demonstrate that our

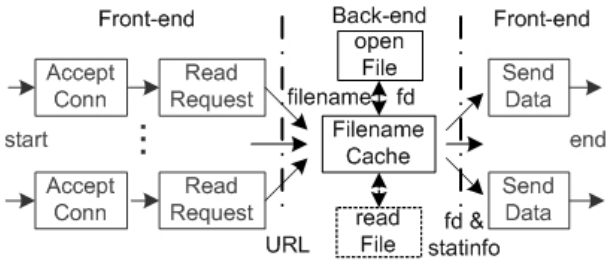

Figure 12: Flashpache architecture

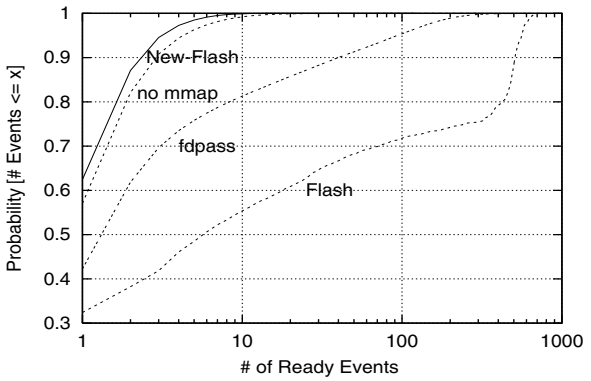

Figure 13: CDFs of # of ready events for Flash variants, infinite-demand workload

new servers overcome the latency and blocking problems previously observed. In our earlier work on DeBox [17], we modified the Flash Web server to avoid blocking. We briefly describe those changes to provide the context for our new results with Apache.

Since the blocking has multiple origins, we believe a portable user-level process is preferable to invasive kernel changes. Accordingly, we modify both servers to reduce blocking. Our new contribution in this respect is to identify how Apache can be easily modified to take advantage of the same kinds of changes that helped Flash. Additionally, we focus on latency and service quality evaluation of the resulting servers, in order to understand how the new techniques work.

Our earlier changes to Flash focus on detecting and avoiding blocking, or moving blocking out of the main server process. The `open()` and `stat()` calls are moved entirely out of the main process, and the helpers return file descriptors and metadata information to the main process using the `sendmsg()` system call. Thus, the main process can operate continuously without blocking. Data copying is eliminated by using `sendfile()` instead of `writev()`, and the memory-mapping calls that were used in conjunction with `writev()` are eliminated. Finally, some other changes are made to `sendfile()` to reduce its memory usage and eliminate disk access. We term the resulting server New-Flash.

5.1 Flashpache

Due to the differences in software architecture, we cannot directly employ the same techniques that we used in New-Flash to improve Apache. However, given our earlier measurements on Apache, we can deduce that filesystem-related calls are likely to block, and with these as candidates, we can leverage the lessons from Flash. Since Apache does not cache file descriptors, 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 step, in which metadata-related system calls occur. 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 is responsible for 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 control back to the Apache processes, thus reducing the number of context switches and the chances of blocking.

We call this new server Flashpache, to reflect its hybrid architecture. The changes involved in this process are relatively small and isolated – fewer than 100 lines of code are modified in Apache, and half of this count is code taken directly from New-Flash. The backend process is similarly derived from parts of New-Flash, and consists of roughly 200 lines of code changes.

This architecture, shown in Figure 12, eliminates unnecessary blocking in two ways. First, in Flashpache, most of the disk access is performed by a small number of helper processes controlled by the backend, reducing the amount of locking contention. This observation is confirmed by the fact that less blocking occurs in Flashpache than in Apache with the same workload. Second, since the backend caches metadata information and keeps files open, it effectively prevents metadata cache entries from being evicted when memory pressure is an issue. However, we do not observe the CPU reduction from caching as the main source of the benefit – the interprocess communication cost between the Apache processes and the backend is almost equivalent to or even a little higher than the original system calls.

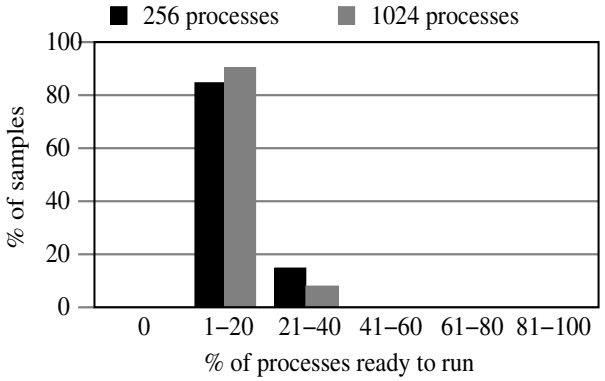

Figure 14: Scheduler burstiness in Flashpache for 256 and 1024 processes, infinite-demand workload

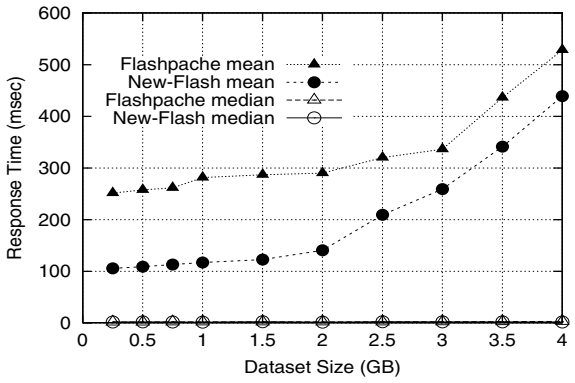

Figure 15: Response times for new servers with different data set sizes and infinite-demand workload

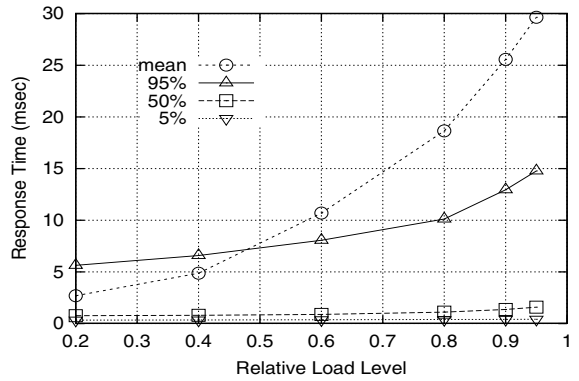

Figure 16: Latency profile of New-Flash (Flash profile shown in Figure 5). Load level 1.0 equals 450 Mb/s

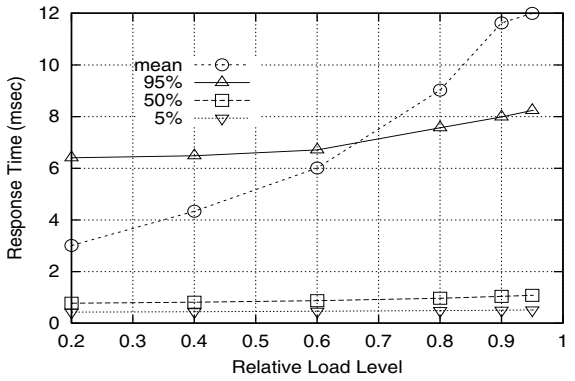

Figure 17: Latency profile of Flashpache (Apache profile shown in Figure 4). Load level of 1.0 equals 273 Mb/s

	Latency (ms)			Capacity
	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 all servers

5.2 Latency Results

We analyze the latency of the new servers by repeating our earlier experiments to understand latency and blocking. We begin by repeating the burstiness measurement, which indicates that blocking-induced burstiness has also been reduced or eliminated in both servers. In Figures 13 and 14, we see that 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. In both cases, the request batching and associated idle periods are eliminated.

We evaluate step-by-step improvements to Flash 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). Throughputs are measured with infinite-demand and response times are measured at 0.95 load level. We can see that the overall capacity of Flash has increased by 34% for this workload, while Apache's capacity increases by 13%.

The more impressive result is the reduction in latency, even when run at these higher throughputs. Flash sees improvements of 40x median, 6x mean, and 54x in 90^{th} percentile latency. Eliminating metadata-induced blocking has improvements of 5.8x median, and 3.6x mean, and eliminating blocking in `sendfile()` reduces a factor of 3 in mean latency. Apache sees improvements of 6x median, 15x mean, and 72x in 90^{th} 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

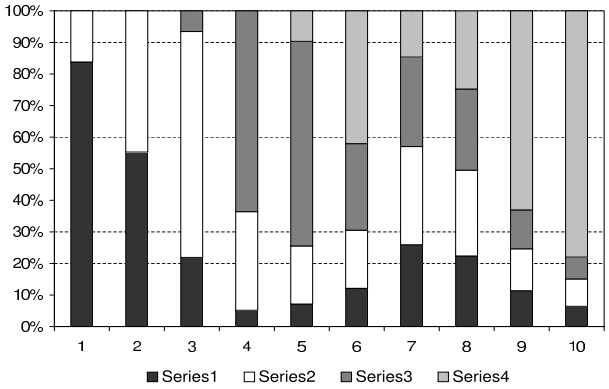

Figure 18: CDF breakdown for New-Flash on 3.0 GB data set, load level 0.95

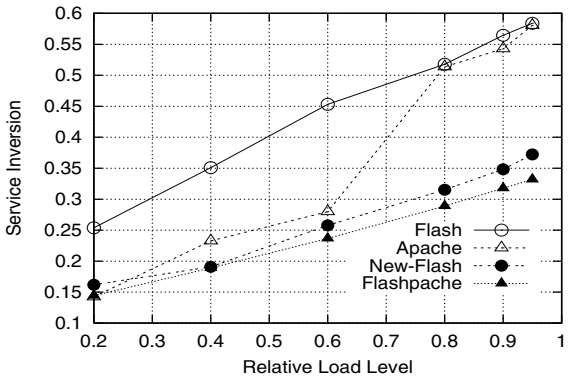

Figure 19: Service inversion of original and modified servers

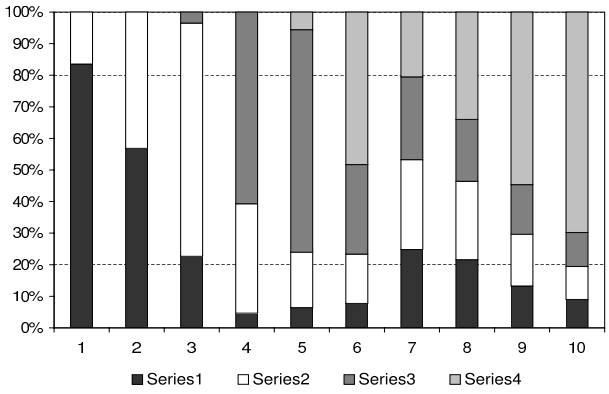

Figure 20: CDF breakdown for New-Flash on in-memory workload, load level 0.95

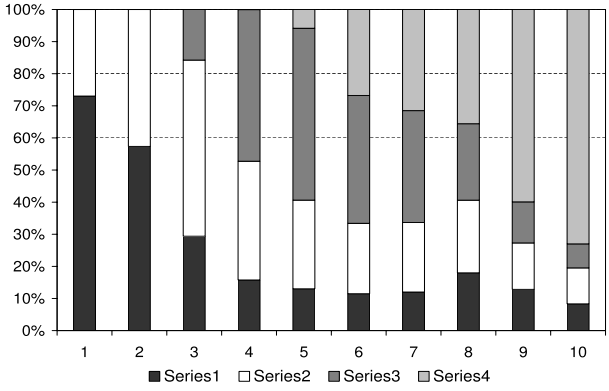

Figure 21: CDF breakdown for Flashpache on 3.0 GB data set, load level 0.95

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.

Not only do the new servers have lower latencies, but they also show qualitatively different latency characteristics. Figure 15 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 16 and 17 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 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, because the time spent on the largest requests (the last 5%) is much higher than the time spent on other requests, as expected from Table 2.

5.3 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 18 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) dominate 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 we saw earlier. Some small responses still appear in the last column, but these may stem from files with low popularity incurring cache misses. Also complicating matters is that the absolute latency value is now below 10ms for 98% of the requests, so the first nine deciles are very compressed. This observation is verified by calculating the service inversion value.

Figure 19 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

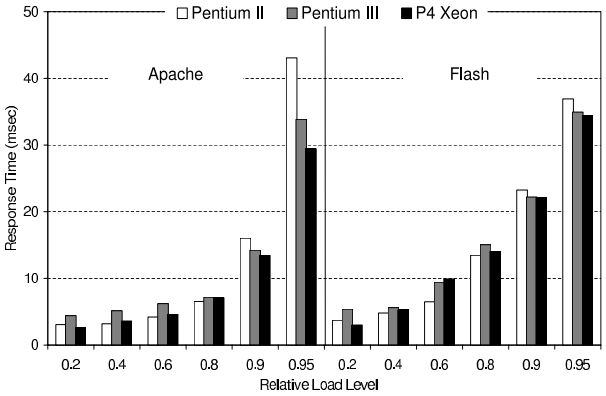

Figure 22: In-memory workload (0.5 GB) latencies of Apache and Flash across three processor generations

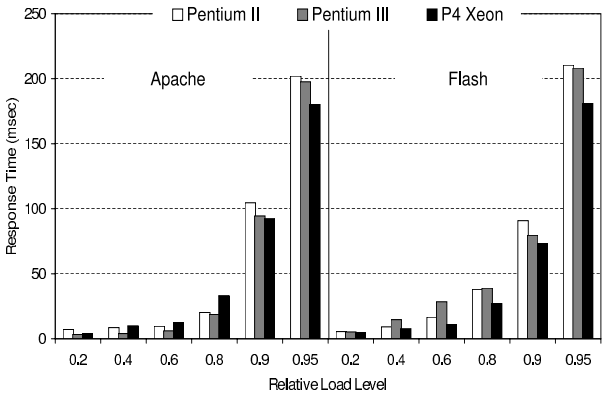

Figure 23: Disk-bound workload (3.0 GB) latencies of Apache and Flash across three processor generations

	Pentium II	Pentium III	Pentium 4
In-memory workload (0.5GB) capacity in Mb/s			
Apache	107.3	248.4	437.6
Flash	210.3	466.0	787.0
Disk-bound workload (3.0GB) capacity in Mb/s			
Apache	98.8	174.1	241.1
Flash	134.1	256.4	336.0
Flashpache	103.3	198.9	272.9
New-Flash	140.4	358.0	450.0

Table 6: Capacities of original and modified servers across three processor generations and two workloads

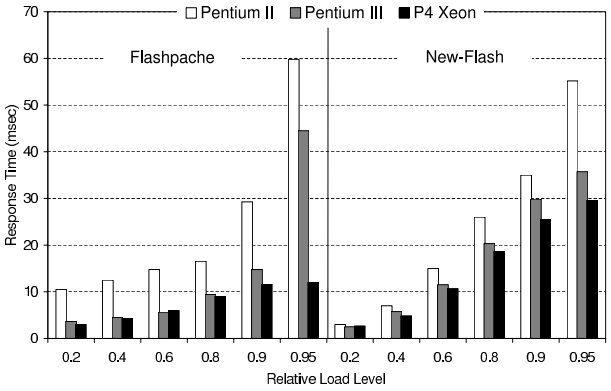

Figure 24: Disk-bound workload (3.0 GB) latencies of New-Flash and Flashpache across three generations

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. Figure 20 shows 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 21, is similar. The values for Flashpache and its original counterpart are also shown in Figure 19, and we can see that our modifications have almost halved the inversion under high load.

5.4 Latency Scalability

To understand how latencies are affected by processor speed, we use three generations of hardware with various processor speed but sharing most of the other hardware components. Details about our server machines are shown in Section 2. We begin our study by measuring the infinite-demand capacity of the two original servers while adjusting the data set size. The results, shown in Table 6, indicate that in-memory capacity of both Apache and Flash scales well with processor speed. But once the data set size exceeds physical memory, performance degrades. Even though the heavy-tailed 3GB Web workload only requires reasonable amount of disk activity, we observe the two faster processors have idle CPU, suggesting performance is tied to disk performance on this workload.

A more detailed examination of server latency is shown in Figures 22 and 23. 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 pro-

cessor speed. Figure 24 shows the scalability of our new servers across processors – even with much lower Pentium-II latencies, improvements in processor speed now reduce latency on both servers. This result confirms that once blocking is avoided, the servers can take more advantage of improvements in hardware performance.

In summary, both new servers demonstrate lower initial latencies, slower latency growth, 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, memory, etc. The fact that these changes eliminate over 80% of the latency answers the question about latency origins – these latencies were dominated by blocking, rather than request queuing.

6 Related Work

Performance optimization of network servers has been an important research area, with much work focused on improving throughput. Some addressed coarse-grained blocking – e.g. Flash [13] demonstrated how to avoid some disk-related blocking using non-blocking system calls. Much evaluation about disk I/O associated overheads has focused on Web proxies [10]. Some of the most aggressive designs have used raw disk, eliminated standard interfaces, and eliminated reliable metadata in order to gain performance [18]. In comparison, we have shown that no kernel or filesystem changes are necessary to achieve much better latency, and that these techniques can be retrofitted to legacy servers with low cost.

More recently, much attention is paid to latency measurement and improvement. Rajamony & Elnozahy [15] measure the client-perceived response time by instrumenting the documents being measured. Bent and Voelker explore similar measurements, but focus on how optimization techniques affect download times [3]. Improvement techniques have been largely limited to connection scheduling, with most of the attention focused on the SRPT policy [4, 5]. Our work examines the root cause of the blocking, and our solutions subsume any need for application-level connection scheduling. Our new servers use the existing scheduling within the operating system, and the results suggest that eliminating the obstacles yields automatic improvement with existing service and fairness policies.

Synchronization-related locking has been a major concern in parallel programming research. Rajwar et al. [16] proposed a transactional lock-free support for multi-threaded systems. The reasons of locking in our study have a broader range and differ in application domain. While head-of-line blocking is a well-known phenomenon in the network scheduling context, e.g. Puente et al. [14] and Jurczyk et al. [8] studied various blocking issues in network environment, we demonstrate that this phenomenon also exists in network server applications and has severe effects on user-perceived latency.

This paper also takes a different approach to fairness than other work, and the difference may be important in some contexts. The SEDA approach [21] tried to schedule requests based on size, but no measurements are presented on the effectiveness of the scheduling itself. Interestingly, despite the four orders of magnitude variation in SPECWeb's file sizes, SEDA handles 80% of requests in 200-1000ms, with a median of over 500ms, over 10 times slower than Flash or Apache. This uniformly slow response time gives it a high score on the Jain fairness index [7] when fairness is evaluated on a per-client basis. On a per-request level, however, we believe that shorter responses should be served faster, and believe that our service inversion metric is more useful. With coast-to-coast latencies in the continental US on the order of 100ms, and with news sites (Yahoo, CNN, etc.) routinely having over 100 embedded objects per page, SEDA's server-induced latency would be a noticeable problem for real Web use.

At the other extreme, Bansal & Harchol-Balter [2] investigate the unfairness of the SRPT scheduling policy under heavy-tailed workloads and draw the conclusion that the unfairness of their approach is barely noticeable. By addressing the latency issues directly rather than scheduling around them, our approach removes the need for the application to explicitly schedule connections. Network scheduling can still be used, particularly for traffic shaping, prioritization, etc.

Finally, we should mention that the FreeBSD SMPng effort, which is released as FreeBSD 5, has completely rewritten much of the locking in FreeBSD, using finer-granularity locks to improve performance on multiprocessor machines. Additionally, some of the filesystem locking appears to have introduced read-shared locks in addition to exclusive locks. These locks could reduce the chance of lock convoys in pathname resolution, eliminating some of the blocking we observed. While we would like to evaluate the behavior of this system, we have been unable to operate it under sufficient load without kernel panics or significant performance degradation.

7 Conclusion

In this paper, we have examined server latency and traced the root of much of the problem to head-of-line blocking within filesystem-related kernel queues. This behavior may have little impact on throughput, but severely degrades latency and service quality. 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 with 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. We performed these changes in user space, in a portable manner, without requiring any modification to the kernel or filesystem layout. Without much effort or extensive modification, we were able to take advantage of these changes in a widely-deployed legacy server. The resulting servers also exhibit lower burstiness, and more fair request handling. Their latency values scale better with improvements in processor speed than their original counterparts, making them better candidates for future improvements. Finally, our results suggest that most server-induced latency is tied to blocking effects, rather than queuing.

In addition to the practical benefits of this research, the delivery of servers with much better latency properties, this work also improves on our fundamental understanding of the interactions between the filesystem, application, and workloads. By addressing the root causes of latency increase in network servers, we believe that we can enhance research in other areas, such as improving quality of service or scheduler policies.

Acknowledgments

We would like to thank the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF grant CNS-0519829.

References

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

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

[3] L. Bent, Geoffrey, and M. Voelker. Whole page performance. In *7th International Workshop on Web Content Caching and Distribution (WCW'02)*, Boulder, CO, Aug. 2002.

[4] 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*, Boulder, CO, Oct. 1999.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] E. P. Markatos, M. Katevenis, D. N. Pnevmatikatos, and M. Flouris. Secondary storage management for web proxies. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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*, San Francisco, CA, March 2001.

[16] 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.

[17] Y. Ruan and V. Pai. Making the "box" transparent: System call performance as a first-class result. In *Proc. of the USENIX 2004 Annual Technical Conference*, Boston, MA, June 2004.

[18] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage management for web proxies. In *Proc. of the USENIX 2001 Annual Technical Conference*, pages 203–216, Boston, MA, 2001.

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

[20] 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.

[21] 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.