Understanding and Improving Modern Network Server Performance

Yaoping Ruan

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2005

Abstract

This thesis presents network server performance analysis and improvement at the operating system (OS), application, and processor levels. At the kernel level, we develope a profiling tool that provides rich OS transparency at low cost, by exposing system call performance as a first-class result via in-band channels. Using this tool on the Flash Web server running the standard SPECweb99 benchmark reveals a series of negative interactions between the server application and the OS. Some of the solutions to these issues have lead to a set of kernel patches to improve networked file transfer, and others contribute in server application design.

At the application level, we redesign the Flash server and the widely-used Apache server, improving Flash's SPECweb99 score by a factor of four and reducing response time by one to two orders of magnitude on both servers. Using these servers, we then examine server latency under load and trace the root cause of server-induced latency to head-of-line blocking within filesystem-related kernel queues. This behavior, in turn, causes batching and burstiness, and gives rise to a phenomenon we call *service inversion*, where requests are served unfairly with long responses served ahead of short responses. Removing blocking not only reduces response time under load and improves latency profiles, but also mitigates burstiness and improves request handling fairness. The resulting servers show better latency scalability with processor speed, making them better candidates for future improvements.

Finally, we investigate the architectural aspects of server performance, conducting detailed analysis of delivered simultaneous multithreading (SMT) systems. Using five different software packages and three hardware platforms, experimental results show that the benefits of the current SMT implementation on Intel Xeon processors are modest for network servers, and short memory latency or extra L3 cache helps SMT yield better speedups. By performing microarchitectural evaluation using processor performance counters, we also provide insight into the instruction-level resource bottlenecks that affect performance on these platforms. Finally, we compare the measured results with similar studies performed using simulation, and discuss the feasibility of these simulation models, both in the context of current hardware, and with respect to future trends.

Acknowledgments

It has been a tremendous privilege to work with Professor Vivek Pai over the past five years. Many thanks to him for having faith in me when I was new in the program, and tailoring different training strategies at each stage of my intellectual development; for his hands-on guidance to set me on the right track, and the crucial enlightenment he readily offered from time to time; for the heart-warming leads in life and career path, and of course, the uninterrupted generous financial support.

I am deeply indebted to my committee members. Professor David August provided critical feedback to many of the thesis projects. Professor Brian Kernighan helped polishing my paper writing and even my research statement. Professor Larry Peterson was always like a supervisor for me, providing advice and encouragement directly and indirectly. I was fortunate to have the opportunity to work with Professor Jennifer Rexford who literally opened another door for me into the fascinating research world.

Special thanks go to Melissa Lawson who put invaluable efforts in improving my communication skills. I would also like to thank departmental staffs for the convenience of various facilities.

The influence of Dr. Erich Nahum and Dr. John Tracey on the progress of my thesis cannot be over-estimated. I would like to thank them for introducing me to a new community and having me in the IBM T.J. Watson research center as a summer intern.

I benefited a lot from many professors' wonderful seminars, such as Professor Kai Li, Professor J.P. Singh, Professor Randy Wang, etc. just to name a few here.

My fellow officemates and dear friends at Princeton have made this place an enjoyable one to work, and have brought many cheers to my life in this country. I am not able to list their names here because the list is too long.

At the end of this prestigious procession of gratitude are the most precious people in my life. My parents sacrificed almost all their comforts in providing my previous education. I am forever indebted to their unconditional love. I am very grateful for my dear wife, Dr. Sheng (Sarah) Tan. Without her unwavering support and fighter spirits, I could not have gone so far.

Finally, I would like to acknowledge NSF grant No.CCR-0093351 which supported the projects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network servers continue to be a dominant form of information delivery for Internet users. Several factors will keep the demand for network-based information delivery increasing: the growing population of Internet users, improvements in end-user bandwidth such as broadband usage, and the increasing quantity of online information. Web services are still in their infancy, with new service models evolving rapidly, such as e-commerce services, online multimedia services, and online gaming.

Researchers within academia and industry have responded to this trend both by developing new server architectures and by developing optimizations for related systems. On one hand, several server models have been proposed and studied, from multi-process and multi-thread based servers, and event-driven servers, to a combination of event-driven and multi-thread. On the other hand, optimization approaches in operating systems have been taken to address performance issues for servers, such as zero-copy I/O and scalable event delivery.

As new services appear, server workloads in benchmarks also become more and more complex. For example, WebStone [55] has grown from relatively simple tools released by SGI to a family of benchmarks. Though WebStone's early popularity with hardware vendors has been largely supplanted by the committee-governed SPECweb [88] benchmarks, SPECweb has also evolved from the static-only SPECweb96 to the more complex SPECweb99, which models real-world web server access patterns and uses dynamic ad rotation services. The newly-released SPECweb2005 is even more intricate with emulation of requests from broadband Internet connections, and new workloads modeling a banking site, an e-commerce site, and a vendor support site.

With the increasing complexity of workloads, how to identify performance bottlenecks under these workloads is challenging, especially when the performance of the ”black box” OS needs to be determined. This thesis proposes an approach of making the OS transparent, by treating system call performance information as a *first-class* result, and returning it *in-band* immediately when the call returns [77]. This approach not only eliminates guesswork about what happens during call processing, but also gives the application control over how this information is collected, filtered, and analyzed. Thus it is customizable and fine-grained for analysis of various characteristics in complex workloads.

One trend in performance-related research in network servers is that much of the work focuses on improving throughput, leaving server response time less well understood. Recent optimization approaches mostly rely on scheduling, assuming that queuing delay is inherent to the system and is the cause of server latency. This thesis examines the root causes of server-induced latency and discovers that blocking in filesystem-related kernel queues is responsible for most excessive latency [78]. By addressing the blocking issues both in the application and in the kernel, we are able to improve response time by more than an order of magnitude under various workloads. Eliminating the blocking also reduces server burstiness and improves service fairness.

Network servers are usually multi-layer systems. Performance of network servers not only depends on the OS and the server software, but also the hardware platforms. Development in hardware platforms, especially processor architecture, always brings new research opportunities for server designers. For example, simultaneous multithreading (SMT) processors are particularly attractive to network servers because of their ability to hide memory latency by increasing the utilization of pipeline functional units. This thesis investigates server performance on the SMT-capable Intel Xeon processors [51]. By using processors with different clock rates and cache hierarchies, we are able to identify processor level bottlenecks in current SMT implementations and give insight into server optimization [79]. With the presence of other SMT processors such as the IBM POWER5 [40] and new architectures such as Chip Multi-Processor(CMP), we believe this is a promising avenue for future work.

1.1 The SPECweb99 Benchmark

For performance evaluation, understanding the type of workload is essential for interpretation of the performance achieved. Similarly, evaluation methods are critical for analysis, with some of them focusing on capacity comparison and others on bottleneck identification. This section presents an overview of the workloads used in this thesis.

Most of the workloads used in this thesis come from the SPECweb99 [88] benchmark, which is designed by the Standard Performance Evaluation Corporation (SPEC). The benchmark is the *de facto* standard in industry [58], with over 200 published results, and is different from most other Web server benchmarks in its complexity and requirements. The workload in this benchmark is modeled after the access patterns of multiple production Web sites.

SPECweb99 tests the overall scalability of Web servers under realistic conditions. It measures scalability by reporting the number of simultaneous connections the server can handle while meeting a specified quality of service. The data set and working set sizes increase with the number of simultaneous connections, and quickly exceed the physical memory of commodity systems. The data set size is calculated using the following formula:

$$Dataset(inMB) = 122 + (3.22 * \#ofSimultaneousConnections) \tag{1.1}$$

In SPECweb99 workloads, 70% of the requests are for static content, with the other 30% for dynamic content, including a mix of HTTP GET and POST requests. 0.15% of the requests require the use of a CGI process that must be spawned separately for each request.

Requests are generated from a set of files following a Zipf-like distribution. The file set size is controlled by the number of expected simultaneous connections, which is translated into the number of directories. Each directory is roughly 5 MB in total, and consists of four classes of files, and each class contains nine files. The details about the file weights and class weights are given in Table 1.1. The file sizes range from 100 bytes to 900 KB and are evenly sized within each class. Popularity of the four classes 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. The directories are chosen using a Zipf distribution with an alpha value of 1. In this model, the n^{th} most popular directory is given a weight of 1/n. The strong bias toward small files leads to the result that the most popular files consume very little aggregate space. Table 1.2 illustrates this heavy-tailed feature – the most popular 99% of the requests occupy at most 14% of the size of data set.

class size	100 – 900	1 – 9 KB	10 – 90 KB	100 – 900 KB
class weight	35%	50%	14%	1%

file size	1	2	3	4	5	6	7	8	9
file weight	3.9%	5.9%	8.8%	17.7%	35.3%	11.8%	7.1%	5.0%	4.4%

Table 1.1: Class and File characteristics in SPECweb99. Each class consists of 9 evenly-sized files, and the file's probability is the class weight times the weight within the class which follows Zipf's law.

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

Table 1.2: SPECWeb's popularity distributions. (Sizes do not scale linearly with data set size due to the Zipf-based popularity distribution of directories)

Beside of the standard SPECweb99 benchmark, we also use the static workloads of the benchmark for throughput and response time evaluation. Because SPECweb normally self-scales, increasing both data set size and number of simultaneous connections with the target throughput, this approach complicates comparisons between different servers. So when the measurement is merely throughput or response time, both values of data set size and number of connections are fixed.

1.2 Thesis Contribution

This thesis contributes network server performance analysis and improvement at three levels – the operating system level, server application design level, and processor architecture level.

- At the kernel level, we have developed a novel kernel profiling tool to provide applications with fine-grained in-channel kernel information at low cost. The tool has been shown to be effective in finding cross-boundary performance bottlenecks and inspired the design of other debugging systems [72].

- We have contributed a set of kernel patches to improve networked file transfer. The patch for `sendfile()` has been adopted by the FreeBSD official release, and has been used to reduce the overhead of other operating system activities [28].

- At application level, we have redesigned a popular research server, Flash, and the widely-used Apache server, improving Flash's SPECweb99 score by a factor of four and reducing response time by one to two orders of magnitude on both servers under various workloads.

- Using these servers, we study the root causes of server-induced latency, and trace the reason to blocking in filesystem-related kernel queues. The blocking is responsible for most of the server burstiness and service inversion. These results suggest that connection scheduling may not be the fundamental approach to reduce excessive latency.

- Finally, we investigate the architectural aspects of server performance, conducting a detailed analysis of delivered SMT systems. Our experiments use five different software packages and three hardware platforms, and perform a microarchitectural evaluation of performance using processor performance counters. We believe this study is more complete than any related work previously published.

- We find that current SMT implementation in Intel Xeons has a modest performance benefit for network servers, both in the uniprocessor and dual-processor environment. By comparing the measurements with earlier simulation results, we discuss the reasons of the discrepancy and the feasibility of these simulation models. Some of the results have yielded better understanding of processor bottlenecks and server optimization.

1.3 Dissertation Overview

This dissertation is organized as follow:

Chapter 2 presents a study of server performance and the interactions with operating systems. This chapter describes the design, implementation and evaluation of DeBox, an effective approach to provide more OS transparency, by exposing system call performance as a first-class result via in-band channels. We then use this tool for a case study on the Flash Web server to reveal various performance issues, and describe the solution for each problem found. Results are evaluated using the SPECweb99 benchmark and other workloads, and tested on different operating systems.

Chapter 3 examines latency issues further and traces the root cause of server-induced latency. By experimenting with workloads of various sizes, we discover a problem of head-of-line blocking within filesystem-related kernel queues, and demonstrates how this blocking causes server burstiness and gives rise to a phenomenon we call *service inversion*. We describe how to improve latency of both the Apache and the Flash server by more than an order of magnitude, with a qualitatively different change in the latency profiles, and propose an approach of quantifying service fairness.

Chapter 4 provides a performance analysis of simultaneous multithreading for server applications, using five software implementations and three hardware platforms. By exercising throughput comparison and microarchitectural analysis using performance counters, we find that the benefits of the current SMT implementation on Intel Xeon processors are modest for network servers, and short memory latency or extra L3 cache helps SMT yield better speedups. This chapter also discusses reasons of the differences between our results and similar studies performed using simulation.

Finally, Chapter 5 provides a summary of this dissertation.

Chapter 2

Server Performance and Interactions with Operating Systems

Server applications usually spend a significant fraction, sometimes more than 90%, of their time in the kernel. For this reason, operating system performance continues to be an active area of research, especially as demanding applications test OS scalability and performance limits. The kernel-user boundary becomes critically important as these applications achieve their work via system calls. As a result, examining the interaction between operating systems and user processes remains a useful area of investigation. This chapter discusses some of the negative interactions, describes the approaches used to explore these problematic issues, and presents solutions to them.

2.1 Introduction

In server design, previously developers could expect to put data-sharing services, such as NFS, into the kernel to avoid the limitations stemming from running in user space.

However, with the rapid rate of evolution in server workloads, using kernel integration to avoid performance problems becomes unrealistic. Especially for Web servers, the real deployment of in-kernel servers is still limited. The reason is because of dynamic content and security concerns.

Much of the earlier work focusing on the kernel-user interface centered around developing new system calls that are more closely tailored to the needs of particular applications. In particular, zero-copy I/O [26, 63] and scalable event delivery [9, 10, 46] are examples of techniques that have been adopted in mainstream operating systems, via calls such as `sendfile()`, `transmitfile()`, `kevent()`, and `epoll()`, to address performance issues for servers. Other approaches, such as allowing processes to declare their intentions to the OS [64], have also been proposed and implemented. Some system calls, such as `madvise()`, provide usage hints to the OS, but with operating systems free to ignore such requests or restrict them to mapped files, programs cannot rely on their behavior.

Some recent research uses the reverse approach, where applications determine how the "black box" OS is likely to behave and then adapt accordingly. For example, the Flash Web Server [62] uses the `mincore()` system call to determine memory residency of pages, and combines this information with some heuristics to avoid blocking. The "gray box" approach [7, 22] manages to infer memory residency by observing page faults and correlating them with known replacement algorithms. In both systems, memory-resident files are treated differently than others, improving performance, latency, or both. These approaches depend on the quality of the information they can obtain from the OS and the accuracy of their heuristics. As workload complexity increases, we believe that such inferences will become harder to make.

To remedy these problems, this thesis proposes a much more direct approach to making the OS transparent: make system call performance information a *first-class* result, and return it *in-band*. In practice, what this entails is having each system call fill a "performance result" structure, providing information about what occurred in processing the call. The term *first-class result* specifies that the performance information gets treated the same as other results, such as errno and the system call return value, instead of having to be explicitly requested via other system or library calls. The term *in-band* specifies that the information is returned to the caller immediately, instead of being logged or processed by some other monitoring processes. While the structure holding the performance feedback is much larger and more detailed than the `errno` global variable, they are conceptually similar. Simple monitoring at the system call boundary, the scheduler, page fault handlers, and function entry and exit is sufficient to provide detailed information about the inner working of the operating system. This approach not only eliminates guesswork about what happens during call processing, but also gives the application control over how this information is collected, filtered, and analyzed, providing more customizable and narrowly-targeted performance debugging than is available in existing tools.

To achieve the above approach, we have designed and implemented a performance analysis tool, called DeBox. DeBox allows users to determine where applications spend their time in the kernel, what causes the performance loss, what resources are under contention, and how the kernel behavior changes with the workload. The flexibility of DeBox allows us to measure very specific information, such as the kernel CPU consumption caused by a single call site in a program.

Experiments presented in this chapter focus on analyzing and optimizing the performance of the Flash Web Server on the industry-standard SPECweb99 benchmark [88]. Using DeBox, we diagnose and fix a series of problematic interactions between the server and the operating system on this benchmark. The resulting system shows an overall factor of four improvement in SPECweb99 score, throughput gains on other benchmarks, and latency reductions ranging from a factor of 4 to 47 on FreeBSD. Most of the issues are addressed by application redesign and the resulting system is portable, as demonstrated by showing improvements on Linux. The kernel modifications, optimizing the `sendfile()` system call, have been integrated into FreeBSD.

DeBox is specifically designed for performance analysis of the interactions between the OS and applications, especially in server-style environments with complex workloads. Its combination of features and flexibility is novel, and differentiates it from other profiling-related approaches. However, it is not designed to be a general-purpose profiler, since it currently does not address applications that spend most of their time in user space or in the “bottom half” (interrupt-driven) portion of the kernel. The design philosophy and implementation detail of DeBox are described in the following sections.

2.2 DeBox Design Philosophy

DeBox is designed to bridge the divide in performance analysis across the kernel and user boundary by exposing kernel performance behavior to user processes, with a focus on server-style applications with demanding workloads. In these environments, performance problems can occur on either side of the boundary, and limiting analysis to only one side potentially eliminates useful information. While some servers may spend most of their time in the kernel, the ultimate cause may be activities under the process’ control. As a result, applications may be able to modify their own behavior to avoid bottlenecks.

Some observations about performance analysis for server applications are discussed below. While some of these measurements could be made in other ways, we believe that

DeBox's approach is particularly well-suited for these environments. Note that replacing any of the existing tools is an explicit non-goal of DeBox, nor do we believe that such a goal is even feasible. Additionally, by making performance information first-class, DeBox provides opportunities not afforded by existing approaches. Some examples are provided below.

- **High overheads hide bottlenecks.** The cost of the debugging tools may artificially stress parts of the system, thus masking the real bottleneck at higher load levels. Problems that appear only at high request rates may not appear when a profiler causes an overall slowdown. Our tests show that for server workloads, kernel `gprof` has 40% performance degradation even when low resolution profiling is configured. Others tracing and event logging tools generate large quantities of data, up to 0.5MB/s in the Linux Trace Toolkit [97]. For more demanding workloads, the CPU or filesystem effects of these tools may be problematic.

DeBox is designed not only to exploit hardware performance counters to reduce overhead, but also to allow users to specify the level of detail to control the overall costs. Furthermore, by splitting the profiling policy and mechanism in DeBox, applications can decide how much effort to expend on collecting and storing information. Thus, they may selectively process the data, discard redundant or trivial information, and store only useful results to reduce the overhead. Not only does this approach make the cost of profiling controllable, but one process desiring profiling does not affect the behavior of others on the system. It affects only its own share of system resources.

- **User-level timing can be misleading.** Figure 2.1 shows user-level timing measurement of the `sendfile()` system call in an event-driven server. This server uses non-blocking sockets and invokes sendfile only for in-memory data. As a result, the high peaks on this graph are troubling, since they suggest the server is blocking. A similar measurement using `getrusage()` also falsely implies the same. Even though the measurement calls immediately precede and follow the system call, heavy system activity causes the scheduler to preempt the process in that small window.

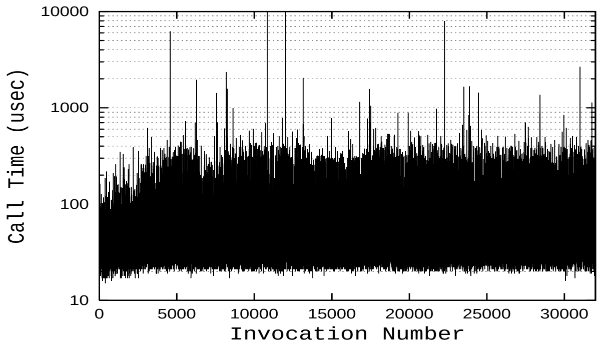


Figure 2.1: User-space timing of the `sendfile` call on a server running the SpecWeb99 benchmark – note the sharp peaks, which may indicate anomalous behavior in the kernel.

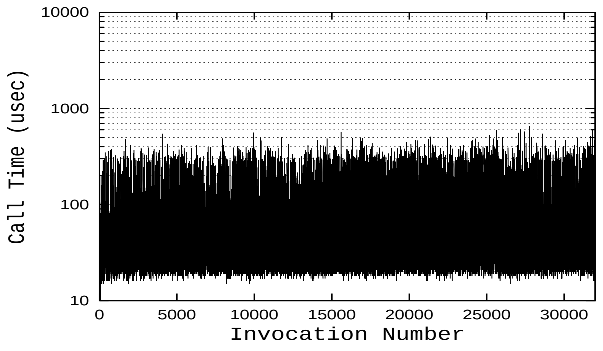


Figure 2.2: The same system call measured using DeBox shows much less variation in behavior.

In DeBox, measurement is integrated into the system call process, so it does not suffer from scheduler-induced measurement errors. The DeBox-derived measurements of the same call are shown in Figure 2.2, and do not indicate such sharp peaks and blocking. Summary data for `sendfile` and `accept` (in non-blocking mode) are shown in Table 2.1.

	`accept()`		`sendfile()`	
	User	DeBox	User	DeBox
Min	5.0	5.0	8.0	6.0
Median	10.0	6.0	60.0	53.0
Mean	14.8	10.5	86.6	77.5
Max	5216.0	174.0	12952.0	998.0

Table 2.1: Execution time (in μsec) of two system calls measured in user application and DeBox – Note the large difference in maximums stemming from the measurement technique.

- **Statistical methods miss infrequent events.** Profilers and monitoring tools may only sample events, with the belief that any event of interest is likely to take "enough" time to eventually be sampled. However, the correlation between frequency and importance may not always hold. Our experiments with the Flash web server indicate that adding a 1 ms delay to one out of every 1000 requests can degrade latency by a factor of 8 while showing little impact on throughput. This is precisely the kind of behavior that statistical profilers are likely to miss.

DeBox eliminates this gap by allowing applications to examine every system call. Applications can implement their own sampling policy, controlling overhead while still capturing the details of interest to them.

- **Data aggregation hides anomalies.** Whole-system profiling and logging tools may aggregate data to keep completeness and reduce overhead at the same time. This approach makes it hard to determine which call invocation experienced problems, or sometimes even which process or call site was responsible for high-overhead calls. This problem gets worse in network server environments where the systems are complex and large quantities of data are generated. It is not uncommon for these applications to have dozens of system call sites and thousands of invocations per second. For example, the Flash server consists of about 40 system calls and 150 calling sites. In these conditions, either discarding call history or full event logging is infeasible.

By making performance information a result of system calls, developers have control over how the kernel profiling is performed. Information can be recorded by process and by call site, instead of being aggregated by call number inside the kernel. Users may choose to save accumulated results, record per-call performance history over time, or fully store some of the anomalous call trace.

- **Out-of-band reporting misses useful opportunities.** As the kernel-user boundary becomes a significant issue for demanding applications, understanding the interaction between kernel and user processes becomes essential. Most existing tools provide measurements out-of-band, making online data processing harder and possibly missing useful opportunities. For example, the online method allows an application to `abort()` or record the status when a performance anomaly occurs, but it is impossible with out-of-band reporting.

When applications receive performance information tied to each system call via in-band channels, they can choose the filtering and aggregation appropriate for the program's context. They can easily correlate information about system calls with the underlying actions that invoke them.

2.3 DeBox Architecture & Implementation

This section describes the DeBox prototype implementation in FreeBSD and measures its overhead. First, we present the user-visible portion of DeBox, and then the kernel modifications. We measure overhead for DeBox support with both common system calls

```
typedef struct PerSleepInfo {
  int numSleeps;              /* # sleeps for the same reason */
  struct timeval blockedTime; /* how long the process is blocked */
  char wmesg[8];              /* reason for sleep (resource label) */
  char blockingFile[32];      /* file name causing the sleep */
  int blockingLine;           /* line number causing the sleep */
  int numWaitersEntry;        /* # of contenders at sleep */
  int numWaitersExit;         /* # of contenders at wake-up */
} PerSleepInfo;

typedef struct CallTrace {
  unsigned long callSite;     /* address of the caller */
  int deltaTime;              /* elapsed time in timer or CPU counter */
} CallTrace;

typedef struct DeBoxInfo {
  int syscallNum;             /* which system call */
  union CallTime {
    struct timeval callTimeval;
    long callCycles;          /* wall-clock time of entire call */
  } CallTime;
  int numPGFaults;            /* # page faults */
  int numPerSleepInfo;        /* # of filled PerSleepInfo elements */
  int traceDepth;             /* # functions called in this system call */
  struct PerSleepInfo psi[5]; /* sleeping info for this call */
  struct CallTrace ct[200];   /* call trace info for this call */
} DeBoxInfo;

int DeBoxControl(DeBoxInfo *resultBuf, int maxSleeps, int maxTrace);
```

Figure 2.3: DeBox data structures and function prototype

and real applications. Examples of how to fully use DeBox and what kinds of information it provides are deferred to the case study in Section 2.5.

2.3.1 User-Visible Portion

The programmer-visible interface of DeBox is intentionally simple, since it consists of some monitoring data structures and a new system call to enable and disable data gathering. Figure 2.3 shows DeBoxInfo, the data structure that handles the DeBox information. It serves as the "performance information" counterpart to other system call results like `errno`. Programs wishing to use DeBox need to perform two actions: declare one or more of these structures as global variables, and call DeBoxControl to specify how much per-call performance information it desires.

At first glance, the DeBoxInfo structure appears very large, which would normally be an issue since its size could affect system call performance. This structure size is not a significant concern, since the process specifies limits on how much of it is used. Most of the space is consumed by two arrays, PerSleepInfo and CallTrace. The PerSleepInfo array contains information about each of the times the system call blocks (sleeps) in the course of processing. The CallTrace array provides the history of what functions were called and how much time was spent in each. Both arrays are generously sized, and we do not expect many calls to fully utilize either one.

DeBoxControl can be called multiple times over the course of process execution for a variety of reasons. Programmers may wish to have several DeBoxInfo structures and use different structures for different purposes. They can also vary the number of PerSleepInfo and CallTrace items recorded for each call, to vary the level of detail generated. Finally, they can specify a NULL value for resultBuf, which deactivates DeBox monitoring for the process.

2.3.2 In-Kernel Implementation

The kernel support for DeBox consists of performing the necessary bookkeeping to gather the data in the DeBoxInfo structure. The points of interest are system call entry and exit, scheduler sleep and wakeup routines, and function entry and exit for all functions reachable from a system call.

Since DeBox returns performance information when each system call finishes, the system call entry and exit code is modified to detect if a process is using DeBox. Once a process calls DeBoxControl and specifies how much of the arrays to use, the kernel stores this information and allocates a kernel-space DeBoxInfo reachable from the process control block. This copy records information while the system call executes, avoiding many small copies between kernel and user. Prior to system call return, the requested information is copied back to user space.

At system call entry, all non-array fields of the process's DeBoxInfo are cleared. Arrays do not need to be explicitly cleared since the counters indicating their utilization have been cleared. Call number and start time are stored in the entry. Time is measured using the CPU cycle counter available on our hardware, but we could also use timer interrupts or other facilities provided by the hardware.

Page faults that occur during the system call are counted by modifying the page fault handler to check for DeBox activation. DeBox currently does not provide more detailed information on where faults occur, largely because we have not observed a real need for it. However, since the DeBoxInfo structure can contain other arrays, more detailed page fault information can be added if desired.

The most detailed accounting in DeBoxInfo revolves around the "sleeps", when the system call blocks waiting on some resource. When this occurs in FreeBSD, the system call invokes the `tsleep()` function, which passes control to the scheduler. When the resource becomes available, the `wakeup()` function is invoked and the affected processes are unblocked. Kernel routines invoking `tsleep()` provide a human-readable label. DeBox defines a new macro for `tsleep()` in the kernel header files that permits us to intercept any sleep points. When this occurs, DeBox records in a PerSleepInfo element where the sleep occurred (blockingFile and blockingLine), what time it started, what resource label was involved (wmesg), and the number of other processes waiting on the same resource (numWaitersEntry). Similarly, DeBox modifies the `wakeup()` routine to provide numWaitersExit and calculate how much time was spent blocked. If the system call sleeps more than once at the same location, that information is aggregated into a single PerSleepInfo entry.

```
DeBoxInfo:
        4, /* system call # */
  3591064, /* call time, microsecs */
      989, /* # of page faults */
        2, /* # of PerSleepInfo used */
        0, /* # of CallTrace used (disabled) */

PerSleepInfo[0]:
           1270  /* # occurrences */
         723903  /* time blocked, microsecs */
          biowr  /* resource label */
 kern/vfs_bio.c  /* file where blocked */
           2727  /* line where blocked */
              1  /* # processes on entry */
              0  /* # processes on exit */

PerSleepInfo[1]:
            325  /* # occurrences */
        2710256  /* time blocked, microsecs */
         spread  /* resource label */
miscfs/specfs/spec_vnops.c /* file where blocked */
            729  /* line where blocked */
              1  /* # processes on entry */
              0  /* # processes on exit */
```

Figure 2.4: Sample DeBox output showing the system call performance of copying a 10MB mapped file

The process of tracing which kernel functions are called during a system call is slightly more involved, largely to minimize overhead. Conceptually, all that has to occur is that every function entry and exit point has to record the current time and function name when it started and finished, similar to what call graph profilers use. The gcc compiler allows entry and exit functions to be specified via the “instrument functions” option, but these are invoked by explicit function calls. As a result, function call overhead increases by roughly a factor of three. Our current solution involves manually inserting entry and exit macros into reachable functions. The entry macro pushes current function address and time in a temporary stack. The exit macro pops out the function address, calculates the wall clock time, and records this information in the CallTrace array. Automating the modification should be possible in the future, such as what is done for kernel profiling using the `mcount()` function.

A sample of the output is given in Figure 2.4 to show information provided in DeBox. We memory-map a 10MB file, and use the `write()` system call to copy its contents to another file. The main DeBoxInfo structure shows that system call 4 (`write()`) was invoked, and it used about 3.6 seconds of wall-clock time. It incurred 989 page faults, and blocked in two unique places in the kernel. The first PerSleepInfo element shows that it blocked 1270 times at line 2727 in vfs_bio.c on "biowr", the block I/O write routine. The second location was line 729 of spec_vnops.c, which caused 325 blocks at "spread", read of a special file. The writes blocked for roughly 0.7 seconds, and the reads for 2.7 seconds.

2.3.3 Overhead

For DeBox to be attractive, it should generate low kernel overhead, especially in the common case. To quantify this overhead, we compare an unmodified kernel, a kernel with DeBox support, and the modified kernel with DeBox activated. These measurements are shown in Table 2.2. The first column indicates the various system calls – `getpid()`, `gettimeofday()`, and `pread()` with various sizes. The second column indicates the time required for these calls on an unmodified system. The remaining columns indicate the additional overhead for various DeBox features.

The measurement for call history tracing is separated since we do not expect it will be activated continuously. The "basic off" column indicates the overhead introduced with a modified kernel supporting DeBox without call tracing. The performance impact is virtually unnoticeable. The "basic on" column shows the impact of activating DeBox without call tracing. We use the CPU cycle counter, since accessing the hardware clock on our system requires 5 microseconds. This overhead is the reason why `gettimeofday()` has a comparable running time to a 512 byte read. These numbers show that the cost to

call name or read size	base time	DeBox without call trace off	DeBox without call trace on	DeBox call trace off	DeBox call trace on
getpid	0.46	+0.00	+0.50	+0.03	+1.45
gettimeofday	5.07	+0.00	+0.43	+0.03	+1.52
pread 128B	3.27	+0.02	+0.56	+0.21	+2.03
256 bytes	3.83	+0.00	+0.59	+0.26	+2.02
512 bytes	4.70	+0.00	+0.69	+0.28	+2.02
1024 bytes	6.74	+0.00	+0.68	+0.27	+2.02
2048 bytes	10.58	+0.03	+0.68	+0.26	+2.01
4096 bytes	18.43	+0.03	+0.74	+0.29	+2.16

Table 2.2: DeBox microbenchmark overheads – Base time uses an unmodified system. All times are in microseconds

support most DeBox features is minimal, and the cost of using the measurement infrastructure is tolerable. Since these costs are borne only by the applications that choose to enable DeBox, the overhead to the whole system is even lower. The performance impact with DeBox disabled, indicated by the 3rd column, is virtually unnoticeable. The cost of supporting call tracing, shown in the 5th column, where every function entry and exit point is affected, is higher, averaging approximately 5% of the system call time. This overhead is higher than ideal, and may not be desirable to have continuously enabled. However, our implementation is admittedly crude, and better compiler support could integrate it with the function prologue and epilogue code. We expect that we can reduce this overhead, along with the overhead of using the call tracing, with optimization.

Since microbenchmarks do not indicate what kinds of slowdowns may be typically observed, we provide some macrobenchmark results to give insight into these costs in Table 2.3. The three systems tested are: an unmodified system, one with only "basic" DeBox without call trace support, and one with complete DeBox support. The first two columns are times for archiving and compressing files of different sizes. The last column is for building the FreeBSD kernel. The overheads of DeBox support range from less than

1% to roughly 3% in the kernel build. We expect that many environments will tolerate this overhead in exchange for the flexibility provided by DeBox.

	tar-gz a directory with		make
	1MB file	10MB file	kernel
base time	275.61 ms	3078.50 ms	236.96 s
basic on	+0.97 ms	+22.73 ms	+1.74 s
full support	+1.03 ms	+44.58 ms	+7.49 s

Table 2.3: DeBox macrobenchmark overheads

2.4 Experimental Setup & Workloads

This section describes the experimental setup and the relevant software components of the system in this section. All of the experiments, except for the portability measurements are performed on a uniprocessor server running FreeBSD 4.6, with a 933MHz Pentium III, 1GB of memory, one 5400 RPM Maxtor IDE disk, and a single Netgear GA621 gigabit ethernet network adapter. The portability experiments are performed on another server box because the Linux kernel crashes on the existing server hardware. The clients consist of ten Pentium II machines running at 300 MHz connected to a switch using Fast Ethernet. All machines are configured to use the default (1500 byte) MTU as required by SpecWeb99.

The main application studied here is the event-driven Flash Web Server, although we also perform some tests on the widely-used multi-process Apache [6] server. The Flash Web Server consists of a main process and a number of helper processes. The main process multiplexes all client connections, is intended to be nonblocking, and is expected to serve all requests only from memory. The helpers load disk data and metadata into memory to allow the main process to avoid blocking on disk. The number of main processes in the system is generally equal to the number of physical processors, while the number of helper processes is dynamically adjusted based on load. In previous tests, the Flash Web Server has been shown to compare favorably to high-performance commercial Web servers [62]. We run with logging disabled to simplify comparison with Apache, where enabling logging degrades performance noticeably.

Our experiments focus on the standard SPECweb99 benchmark, with 30% dynamic content including a mix of HTTP GET and POST requests. A detailed description of the benchmark is provided in Section 1.1.

2.5 Using DeBox on the Flash Server

This section demonstrates how we use DeBox to analyze and optimize the behavior of the Flash Web Server. We discover a series of problematic interactions, trace their causes, and find appropriate solutions to avoid them or fix them. In the process, we gain insights into the causes of performance problems and how conventional solutions, such as throwing more resources at the problem, may exacerbate the problem. Our optimizations quadruple our SPECweb99 score and also sharply decrease latency.

2.5.1 Initial experiments

Our first run of SPECweb99 on the publicly available version of the Flash Web Server yields a SPECweb99 result of roughly 200 simultaneous connections, much lower than the published score of 575 achieved on comparable hardware using TUX, an in-kernel Linux-only HTTP server. At 200 simultaneous connections, the dataset size is roughly 770MB, which is smaller than the amount of physical memory in the machine. Not surprisingly, the workload is CPU-bound, and a quick examination shows that the `mincore()` system call is consuming more resources than any other call in Flash.

The underlying problem is the use of linked lists in the FreeBSD virtual memory subsystem for handling virtual memory objects. The heavy use of memory-mapped files in Flash generates large numbers of memory objects, and a linear walk utilized by `mincore()` generates significant overhead. We apply a patch from Alan Cox of Rice University that replaces the linked lists with splay trees, and this brings `mincore()` in line with other calls. Our SPECweb99 score rises to roughly 320, a 60% improvement. At this point, the working set has increased to 1.13GB, slightly exceeding our physical memory.

Once the `mincore()` problem is addressed, the system still appears to be CPU-bound. We suspect the data copying is the bottleneck. So we update the Flash server to use the zero-copy I/O system call, `sendfile()`. However, using `sendfile()` requires that file descriptors be kept open, greatly increasing the number of file descriptors in use by Flash. To mitigate this impact, we implement support for `sendfile()` concurrently with support for `kevent()`, which is a scalable event delivery mechanism recently incorporated into FreeBSD. After these changes, we are not surprised by the drop in CPU utilization, but are surprised that the SPECweb99 score drops to 300.

2.5.2 Successive refinement of detail

With the server exhibiting idle CPU time but an inability to meet SPECweb99's quality-of-service requirements, an obvious candidate is blocking system calls. However, Flash's main process is designed to avoid blocking. We tried tracing the problem using existing tools, but found they suffered from the problems discussed in section 2.2. These experiences motivated the creation of DeBox.

The DeBox structures provide various levels of detail, allowing applications to specify what to measure. A typical use would first collect the basic DeBoxInfo to observe anomalies, then enable more details to identify the affected system calls, invocations, and finally the whole call trace.

We first use DeBox to get the blocking information, which is stored in the PerSleepInfo field. The PerSleepInfo data shows seven different system calls blocking in the kernel, and examination of the resource labels (wmesg) shows four reasons for blocking. These results are shown in Table 2.4, where each column header shows the resource label causing the blocking, followed by the total number of times blocked at that label. The elements in the column are the system calls that block on that resource, and the number of invocations involved. As evidenced by the calls involved, the "biord" (block I/O read) and "inode" (vnode lock) labels are both involved in opening and retrieving files, which is not surprising since our data set exceeds the physical memory of the machine.

biord/166	inode/127	getblk/1	sfpbsy/1
open/162	*readlink*/84	*close*/1	*sendfile*/1
read/3	*open*/28		
unlink/1	*read*/9		
	stat/6		

Table 2.4: Summarized DeBox output showing blocking counts – The layout is organized by resource label and system call name. For example, of the 127 times this test blocked with the "inode" label, 28 were from the `open()` system call

The finest-grained kernel information is provided in the CallTrace structure, and we enable this level of detail once the PerSleepInfo identifies possible candidates. The main process should only be accessing cached data, so the fact that it blocks on disk-related calls is puzzling. For portability, the main process in Flash uses the helpers to demand-fetch disk data and metadata into the OS caches, and repeats the operation immediately after the helpers have completed loading, assuming that the recently loaded information will prevent it from blocking. Observing the full CallTrace of some of these blocking invocations shows the blocking is not caused by disk access, but contention on filesystem locks. Combining the blocking information from helper processes reveals that when the main process blocks, the helpers are operating on similarly-named files as the main process. We solve this problem by having the helpers return open file descriptors using `sendmsg()`, eliminating duplication of work in the main process. With this change, we are able to handle 370 simultaneous connections from SPECweb99, with a dataset size of 1.28GB.

2.5.3 Capturing rare anomaly paths

We find that the `sendmsg()` change solves most of the filesystem-related blocking. However, one `open()` call in Flash still shows occasional blocking at the label "biord" (reading a disk block), but only after the server has been running for some time and under heavy workloads. Only revealing which call induced the problem may not suffice a complete picture, because the reason of invoking that call is unclear. In a system with multiple identical system calls, existing tools do not have an efficient way to find which one causes the problem and the calling path involved.

Because DeBox information is returned in-band, the user-space context is also available when kernel performance anomalies are detected. On finding a blocking invocation of `open()`, we capture the path through the user code by calling `abort()` and using `gdb` to dump the stack[1]. This approach uncovers a subtle performance bug in Flash induced by mapped-file cache replacement. Flash has two independent caches – one for URL-to-filename translations (name cache), and another for memory-mapped regions

[1]Alternately, we could invoke `fork()` followed by `abort()` to keep the process running while still obtaining a snapshot, or we could record the call path manually.

(data cache). For this workload, the name cache does not suffer from capacity misses, while the data cache may evict the least recently used entries. Under heavy load, a name cache hit and a data cache capacity miss causes Flash to erroneously believe that it had just recently performed the name translation and has the metadata cached. When Flash calls `open()` to access the file in this circumstance, the metadata associated with the name conversion is missing, causing blocking. We solve this problem by allowing the second set of helpers, the read helpers, to return file descriptors if the main process does not already have them open. After fixing this bug, we are able to handle 390 simultaneous connections, and a 1.34GB dataset.

2.5.4 Tracking call histories

With all blocking eliminated and with a much higher request rate, we return to the issue of CPU consumption. By storing the CallTime field of each system call, we can track per-call performance by invocation, both to observe trends and to identify time-related problems. Traditional profiling tools usually report average CPU consumption of each function, thus hiding any performance trends. User-space timing functions may catch the general trend in spite of the measurement error, but involve much more work to track each system call and find the problematic ones.

Process creation overhead

By recording all CPU time values, we find that the largest call times are for the `fork()` system call and that its cost grows with the number of invocations, approaching 130 msec. Figure 2.5 shows the per-call time as a function of invocation. We observe that `fork()` time increases as the program runs, starting as low as 0.3 msec. These calls stem from the SPECweb99 workload's requirement that 0.15% of the requests be handled by forking new processes.

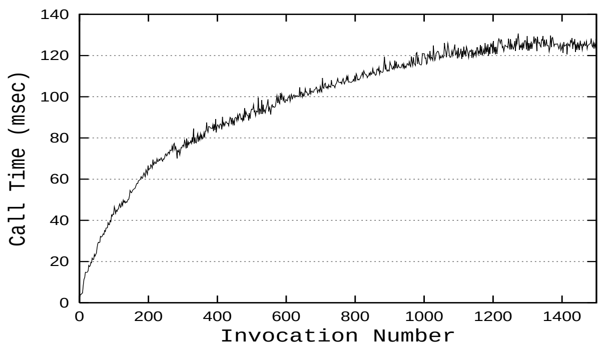


Figure 2.5: Call time of `fork()` as a function of invocation

A full call trace indicates that `fork()` spends the bulk of its time copying file descriptors and VM map entries (for mapped regions). We confirm this observation by varying the sizes of the caches in Flash and seeing their impact on `fork()` times. Rather than changing the implementation of `fork()`, we opt to slightly modify the Flash architecture. We introduce a new helper process that is responsible for creating the CGI processes. Since this new process does not map files or cache open files, its `fork()` time is not affected by the main process size. This change yields a 10% improvement, to 440 simultaneous connections and a 1.50GB dataset size.

Memory lookup overhead

Though the dataset size now exceeds physical memory by over 50%, the system bottleneck remains CPU. Examining the time consumption of each system call again reveals that most time is being spent in memory residency checking. Though our modified Flash uses `sendfile()`, it uses `mincore()` to determine memory residency, which requires that files be memory-mapped. The cumulative overhead of memory-map operations is the largest consumer of CPU time. As can be seen in Figure 2.6, the per-call overhead of `mmap()` is significant and increases as the server runs. The cost increase is presumably due to finding available space as the process memory map becomes fragmented.

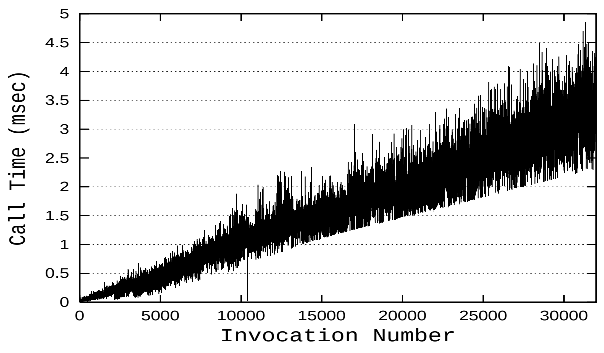


Figure 2.6: Call time of `mmap()` as a function of invocation

To avoid the memory-residency overheads, we use Flash's mapped-file cache bookkeeping as the sole heuristic for guessing memory residency. We eliminate all `mmap, mincore, and munmap` calls but keep track of what pieces of files have been recently accessed. Sizing the cache conservatively with respect to main memory, we save CPU overhead but introduce a small risk of having the main process block. The CPU savings of this change is substantial, allowing us to reach 620 connections (2GB dataset).

2.5.5 Profiling by call site

We take advantage of DeBox's flexibility by separating the kernel time consumption based on call site rather than call name. We are interested in the cost of handling dynamic content since SPECweb99 includes 30% dynamic requests which could be processed by various interfaces. Flash uses a persistent CGI interface similar to FastCGI [60] to reuse CGI processes when possible, and this mechanism communicates over pipes. Although the `read()` and `write()` system calls are used by the main process, the helpers, and all of the CGI processes, we measure the overhead of only those involved in communication with CGI processes.

Our measurements show that the single call site responsible for most of the time is where the main process reads from the CGIs, consuming 20% of all kernel time, (176 seconds out of 891 seconds total). Writing the request to the CGI processes is much smaller, requiring only 24.3 seconds of system call time. This level of detail demonstrates the power of making performance a first-class result, since existing kernel profilers would not have been able to separate the time for the `read()` calls by call sites. By modifying our CGI interface slightly, the main process writes only the HTTP header to the client, and passes the socket to the CGI application to let it write the data directly. This change allows us to reach 710 connections (2.35GB dataset).

2.5.6 Other optimization opportunities

By replacing our exact memory residency check with a cheaper heuristic, we gain performance, but introduce blocking into the `sendfile()` system call. New PerSleepInfo measurements of the blocking behavior of `sendfile()` are shown in Table 2.5.

time	label	kernel file	line
6492	sfbufa	kern/uipc_syscalls.c	1459
702	getblk	kern/kern_lock.c	182
984544	biord	kern/vfs_bio.c	2724

Table 2.5: New blocking measurements of `sendfile()`

The resource label “sfbufa” indicates that the kernel has exhausted the sendfile buffer used to map filesystem pages into kernel virtual memory. We confirm that increasing the number of buffer elements during boot time may mitigate this problem in our test. However, based on the results of previous copy-avoidance systems [26, 63], we opt instead to implement recycling of kernel virtual address buffers. We use a hash table to maintain the buffer elements with each element indexed by the physical address of the page to be sent, and use a least-recently-used (LRU) list to store inactive elements. For each buffer element, we introduce a reference count to track its liveness. When the kernel initially boots, both the hash table and the buffer are initialized and all buffer elements are put in the LRU queue. The typical operations of this recycling process are as follow:

When a file page is scheduled to send, its address is used to compute the hash table entry. If the corresponding buffer element is already put in the hash table, the element is removed from the LRU list and we increment the reference count. Otherwise, we pull the head element in the LRU list, map the page to be sent, and put the element in the hash table. After the page address is successfully transfered to the lower level of the networking layer, we decrease the reference count instead of freeing the mapping immediately. When the reference count reaches zero, the mapping is freed and the sendfile buffer is put into the tail of the LRU list for reuse.

With this change, many requests to the same file do not cause multiple mappings, and eliminates the associated virtual memory and physical map (pmap) operations. Caching these mappings may temporarily use more wired memory than no caching, but the reduction in overhead and address space consumption outweighs the drawbacks.

The other two resource labels, “getblk” and “biord”, are related to disk access initiated within `sendfile()` when the requested pages are not in memory. Even though the socket being used is nonblocking, that behavior is limited only to network buffer usage. We introduce a new flag to `sendfile()` so that it returns a different `errno` value if disk blocking would occur. This change allows us to achieve the same effect as we had with `mincore()`, but with much less CPU overhead. We may optionally have the read helper process send data directly back to the client on a filesystem cache miss, but have not implemented this optimization.

However, even with blocking eliminated, we find performance barely changes when using `sendfile()` versus `writev()`, and we find that the problem stems from handling small writes. HTTP responses consist of a small header followed by file data. The `writev()` code aggregates the header and the first portion of the body data into one packet, benefiting small file transfers. In SPECweb99, 35% of all static requests are for files 1KB or smaller.

The FreeBSD `sendfile()` call includes parameters specifying headers and trailers to be sent with the data, whereas the Linux implementation does not. Linux introduces a new socket option, TCP_CORK, to delay transmission until full packets can be assembled. While FreeBSD's "monolithic" approach provides enough information to avoid sending a separate header, its implementation uses a kernel version of `writev()` for the header, thus generating an extra packet. We improve this implementation by creating an mbuf chain using the header and body data before sending it to lower levels of the network stack. This change generates fewer packets, improving performance and network latency. Results of these changes on a microbenchmark are shown in Figure 2.7. With the `sendfile()` changes, we are able to achieve a SPECweb99 score of 820, with a dataset size of 2.7GB.

2.5.7 Case Study Summary

By addressing the interaction areas identified by DeBox, we achieve a factor of four improvement in our SPECweb99 score, supporting four times as many simultaneous connections while also handling a data set that almost three times as large as the physical memory of our machine. The SPECweb99 results of our modifications can be seen in

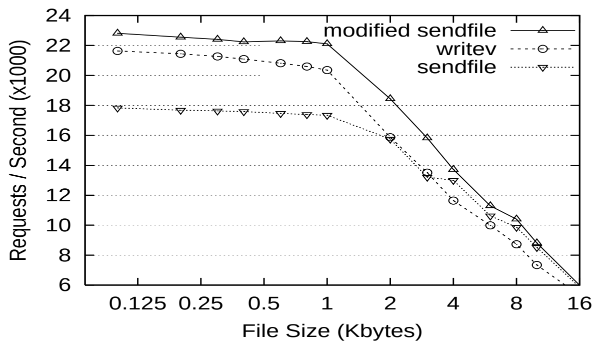


Figure 2.7: Microbenchmark performance comparison of writev, sendfile, and modified sendfile – In this test, all clients request a single file at full speed using persistent connections.

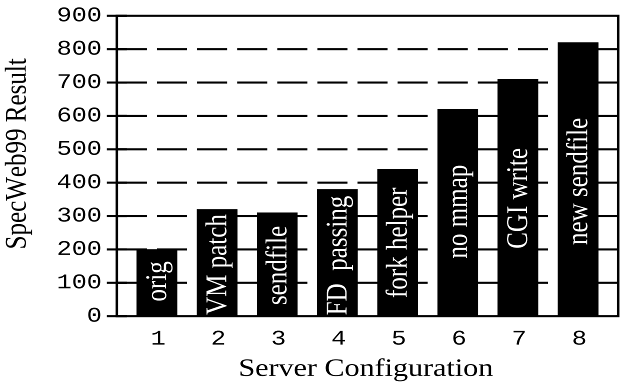


Figure 2.8: SPECweb99 summary – 1. Original 2. VM patch 3. Using sendfile() 4. FD-passing helpers 5. Fork helper 6. Eliminate mmap 7. New CGI interface 8. New sendfile()

Figure 2.8, where we show the scores for all of the intermediate modifications we made. Our final result of 820 compares favorably to published SPECweb99 scores, though no directly comparable systems have been benchmarked. We outperform all uniprocessor systems with similar memory configurations but using other server software – the highest score for a system with less than 2GB of memory is 575.

Most of our changes are portable architectural modifications to the Flash Web Server, including (1) passing file descriptors between the helpers and the main process to avoid most disk operations in the main process, (2) introducing a new `fork()` helper to handle forking CGI requests, (3) eliminating the mapped file cache, and (4) allowing CGI processes to write directly to the clients instead of writing to the main process. Figure 2.9 shows the original and new architectures of the static content path for the server.

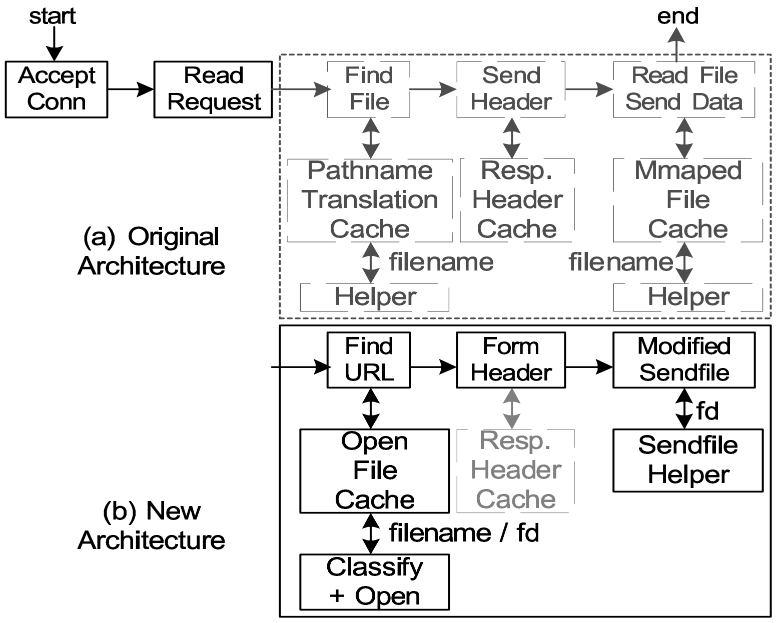


Figure 2.9: Architectural changes – The architecture is greatly simplified by using file descriptor passing and eliminating mapped file caching. Modified components are indicated with dark boxes.

The changes we make to the operating system focus on `sendfile()`, including (1) adding a new flag and return value to indicate when blocking on disk would occur, (2) caching kernel address space mapping to avoid unnecessary physical map operations, and (3) sending headers and file data in a single mbuf chain to avoid multiple packets for small responses. Additionally, we apply a virtual memory system patch that ultimately is superfluous since we remove the memory-mapped file cache. We have provided our modifications to the FreeBSD developer group and all three optimizations have been incorporated into FreeBSD.

2.6 Latency

Since we identify and correct many sources of blocking, we are interested in the effects of our changes on server latency. We first compare the effect of our changes on the SPECweb99 workload, and then reproduce workloads used by other researchers in studying static content latencies. In all cases, we compare latencies using a workload below the maximum of the slowest server configuration under test.

2.6.1 On SPECweb99 workloads

On the SPECweb99 workload, we find that mean response time is reduced by a factor of four by our changes. The cumulative distribution of latencies can be seen in Figure 2.10. We use 300 simultaneous connections, and compare the new server with the original Flash running on a patched VM system. Since 30% of the requests are for longer-running dynamic content, we also test the latencies of a SPECweb99 test with only static requests. The mean of this workload is 7.1 msec, lower than the 10.6 msec mean for the new server running the complete workload. This difference suggests that further optimization of dynamic content handling may lead to even better performance. To compare the difference between static and dynamic request handling, we calculate the 5^{th}, 50^{th}, and 95^{th} percentiles of the latencies for requests on the SPECweb99 workload. These results are shown in Table 2.6, and indicate that dynamic content is served at roughly half the speed of its static counterpart. The latency difference between the new server and the original Flash on this test is not as large as expected because the working set still fits in physical memory.

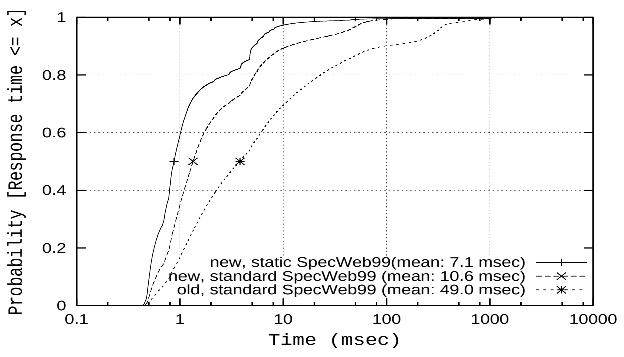


Figure 2.10: Latency summary for 300 SPECweb99 connections

	5%(ms)	50%(ms)	95%(ms)	mean(ms)
static	0.51	1.45	59.81	9.92
dynamic	0.99	2.83	91.31	12.19

Table 2.6: Separating SPECweb99 static and dynamic latencies

2.6.2 On Disk-bound static workload

To determine our latency benefit on a more disk-bound workload and to compare our results with those of other researchers, we construct a static workload similar to the one used to evaluate the Haboob server [96]. In this workload, 1020 simulated clients generate static requests to a 3.3GB data set. Persistent connections are used, with clients issuing 5 requests per connection before closing it. To avoid overload, the request rate is fixed at 2300 requests/second, which is roughly 90% of the slowest server's capacity.

We compare several configurations to determine the latency benefits and the impact of parallelism in the server. We run the new and original versions of Flash with a single instance and four instances, to compare uniprocessor configurations with what would be expected on a 4-way SMP. We also run Apache with 150 and 300 server processes.

The results, given in Figure 2.11 and Table 2.7, show the response time of our new server under this workload exhibits improvements of more than a factor of twelve in mean response time, and a factor of 47 in median latency. With four instances, the differences

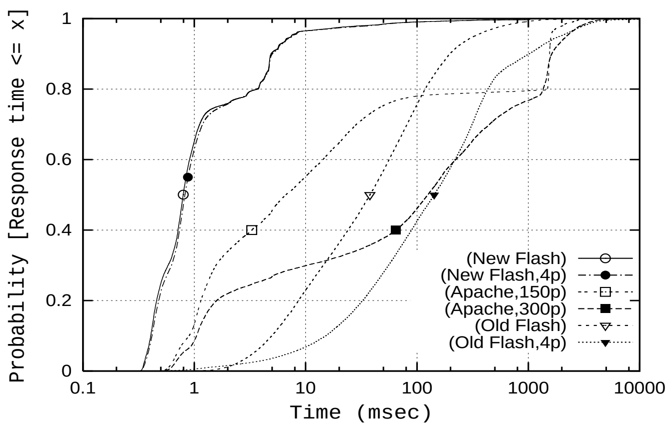


Figure 2.11: Response latencies for the 3.3GB static workload

	5% (ms)	median (ms)	95% (ms)	mean (ms)
New Flash	0.37	0.79	7.45	7.56
New Flash, 4p	0.38	0.82	7.51	7.72
Old Flash	3.36	37.59	326.40	92.37
Old Flash, 4p	7.05	142.65	1924.42	420.85
Apache 150p	0.70	6.64	1599.50	360.62
Apache 300p	0.78	124.98	2201.63	545.93

Table 2.7: Summaries of the static workload latencies

are a factor of 54 in mean response time and 174 in median time. We measure the maximum capacities of the servers when run in infinite-demand mode, and these results are shown in Table 2.8. While the throughput gain from our optimizations is significant, the scale of gain is much lower than the SPECweb99 test, indicating that our latency benefits do not stem purely from extra capacity.

data set	Apache	Old Flash	New Flash
500MB	240.3	485.2	660.9
1.5GB	230.7	410.6	580.3
3.3GB	210.6	264.5	326.4

Table 2.8: Server static workload capacities (Mb/s)

2.6.3 Excess parallelism

We also observe that all servers tested show latency degradation when running with more processes, though the effect is much lower for our new server. This observation is in line with the self-interference between the helpers and the main Flash process which we described earlier. We increase the number of helper processes and measure its effect on the SPECweb99 results, as shown in Table 2.9. We observe that too few helpers is insufficient to fully utilize the disk, and increasing their number initially helps performance. However, the blocking from self-interference increases, eventually decreasing performance. A similar phenomenon, stemming from the same problem, is also observed with Apache. Using DeBox, we find that Apache with 150 processes, sleeps 3667 times per second, increasing to 3994 times per second at 300 processes. This behavior is responsible for Apache's latency increase in Figure 2.11.

# of helpers	1	5	10	15
Blocking count	114	295	339	394
% Conforming	40.9%	95.1%	96.9%	89.5%

Table 2.9: Parallelism benefits and self-interference – The conformance measurement indicates how many requests meet SPECweb99's quality-of-service requirement.

This result suggests that excess parallelism, where server designers use parallelism for convenience, may actually degrade performance noticeably. This observation may explain the latency behavior reported for Haboob [96].

2.7 Results Portability

The main goal of this work is to provide developers with tools to diagnose and correct the performance problems in their own applications. Thus, we hope that the optimizations made on one platform also have benefit on other platforms. To test this premise, we test the performance of the new servers on Linux, which has no DeBox support.

Unfortunately, we were unable to get Linux to run properly on our existing hardware, despite several attempts to resolve the issue on the Linux kernel list. So, for these numbers, we use a server machine with a 3.0 GHz Pentium 4 processor and two Intel Pro1000/MT Gigabit adapters, 1GB of memory, and a similar disk. The experiments were performed on 2.4.21 kernel with `epoll()` support.

We compare the throughput and latency of four servers: Apache 1.3.27, Haboob, Flash, and the new Flash. We increase the max number of clients to 1024 in Apache and disable logging. Both the original Flash and the new Flash server use the maximum available cache size for LRU. We also adjust the cache size in Haboob for the best performance. The throughput results, shown in Table 2.10, are quite surprising. The Haboob server, despite having aggressive optimizations and event-driven stages, performs slightly better than Apache on disk-bound workload but worse than Apache on an in-memory workload. We believe that its dependence on excess parallelism (via its threaded design) may have some impact on its performance. The new Flash server gains about 17-24% over the old one for the smaller workloads, and all four servers have similar throughput on the larger workload because of the disk bottleneck.

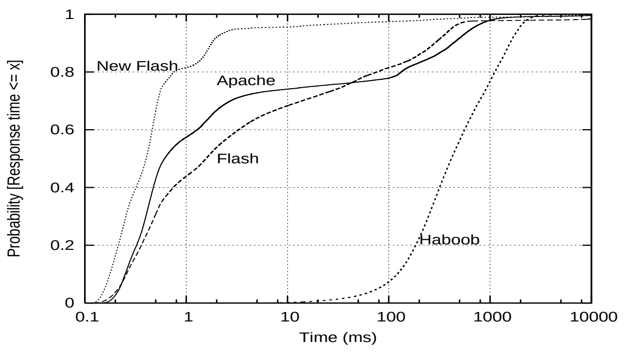


Figure 2.12: Response time on Linux with 3.3GB dataset

Throughput (Mb/s)				
data set	Haboob	Apache	Flash	New Flash
500MB	324.9	434.3	1098.1	1284.7
1.5GB	303.4	372.4	661.7	822.5
3.3GB	184.1	177.4	173.8	199.1

Response Time (ms)				
profile	Haboob	Apache	Flash	New Flash
5%	78.2	0.22	0.21	0.15
median	414.3	0.61	1.56	0.42
95%	1918.9	661.8	412.5	3.68
mean	656.2	418.0	512.5	141.9

Table 2.10: Throughput measurement on Linux with 1GB memory

Despite similar throughputs at the 3.3GB data set size, the latencies of the servers, shown in Figure 2.12 and Table 2.10, are markedly different. The Haboob latency profile is very close to the published result [96], but is worse by all of the other servers. We surmise that the minimal amount of tuning done to configurations of Apache and the original Flash yield much better results than the original Haboob comparison. The benefit of our optimization is still valid on this platform, with a factor of 4 both in median and mean latency over the original Flash. One interesting observation is that the 95% latency of the new Flash is a factor of 39 lower than the mean value. This result suggests that the small fraction of long-latency requests is the major contributor to the mean latency. Though our Linux results are not directly comparable to our FreeBSD ones due to the hardware differences, we do notice this phenomenon is less obvious on FreeBSD. Presumably one of the causes of this is the blocking disk I/O feature of `sendfile()` on Linux. Another reason may be Linux's filesystem performance, since this throughput is worse than what we observed on FreeBSD.

2.8 Related Work

To compare DeBox's approach of making performance information a first-class result, we describe three categories of tools currently in use, and explain how DeBox relates to these approaches.

- **Function-based profilers –** Programs such as `prof`, `gprof` [29], and their variants are often used to detect hot-spots in programs and kernels. These tools use compiler assistance to add bookkeeping information (count and time) to the program. Data is gathered while running and analyzed offline to reveal function call counts and CPU usage, often along edges in the call graph. This approach often suffers from high overhead, especially when function call times are small.
- **Coverage-based profilers –** These profilers divide the program of interest into regions and use a clock interrupt to periodically sample the location of the program counter. Like function-based profilers, data gathering is done online and analyzed offline. Tools such as `profil()`, `kernbb`, and `tcov` can then use this information to show what parts of the program are most likely to consume CPU time. Coverage-only approaches may miss infrequently-called functions entirely and may not be able to show call graph behavior. Coverage information combined with compiler analysis can be used to show usage on a basic-block basis.
- **Hardware-assisted profilers –** These profilers are similar to coverage-based profilers, but use special features of the microprocessor (event counters, timers, programmable interrupts) to obtain high-precision information at lower cost. The other major difference is that these profilers, such as DCPI [4], Morph [98], VTune [34], Oprofile [61], and PP[3], tend to be whole system profilers, capturing activity across all processes and the operating system.

In this category, DeBox is logically closest to kernel `gprof`, though it provides more than just timing information. DeBox's full call trace allows more complete call graph generation than gprof's arc counts, and with the data compression and storage performed in user space, overhead is moved from the kernel to the process, so processes have control over the profiling cost. Compared to path profiling, DeBox allows developers to customize the level of detail they want about specific paths, and to act on that information as it is generated. In comparison to low-level statistical profilers such as DCPI, coverage differs since DeBox measures functions directly used in the system call. As a result, the difference in approach yields some differences in what can be gathered and the difficulty in doing so – DCPI can gather bottom-half information, which DeBox currently cannot. However, DeBox can easily isolate problematic paths and their call sites, which DCPI's aggregation makes more difficult.

- **System activity monitors** – Tools such as `top`, `vmstat`, `netstat`, `iostat`, and `systat` can be used to monitor a running system or determine a first-order cause for system slowdowns. The level of precision varies greatly, with `top` showing per-process information on CPU usage, memory consumption, ownership, and running time, to `vmstat` showing only summary information on memory usage, fault rates, disk activity, and CPU usage.
- **Trace tools** – Trace tools provide a means of observing the system call behavior of processes without access to source code. Tools such as `truss`, PCT [16], `strace` [2], and `ktrace` are able to show some details of system calls, such as parameters, return values, and timing information. Recent tools, such as Kitrace [43] and the Linux Trace Toolkit [97], also provide data on some kernel state that changes as a result of the system calls. These tools are intended for observing another process, and as a result, produc-

ing out-of-band measurements and data aggregation, often requiring post-processing to generate usable output.

• **Timing calls** – Using `gettimeofday()` or similar calls, programmers can manually record the start and end times of events to infer information based on the difference. The `getrusage()` call adds some information beyond timings (context switches, faults, messages and I/O counts) and can similarly used. If per-call information is required, not only do these approaches introduce many more system calls, but the information can be misleading.

DeBox compares favorably with a hypothetical merger of the timing calls and the trace tools in the sense that timing information is presented in-band, but so is the other information. In comparison with the Linux Trace Toolkit, our focus differs in that we gather the most significant pieces of data related to performance, and we capture it at a much higher level of detail.

• **Microbenchmarks** – Tools such as lmbench [53] and hbench:OS [18] can measure best-case times or the isolated cost of certain operations (cache misses, context switches, etc.). Common usage for these tools is to compare different operating systems, different hardware platforms, or possible optimizations.

• **Latency tools** – Recent work on attempting to find the source of latency on desktop systems not designed for real-time work have yielded insight and some tools. The Intel Real-Time Performance Analyzer [66] helps automate the process of pinpointing latency. The work of Cota-Robles and Held [24] and Jones and Regehr [38] demonstrate the benefits of successive measurement and searching.

• **Instrumentation** – Dynamic instrumentation tools provide mechanisms to instrument running systems (processes or the kernel) under user control, and to obtain precise kernel information. Examples include DynInst [19], KernInst [89], ParaDyn [54], Etch [74], and ATOM [86]. The appeal of this approach versus standard profilers is the flexibility (arbitrary code can be inserted) and the cost (no overhead until use). Information is presented out-of-band.

Since DeBox measures the performance of calls in their natural usage, it resembles the instrumentation tools. DeBox gains some flexibility by presenting this data to the application, which can filter it on-line. One major difference between DeBox and kernel instrumentation is that we provide a rich set of measurements to any process, rather than providing information only to privileged processes.

Beyond these performance analysis tools, the idea of observing kernel behavior to improve performance has appeared in many different forms. We share similarities with Scheduler Activations [5] in observing scheduler activity to optimize application performance, and with the Infokernel system by Arpaci-Dusseau *et al*. [8]. Our goals differ, since we are more concerned with understanding why blocking occurs rather than reacting to it during a system call. Our non-blocking `sendfile()` modification is patterned on non-blocking sockets, but it could be used in other system calls as well. In a similar vein, RedHat has applied for a patent on a new flag to the `open()` call, which aborts if the necessary metadata is not in memory [57].

Our observations on blocking and its impact on latency may impact server design. Event-driven designs for network servers have been a popular approach since the performance studies of the Harvest Cache [17] and the Flash server [62]. Schmidt and Hu [80] performed much of the early work in studying threaded architectures for improving server performance. A hybrid architecture was used by Welsh *et al*. [96] to support scheduling, while Larus and Parkes [45] demonstrate that such scheduling can also be performed in event-driven architectures. Qie et al. [67] show that such architectures can also be protected against denial-of-service attacks. Adya et al. [1] discuss the unification of the two models. We believe that DeBox can be used to identify problem areas in other servers and architectures as well.

2.9 Discussion

We have shown how DeBox can be used in a variety of examples, allowing developers to shape profiling policy and react to anomalies in ways that are not possible with other tools. Although DeBox does require access to kernel source code for achieving the highest impact, we do not believe that such a restriction is significant. FreeBSD, NetBSD, and Linux sources are easily available, and with the advent of Microsoft's Shared Source initiatives, few hardware platforms exist for which some OS source is not available. Also, general information about kernel behavior instead of source code may be enough to help application redesign. Our performance portability results also demonstrate that our new system achieves better performance even without kernel modification. A further implication of this is that it is possible to perform analysis and modifications while running on one operating system, and still achieve some degree of benefit in other environments.

Part of the thesis focuses on how DeBox can be used as a performance analysis tool, but we have not implemented its utilization in general-purpose monitoring. Given its low overheads, DeBox is an excellent candidate for monitoring long-running applications. This approach can be reached by modifying the `libc` library and associated header files so that a simple recompile and relink will enable monitoring of applications using DeBox. It is also possible to process results automatically by allowing user-specified analysis policies.

While we have shown DeBox to be effective in identifying performance problems in the interaction between the OS and applications, the current version of DeBox does not handle the bottom-half activities in the kernel. DeBox's current focus on the system call boundary also makes it less useful for tracing problems arising purely in user space. However, we believe that the promise of the DeBox approach can be adapted to other areas such as multiprocessor OS support, preemptive kernels, and analysis of the top-half/bottom-half boundary within the operating system.

Chapter 3

Server Response Time Under Heavy Load

In the previous chapter we evaluated server latency using the standard SPECweb99 workload and at a fixed request rate with a disk-bound static workload. The significant amount of improvement in the new server prompts us to identify the root causes of server latency. This chapter investigates the impact of head-of-line blocking on server-induced latency, discusses how it can be observed in server burstiness and response time under load, and finally presents a way of quantifying these effects.

3.1 Introduction

Much of the performance-related research in network servers has focused on improving throughput, with less attention paid to latency [33, 62]. 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 SPECWeb [88] and WebStone [56], 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. Some recent work addresses the issue of measuring end-user latency [13, 70], with optimization approaches mostly focusing on scheduling [31, 45, 95, 96].

However, little is understood about the trends in network server latencies, or how the 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. 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 enable other research, such as improving Quality-of-Service (QoS) or scheduling policies.

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

The 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.

3.2 Background

In this section we provide some background on network servers, experimental setup, workloads, and methodology since we begin our analysis with experimental measurements of the servers. Performance debugging which we discussed in the previous chapter [77] examined blocking in servers, but did not specifically try to understand the origins of latency, the main topic of this chapter.

3.2.1 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 [6], version 1.3.27. To test high-performance servers, we use the event-driven Flash Web Server [62], 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 the OS's 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.

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 by recompiling with an increased HARD_SERVER_LIMIT. Since Apache's logging causes a noticeable performance loss, we disable access logging in both servers.

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

Table 3.1: Server hardware information

3.2.2 Experimental Setup & Workloads

We use a LAN to expose the server-induced latencies. 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, 16K 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 3.1, with measured values provided by lmbench [53].

In order to use a widely-understood workload while still maintaining tractability in the analysis, we focus on the static content workload of the SPECweb99 benchmark. Details of this workload is described in Section 1.1. To facilitate comparisons with previous work such as Haboob [96] and Knot [95], we use the same parameters as these works used – 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. Measurements here also adopt the persistent connection model from those tests, with clients issuing 5 requests per connection before closing it. With these parameters, the per-client throughput level is comparable to SPECweb99's quality-of-service requirements.

3.2.3 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 model, 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 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.

3.3 Blocking in Web Servers

In this section we discuss how blocking can be observed in various servers and the underlying causes for the blocking.

3.3.1 Observing Blocking in Flash

In Chapter 2 we discussed 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 when using our workloads. 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()` further confirms the analysis of blocking.

These calls take a list of file descriptors as input, and returns a count of how many of them are ready for activity. They usually form the main loop of an event-driven server, and are invoked as many times as needed as long as the system is active. Since we only have 1024 simultaneous connections and do not keep files open, the polling cost is relatively low. Thus at this load level, even the more scalable event delivery mechanism, `kevent()` does not seem to help. We show a CDF of the returned ready descriptors from `select()` and `kevent()` in Figure 3.1. The results 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.

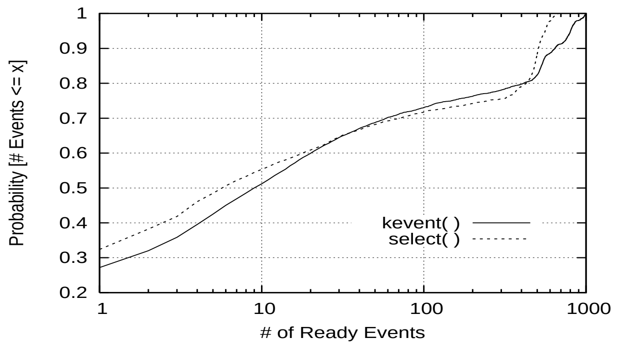


Figure 3.1: CDF of number of ready events (the return values from `select()`) in Flash

At these levels of activity, no free CPU should exist – the main loop should call `select()` or `kevent()` more often, decreasing the number of ready descriptors per call. 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.

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

3.3.2 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 any easily-testable invariant 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, 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 the more processes involved, the higher the burstiness, and the more variability in the behavior of the run queue. Because it is hard to differentiate unnecessary blocking in the blocked process queue length, we measure the number of runnable processes to reflect the burstiness.

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 3.2, we show what percentage of the Apache processes is runnable at any given time.

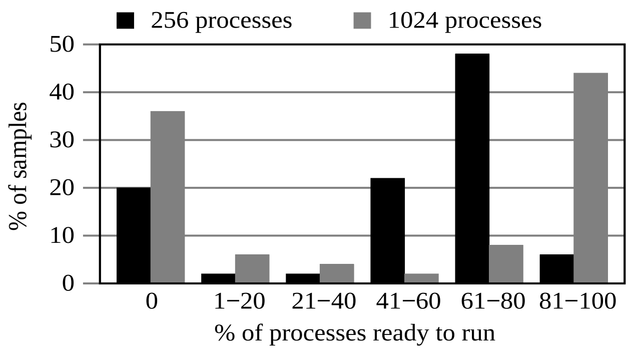


Figure 3.2: Scheduler burstiness (via the instantaneous run queue lengths) in Apache for 256 and 1024 processes

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."

3.3.3 Causes of Blocking

Our earlier work on developing the DeBox tool [77] 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 3.3. 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.

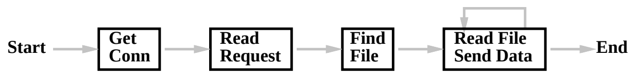


Figure 3.3: HTTP request processing steps

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 contention 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. This type of problem is not FreeBSD-specific – in

Linux, we have observed metadata cache misses commonly occurring when the data set exceeds the physical memory size, causing blocking in otherwise cached requests.

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 designers rely 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()` or `stat()` call is unexpected. When a completed disk access releases an exclusive lock, all of the processes waiting for it become runnable, leading to the bursty scheduler behavior we observed.

Another source of blocking is rooted in data-sending system calls such as `sendfile()`. When these system calls are operating on data files which require disk I/O, or exhaust related buffer resources such as the sendfile buffer, event-driven servers have to block, thus other requests are delayed until the process is unblocked. Asynchronous system calls may reduce chances of blocking on disk I/O but do not reduce buffer pressure. Blocking caused by these system calls may have less impact on process or threaded servers.

3.3.4 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 3.4 and 3.5, 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. The mean latency for Apache becomes noticeably worse at that level, with a

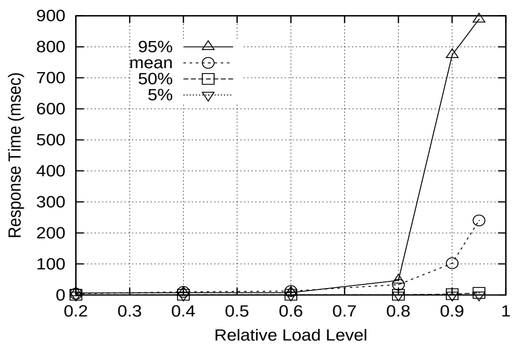


Figure 3.4: Apache Latency Profile. The relative load of 1.0 equals 241 Mb/s

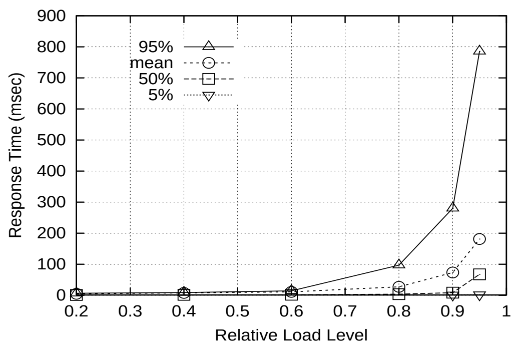


Figure 3.5: Flash Latency Profile. The relative load of 1.0 equals 336 Mb/s

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 3.6 and 3.7. Both servers exhibit latency degradation as the server load approaches infinite demand, with the median value rising over one hundred fold. 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.

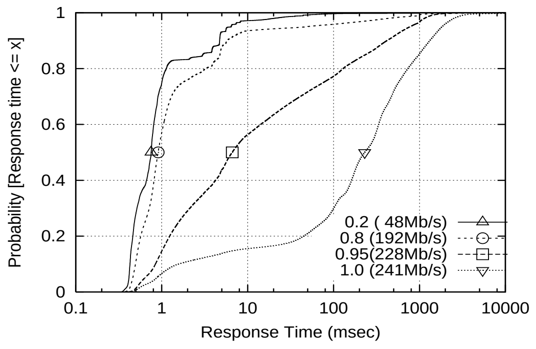


Figure 3.6: Apache latency CDFs for various load levels

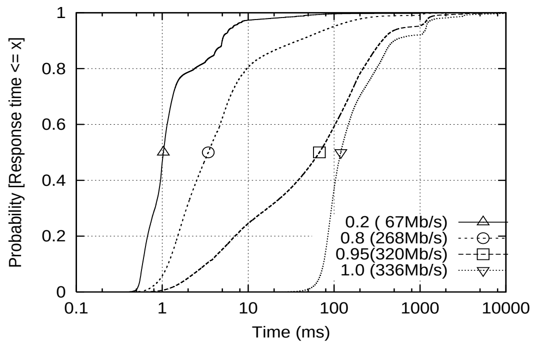


Figure 3.7: Flash latency CDF for various load levels

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.

3.3.5 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 3.8 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

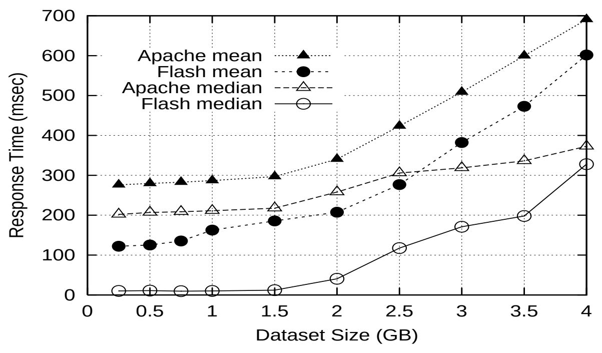


Figure 3.8: Median and mean latencies of Apache and Flash with various data set sizes

served out of the filesystem cache. The cache hit rate is in line with what we showed in Table 1.2, which is discussed in Section 1.1. These factors confirm that the small amount of cache miss activity is interfering with the accesses for requests 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 in disk speed, 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.

3.4 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. We term this phenomenon *service inversion* since the resulting latencies would be inverted compared to the ideal latencies. The term is conceptually similar to priority inversion in OSes, but in different subjects. 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.

To understand how head-of-line blocking causes service inversion, consider the scenario in Figure 3.9, 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.

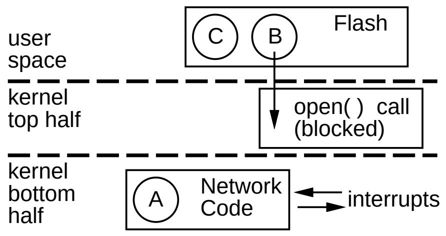


Figure 3.9: Service inversion example – Assume three requests (A, B, and C) arrive at the same time, and A is processed first. If it is cached and is sent to the networking code in the kernel bottom half, interrupt-based processing for it can continue even if the the process gets blocked. In this case, even if A is large, it may get finished before processing on C even starts.

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.

3.4.1 Identifying Service Inversion

To qualitatively understand the prevalence of service inversion, we take the latency CDFs from Figures 3.6 and 3.7 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 3.2.

The graphs in Figures 3.10 and 3.11 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.

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 3.2: Workload categories for latency breakdowns

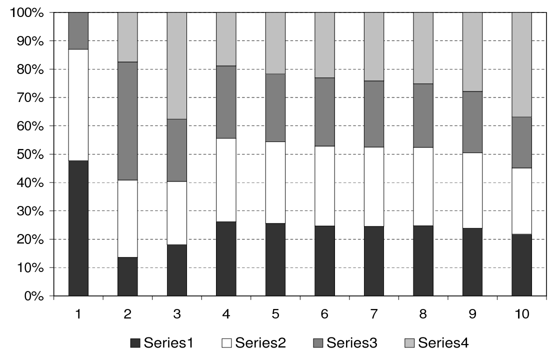


Figure 3.10: Apache CDF breakdown by decile at load 0.95

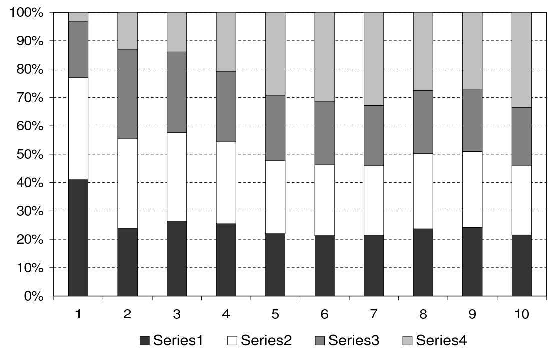


Figure 3.11: Flash CDF breakdown by decile at load 0.95

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 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 this chapter.

3.4.2 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:

$$\sum_{i=1}^{n} Distance(i)/\lfloor n^2/2 \rfloor \tag{3.1}$$

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

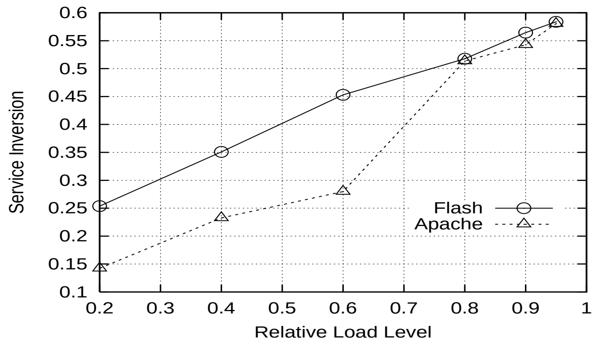


Figure 3.12: Service inversion versus load level for Apache and Flash

distance of file B is 1, C is 1, A is 2, and D, E are 0. The inversion value is $4/12 = 0.33$. Since this measurement requires only the response sizes and latencies, as long as the distribution of sizes is the same, it can be used to compare two different servers or the same server at multiple load levels. To handle the case of multiple requests with the same response size, we calculate distance by comparing the N^{th} observed position with the N^{th} ideal position for each response of the same size.

By measuring service inversion as a function of load level, we discover that this effect is a major contributor to the latency increase under load. Figure 3.12 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.

3.5 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 new servers overcome the latency and blocking problems previously observed. In our earlier work on DeBox [77], 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.

3.5.1 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 process, 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 takes the responsibility of finding the requested file, opening the file, and sending the file descriptor and metadata information back to the Apache processes. Upon receiving a valid open file descriptor from the backend, the Apache process can return the associated data to the client. Since the backend handles URL lookup for all Apache processes, it is possible to combine duplicated requests and even preload data blocks into the filesystem cache before passing the control back to Apache processes, thus reducing the number of context switches and the chances of more blocking. We call this new server Flashpache, to reflect its hybrid architecture. Figure 3.13 shows major components of Flashpache.

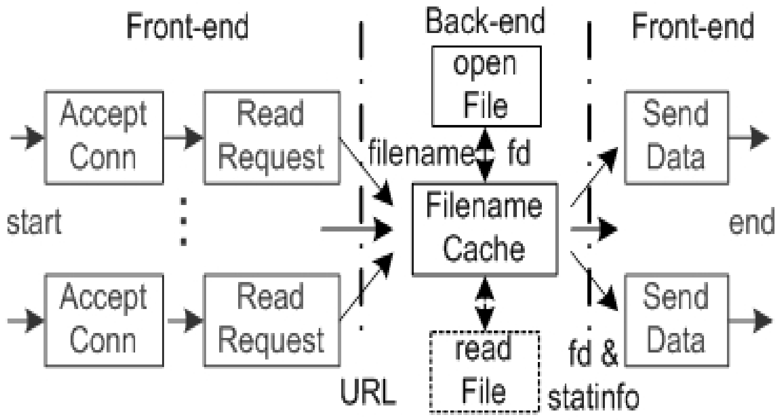


Figure 3.13: Flashpache architecture

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 100 lines of code.

This architecture 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.

3.5.2 Latency Results

We analyze the latency of the new servers by repeating our earlier experiments to understand latency and blocking. We begin our evaluation by repeating the burstiness measurement, which indicates that blocking-induced burstiness has also been reduced or eliminated in both servers. Figure 3.14 shows number of ready events for the original Flash server, the new server, as well as the intermediate steps of file descriptor passing (fd pass) and removing memory-mapped files (no mmap). 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, Figure 3.15 shows the distribution of ready processes of the Flashpache server. 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.

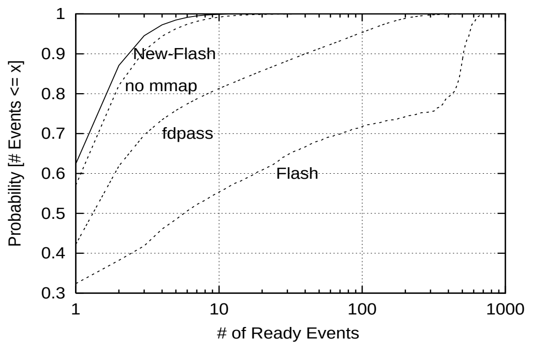


Figure 3.14: CDFs of # of ready events for Flash variants

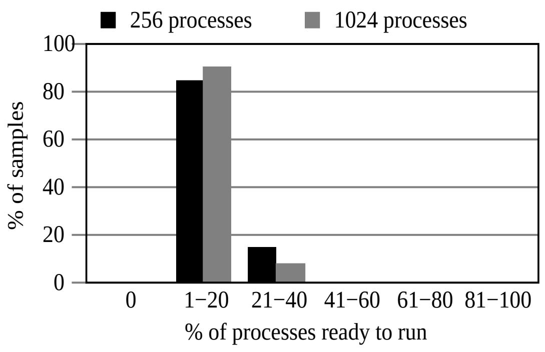


Figure 3.15: Scheduler burstiness in Flashpache for 256 and 1024 processes

We evaluate step-by-step improvements to Flash with the results shown in Table 3.3. Included are the figures for the original server and the intermediate steps. 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%.

	Latency (ms)			Capacity (Mb/s)
	median	mean	90%	
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 3.3: Latencies & capacities for original and modified servers

The more impressive result is the drastic 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 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 3.16 shows that median latency no longer grows with data set size, despite the increase in mean latencies. Mean latency still increases due

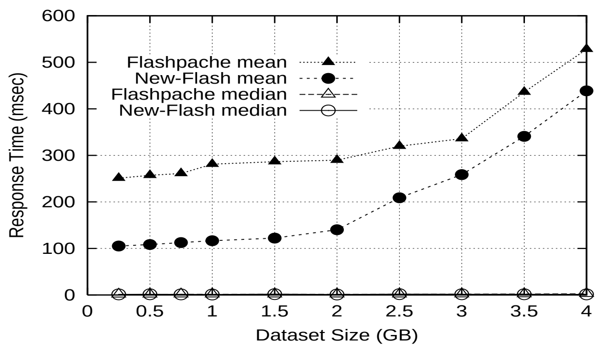


Figure 3.16: Response time of New-Flash and Flashpache with different data set sizes

to cache misses, but the median request is a cache hit in all cases. Figures 3.17 and 3.18 show the latency CDFs for 5^{th} percentile, mean, median, and 95^{th} percentile with varying load. Though the mean latency and 95^{th} percentile increase, the 95^{th} percentile shows less than a tripling versus its minimum values, which is much less growth than the two orders of magnitude observed originally. The other values are very flat, indicating that most of the requests are served with the same quality at different load levels. More importantly, the 95^{th} percentile CDF values are lower than the mean latency. The reason for this is that the time spent on the largest requests (the last 5%) is much higher compared to time spent on other requests. This result conforms to the workload expectations stated in Table 1.2.

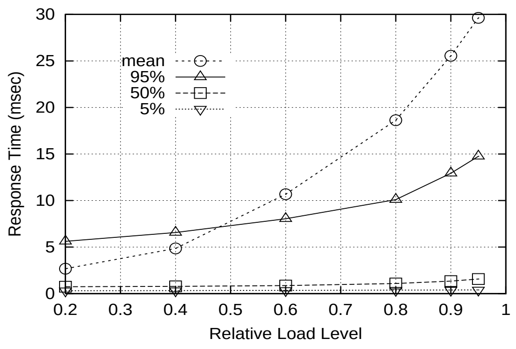


Figure 3.17: Latency profile of New-Flash (Flash profile shown in Figure 3.5)

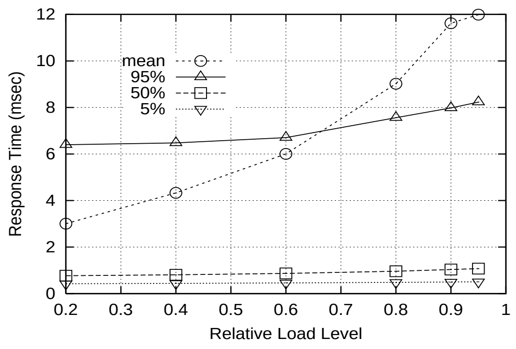


Figure 3.18: Latency profile of Flashpache (Apache profile shown in Figure 3.4)

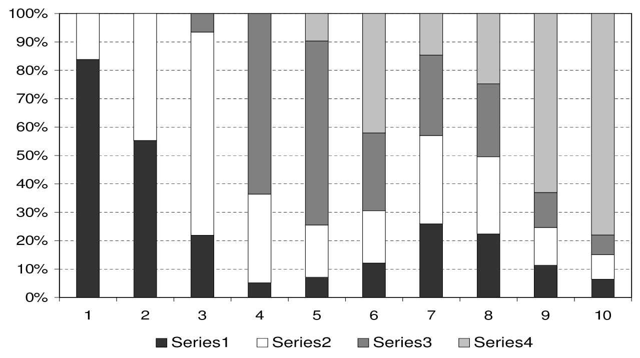


Figure 3.19: CDF breakdown for New-Flash on 3.0 GB data set, load 0.95

3.5.3 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 3.19 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 3.20 shows the change of the inversion value with the load level. Compared to the old system, we reduce the inversion by over 40%, suggesting requests are treated more fairly in the new system. The fact that the inversion value still increases with the load is a matter for further investigation. However, this may be a limitation of our service inversion calculation itself.

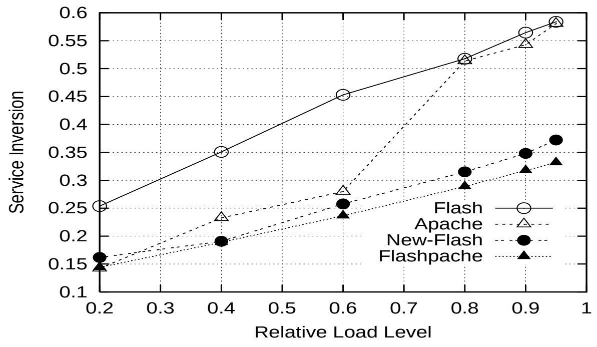


Figure 3.20: Service inversion of original and modified servers

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 3.21 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 3.22, is similar. The values for Flashpache and its original counterpart are also shown in Figure 3.20, and we can see that our modifications have almost halved the inversion under high load.

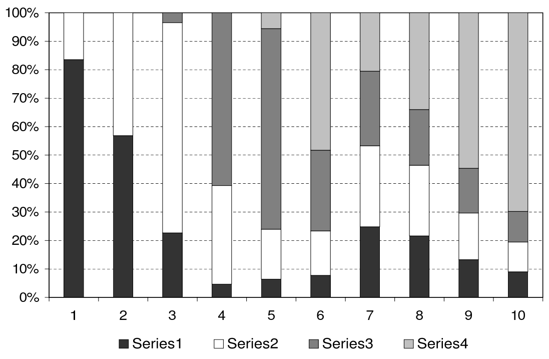


Figure 3.21: CDF breakdown for New-Flash on in-memory workload, load 0.95

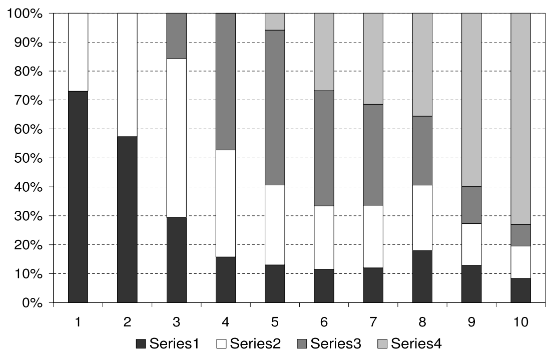


Figure 3.22: CDF breakdown for Flashpache on 3.0 GB data set, load 0.95

3.5.4 Latency Scalability

To understand how latencies are affected by processor speed, we use three generations of hardware with various processor speeds but sharing most of the other hardware components. Details about our server machines are shown in Section 3.2. 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 3.4, 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.

	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 3.4: Capacities of original and modified servers across three processor generations and different workloads

A more detailed examination of server latency is shown in Figures 3.23 and 3.24. 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

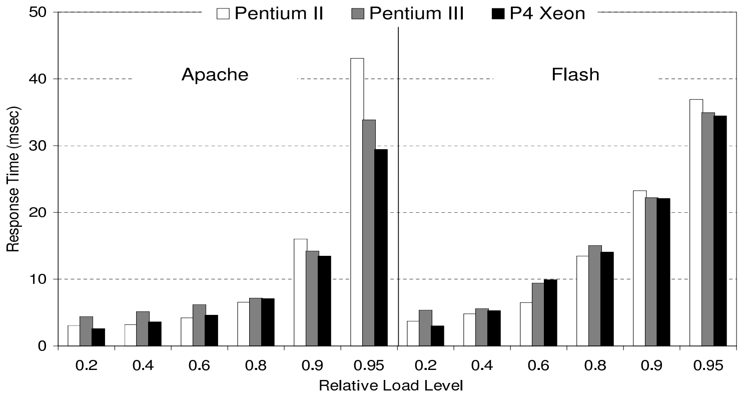


Figure 3.23: In-memory workload (0.5 GB) latency profiles of Apache and Flash across three processor generations

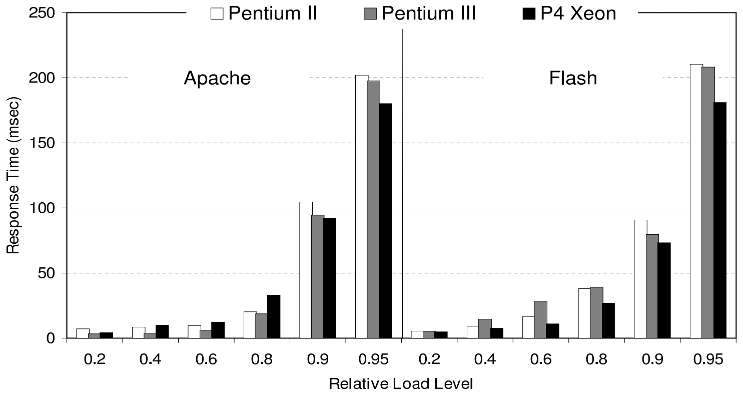


Figure 3.24: Disk-bound workload (3.0 GB) latency profiles of Apache and Flash across three processor generations

only minor improvement with processor speed. Figure 3.25 shows the scalability of our new servers across processors. Both of the servers show much lower latencies on all of the three 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 the faster processors.

In summary, both new servers demonstrate lower initial latencies, slower growth in latency, and better decrease of latency with processor speed. These servers are no longer dominated by disk access times, and should scale with improvements in processors. That

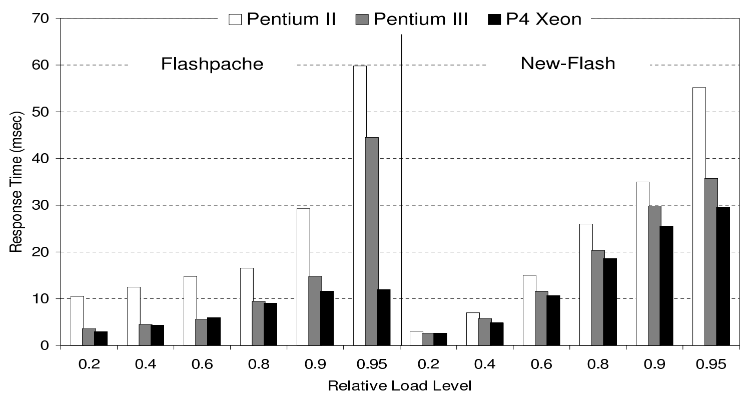


Figure 3.25: Disk-bound workload (3.0 GB) latency profile of New-Flash and Flashpache across three processor generations

these changes eliminate over 80% of the latency answers the question about latency origins – these latencies were dominated by blocking, rather than request queuing.

3.6 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 [62] 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 [50]. Some of the most aggressive designs have used raw disk, eliminated standard interfaces, and eliminated reliable metadata in order to gain performance [81]. In comparison, we have shown that no kernel or filesystem changes are necessary to achieve much better latency, and we show that these techniques can be retrofitted to legacy servers with low cost. Our investigation of blocking has also been much finer-grained, usually at resource locking level, which is not amenable to previous work in asynchronous I/O interfaces.

More recently, much attention is paid to latency measurement and improvement. Rajamony & Elnozahy [70] 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 [13]. Olshefski et al. [59] propose a way of inferring client response time by measuring server-side TCP behaviors. Improvement techniques have been largely limited to connection scheduling, with most of the attention focused on the SRPT policy [25, 31], including server modification and kernel instrumentation for network stack scheduling [31]. Cohort scheduling [45] focuses on gaining performance by batching similar requests but does not examine why queuing occurs.

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. [71] 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. [65] and Jurczyk et al. [39] 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.

Our approach of fairness evaluation may be more suitable for network servers than the Jain fairness index [37] used in other work [96], since we focus more on the latencies of individual requests rather than coarse-grained characteristics of clients. Bansal & Harchol-Balter [11] investigate the unfairness of SRPT scheduling policy under heavy-tailed workloads and draw the conclusion that the unfairness of their approach barely noticeable. Our approach does not have this concern, since we address the latency issues directly rather than try to schedule around them.

Chapter 4

Server Performance on Simultaneous Multithreaded Processors

With the rapid development in processor architecture, it is always interesting for server designers to test server performance on new hardware platforms, because network servers are usually hardware-demanding. This chapter examines server performance on simultaneous multithreaded processors using multiple server software packages on multiple processors

4.1 Introduction

Simultaneous multithreading (SMT) has recently moved from simulation-based research to reality with the advent of commercially available SMT-capable microprocessors. Simultaneous multithreading allows processors to handle multiple instruction streams in the pipeline at the same time, allowing higher functional unit utilization than is possible from a single stream. Since the hardware support for this extra parallelism seems to be minimal, SMT has the potential to increase system throughput without significantly affecting system cost. While academic research on SMT processors has been performed since the mid-1990's [27, 92], the recent availability of SMT-capable Intel Xeon processors allows performance analysts to perform direct measurements of SMT benefits under a wide range of workloads.

One of the biggest opportunities for SMT is in network servers, such as Web, FTP, or file servers, where tasks are naturally parallel, and where high throughput is important. While much of the academic focus on SMT has been on scientific or computation-intensive workloads, suitable for the High Performance Computing (HPC) community, a few simulation studies have explicitly examined Web server performance [52, 73]. The difficulty of simulating server workloads versus HPC workloads is in accurately handling operating system (OS) behavior, including device drivers and hardware-generated interrupts. While processor-evaluation workloads like SPEC CPU [87] explicitly attempt to avoid much OS interaction, server workloads, like SPECweb [88] often include much OS, filesystem, and network activity..

While simulations clearly provide more flexibility than actual hardware, evaluation on real hardware also has its advantages, including more realism and faster evaluation. Using actual hardware, researchers can run a wider range of workloads (e.g., bottom-half heavy workloads) than is feasible in simulation-based environments. Particularly for workloads with large data set sizes that are slow to reach steady state, the time difference between simulation and evaluation can be substantial. The drawback of hardware, however, is the lack of configuration options that is available in simulation. Some flexibility in the hardware analysis can be gained by using processors with different characteristics, though this approach is clearly much more constrained than simulators.

Our evaluation suggests that the current SMT support is sensitive to application and workloads, and may not yield significant benefits for network servers, especially for OS-heavy workloads. We find that enabling SMT usually produces only slight performance gains, and can sometimes lead to performance loss. In the uniprocessor case, simulations appear to have neglected the OS overhead in switching from a uniprocessor kernel to an SMT-enabled kernel. The performance loss associated with such support is comparable to the gains provided by SMT. In the 2-way multiprocessor case, the higher number of memory references from SMT often causes the memory system to become the bottleneck, offsetting any processor utilization gains. This effect is compounded by the growing gap between processor speeds and memory latency. We find that SMT on the Xeon tends to provide better gains when coupled with large L3 caches. By comparing performance gains across variants of the Xeon, we argue that such caches will only become more crucial for SMT as clock rates increase. If these caches continue to be one of the differentiating factors between commodity and higher-cost processors, then commodity SMT will see eroding gains going forward. We believe this observation also applies to architectures other than the Xeon, since SMT only yields benefits when it is able to utilize more processor resources.

Using these results, we can also examine how simulation suggested a much more optimistic scenario for SMT, and why it differs from what we observe. For example, when calculating speedups, none of the simulations used a uniprocessor kernel when measuring the non-SMT base case. Furthermore, the simulations use cache sizes that are larger than anything commonly available today. These large caches appear to have supported the higher number of threads used, yielding much higher benefits than what we have seen, even when comparing with the same number of threads. We do not believe that the processor models used in the simulation are simply more aggressive than what is available today or likely to be available in the near-future. Instead, using comparable measurements from the simulations and existing hardware, we show that the type of processors commonly modeled in the simulations is unlikely to ever appear as slightly-modified mainstream processors. We argue that they have characteristics that suggest they could be built specifically for SMT, and would sacrifice single-thread performance.

In summary, this chapter makes four contributions: (1) We provide a thorough experimental evaluation of SMT for network servers, using five different software packages and three hardware platforms. We believe this study is more complete than any related work previously published. (2) We show that SMT has a smaller performance benefit than expected for network servers, both in the uniprocessor and dual-processor cases. In each case, we identify the macro-level issues that affect performance. (3) We perform a microarchitectural evaluation of performance using the Xeon's hardware performance counters. The results provide insight into the instruction-level issues that affect performance on these platforms. (4) We compare our measurements with earlier simulation results to understand what aspects of the simulated processors yielded much larger performance gains. We discuss the feasibility of these simulation models, both in the context of current hardware, and with respect to expected future trends.

4.2 Background

In this section we present an overview of the Intel Xeon processor with Hyper-Threading (Intel's term for SMT), then describe our experimental platform including hardware parameters and server configuration, our workloads and measurement methodology.

4.2.1 SMT Architecture

The SMT architecture was proposed in the mid-1990's, and has been an active area for academic research since that time [49, 91, 92], but the first general-purpose processor with SMT features was not shipped until 2003. The main intent of SMT is to convert thread-level parallelism into instruction-level parallelism. In SMT-enabled processors, instructions from multiple processes or threads can be fetched together, without context switching, and can be executed simultaneously on shared execution resources. From either the operating system's or user program's perspective, the system appears to have multiple processors. Currently, we are aware of only two processors in production that support SMT – the Intel Xeon with Hyper-Threading and the IBM POWER5. The Xeon has been available longer, and since it is available in a wide range of configurations, it provides us with an opportunity to affordably evaluate the impact of several features.

The Xeon is Intel's server-class x86 processor, designed to be used in high-end applications. It is differentiated from the Pentium 4 by the addition of extra on-chip cache, support for SMT (though this is now beginning to appear on standard P4 processors), and on-chip support for multiprocessing. It is a superscalar, out-of-order processor with a deep pipeline, ranging from 20 to 30 stages depending on processor version and clock speed. It has two hardware contexts (threads) per processor, which share most of the resources, such as caches, execution units, branch predictor, control logic, and buses. Its native x86 instruction set architecture is CISC, but it internally translates instructions into RISC-like micro-operations (μops) before executing them. Buffering queues between major pipeline logic blocks, such as μop queues, and the reorder buffer, are partitioned when SMT is enabled, but are recombined when only one software thread is active [51]. The basic hardware information for the Xeon can be found in Table 4.1.

clock rate	2.0 or 3.06 GHz
pipeline	20 stages or 30 stages starting from the TC
Fetch Policy	6 μops per cycle round robin for logical processors
Retirement	3 μops per cycle
Shared Resources	caches, branch predictors, decoder logic DTLB, execution units, buses
Duplicated Resources	interrupt controller, status registers ITLB, renaming logic
Partitioned Resources	μop queue, re-ordering buffer load/store buffer, general instruction buffer

Table 4.1: Intel Xeon hardware parameters.

4.2.2 Experimental setup

To reduce the number of variables in our experiments, all of our tests use the same motherboard, an Intel SE7505VB2 with 4GB memory, which is capable of supporting up to two processors. Our processors are the 3.06 GHz Xeon with no L3 cache, the 3.06 GHz Xeon with a 1MB L3 cache, and the 2.0 GHz Xeon without L3 cache. Using these three processors, we can determine the effect of different clock rates, and the effect of the presence or absence of an L3 cache. All processors have a 533 MHz front-side bus (FSB). The 2.0 GHz use a 20-stage pipeline starting from the trace cache (TC), while the 3.06 GHz Xeons use a 30-stage pipeline. All tests use the same physical motherboard, and we manually replace processors as needed, in order to reduce the chance that variations in memory manufacturing, etc., can affect the results. The memory hierarchy details for our system are provided in Table 4.2. Using lmbench [53], we find the main memory latencies are 225 cycles for the 2.0 GHz Xeon, 320 cycles for the 3.06 GHz Xeon with L3, and 344 cycles for the 3.06 GHz processor without L3 cache.

The increase in memory latency (measured in cycles) for the 3.06 GHz processors is not surprising, since the cycles are shorter in absolute time. The absolute latency is rela-

Level	Capacity	Associa-tivity	Line Size	Latency (cycles)
TC	12K μops	8 way	6 μops	N/A
D-L1	8 KB	4 way	64 bytes	2
L2	512 KB	8 way	128 bytes	18
Memory	4 GB	N/A	N/A	225 - 344
ITLB	128 entries, 20 cycles miss penalty			
DTLB	64 entries, 20 cycles miss penalty			

Table 4.2: Intel Xeon memory hierarchy information. The latency cycles of each level of the memory hierarchy includes the cache miss time of the previous level

tively constant since the FSB speed is the same. The impact on bandwidth is 22%, much less than the clock speed difference – the 2.0 GHz system has a read bandwidth of 1.8 GB/sec while the 3.06 GHz system has a value of 2.2 GB/sec. While higher bandwidth is useful for copy-intensive applications, the memory latency is more important to applications that perform heavy pointer-chasing. Early Web servers performed significant numbers of memory copies to transfer data, but with the introduction of zero-copy [63] support into servers, copy bandwidth is less of an issue.

Our testing harness consists of 12 uniprocessor client machines with AMD Duron processors at 1.6 GHz. The aggregate processor power of the clients is enough to ensure that the clients are never the bottleneck. To ensure adequate network bandwidth, the clients are partitioned into four groups of three machines. Each group is connected to the server via a separate switched Gigabit Ethernet, using four Intel e1000 MT server adapters at the server.

We compare five different OS/processor configurations, based on whether a uniprocessor or multiprocessor kernel is used, and whether SMT is enabled or disabled. Using the BIOS support and OS boot parameters, we can select between one or two processors, and enable or disable SMT. For most of our tests, we use a multiprocessor-enabled

	1T-UP	1T-SMP	2T	2P	4T
# CPUs	1	1	1	2	2
SMP kernel	No	Yes	Yes	Yes	Yes
SMT enabled	No	No	Yes	No	Yes

Table 4.3: Notation used in this paper reflecting different hardware and kernel configurations

(SMP) kernel, since the OS sees an SMT-enabled processor as two logical processors. However, when we run with one physical processor and SMT disabled, we also test on a uniprocessor kernel. These combinations yield the five configurations studied in this paper: one processor with uniprocessor kernel (1T-UP), one processor with SMP kernel (1T-SMP), one processor with SMP kernel and SMT enabled (2T), two processors (2P), and two processors with SMT enabled (4T). Key features of the five configuration and their names used in this paper are shown in Table 4.3. The operating system on the server is Linux, with kernel version 2.6.8.1. This version includes optimizations for SMT, which we enable. The optimizations are described next.

4.2.3 Kernel Versions and Overheads

In evaluating SMT performance on uniprocessors, it is important to understand the distinction between the types of kernels available, because they affect the delivered performance. Uniprocessor kernels, as the name implies, are configured to only support one processor, regardless of how many physical processors are in the system. Multiprocessor kernels are configured to take advantage of all processors in the system using a single binary image. While intended for multiple processors, they are designed to operate without problems on a single processor.

Uniprocessor kernels can make assumptions about what is possible during execution, since all sources of activity are taking place on one processor. Specifically, the OS can make two important assumptions: that only one process or thread can be actively running in the kernel at once, and that when the kernel is executing on behalf of that process or thread, the only other source of execution is hardware interrupts. The first condition is important for protecting data in the kernel – when the kernel is executing, it generally does not have to worry about locking kernel data structures unless it may block on some resource. The only data sharing that remains is for data used by any interrupt servicing code. The existence of only one processor also simplifies this code, since it can simply disable interrupts when manipulating such data, and enable interrupts after the critical section. Since enabling or disabling interrupts is a single instruction on the x86, this code can be compact.

On multiprocessors (SMP), the invalidation of both assumptions causes the need to have more synchronization code in the kernel, leading to more overhead. Both processors can be executing kernel code simultaneously, so any global data in the kernel must be protected from race conditions. The simplest approach, using a "giant kernel lock" to ensure only one processor is in the kernel at a time, reduces the performance of OS-intensive workloads, and has been replaced with fine-grained locking on all major OSes. Interrupt handling must also differ – since interrupts can be delivered to a different processor than the one using data shared with the interrupt handler, the kernel cannot simply locally disable interrupts. Instead, all data accessible by an interrupt handler must also be protected using locks, to prevent another processor from accessing it simultaneously.

For uniprocessors, running a multiprocessor version of the kernel can therefore cause a much larger performance loss than might be expected, because instead of one extra lock operation per system call, many lock operations may be necessary for fine-grained data sharing. For network servers, this overhead can be significant, if every packet and acknowledgment invokes extra code that is not necessary in the uniprocessor case.

Since the OS treats an SMT-enabled processor as two logical processors, it must use the SMP kernel, with the associated overheads. Kernel designers have taken steps to reduce some overheads, knowing that some operations can be performed more efficiently on an SMT with two logical processors than a multiprocessor with two physical processors. However, since SMTs interleave instructions from multiple contexts, these overheads cannot be reduced to the level of uniprocessor kernels. The Linux kernel implements a number of SMT-specific optimizations, mostly related to processor affinity and load balancing [12]. Task run queues are shared between contexts on each physical processor, eliminating the chance of one context being idle while the other has multiple tasks waiting. This balancing occurs whenever a task wakes up or when any other task on the same physical processor finishes. Processor affinity, intended to minimize cache disruption, is also performed on physical processor instead of logical processors.

4.2.4 Test & Measurement methodology

We focus on Web (HTTP) servers and workloads because of their popularity and the diversity of server implementations available. The server applications we use are Apache 2.0 [6], Flash [62], TUX [93], and Haboob [96]. Each server has one or more distinguishing features which increases the range of systems we study. All of the servers are written in C, except Haboob, which uses Java. TUX is in-kernel, while all of the others are user-space. Flash and Haboob are event-driven, but Haboob also uses threads to isolate different steps of request processing. We run Apache in two configurations – with multiple-processes (dubbed Apache-MP), and multiple threads (dubbed Apache-MT) using Linux kernel threads, because the Linux 2.6 kernel has better support for threads than the 2.4 series, and the Xeon has different cache sharing for threaded applications. Threaded applications share the same address space register while multi-process applications usually have different registers. Flash has a main process handling most of the work with helpers for disk IO access. We run the same number of Flash main processes as the number of hardware contexts. TUX uses a thread-pool model, where multiple threads handle ready events. With the exception of Haboob, all of the servers use the zero-copy interfaces available on Linux, reducing memory copy overhead when sending large files. For all of the servers, we take steps described in the literature to optimize their performance. While performance comparison among the servers is not the focus of this paper, we are interested in examining performance characteristics of SMT on these different software styles.

We use the SPECweb96 [88] benchmark mostly because it was used in previous simulation studies. Compared to its successor, the SPECweb99 benchmark, it spends more time in the kernel because all requests are static, which resembles other server workloads such as FTP and file servers. We also include SPECweb99 benchmark results for comparison. SPECweb is intended to measure a self-scaling capacity metric, which means that the workload characteristics change in several dimensions for different load levels.

To simplify this benchmark while retaining many of its desirable properties, we use a more tractable subset when measuring bandwidths. In particular, we fix the data set size of the workload to 500MB, which fits in the physical memory of our machine. We perform measurements only after an initial warm-up phase, to ensure that all necessary files have been loaded into memory. During the bandwidth tests, no disk activity is expected to occur. We disable logging, which causes significant performance losses in some servers. SPECweb99 measures the number of simultaneous connections each server is able to sustain while providing the specified quality of service to each connection. The SPECweb99 client software introduces latency between requests to decrease the per-connection bandwidth. SPECweb96 does not have this latency, allowing all clients to issue requests in a closed loop, infinite-demand model. We use 1024 simultaneous connections, and report the aggregate response bandwidth received by the clients.

We use a modified version of OProfile [61] to measure the utilization of microarchitectural resources via the Xeon's performance-monitoring events. Since we are interested in system-wide performance, we do not need the granularity available in DeBox. Moreover, our current DeBox implementation doenot explore processor level performance events. OProfile ships with the Linux kernel and is able to report user, kernel or aggregated event values. OProfile operates similarly to DCPI [4], using interrupt-based statistical sampling of event counters to determine processor activity without much overhead. We find that for our experiments, the measurement overhead is generally less than 1%. While OProfile supports many event counts available on the Xeon, we enhance the released code to support several new events, such as L1 data cache miss, DTLB miss, memory loads, memory stores, resource stalls, etc.

4.3 SMT Performance

In this section we evaluate the throughput improvement of SMT in both uniprocessor and multiprocessor systems. Particular attention is given to the comparison between configurations with and without SMT enabled, and kernels with and without multiprocessor support. We first analyze trends at a macroscopic level, and then use microarchitectural information to understand what is causing the macroscopic behavior. Our bandwidth result for the basic 3.06 GHz Xeon, showing five servers and five OS/processor configurations, can be seen in Figure 4.2. Results for 2.0 GHz and 3.06 GHz with L3 cache are seen in Figures 4.1 and 4.3, respectively. For each server, the five bars indicate the

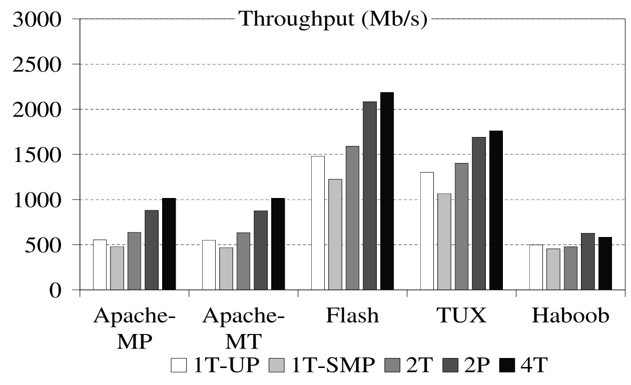


Figure 4.1: Throughput of Xeon 2.0GHz processor without L3 cache

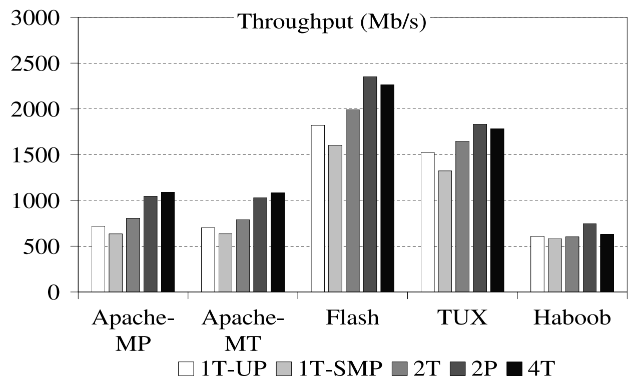


Figure 4.2: Throughput of base Xeon 3.06GHz processor

maximum throughput achieved using the specified number of processors and OS configuration.

While bandwidth is influenced by both the server software as well as the OS/processor configuration, the server software usually has a large effect (and in this case, dominant effect) on bandwidth. Heavily-optimized servers like Flash and TUX are expected to outperform Apache, which is designed for flexibility and portability instead of raw performance. The relative performance of Apache, Flash, and Haboob is in-line with previous studies [77]. TUX's relative performance is somewhat surprising, since we assumed an in-kernel server would beat all other options. To ensure it was being run correctly, we

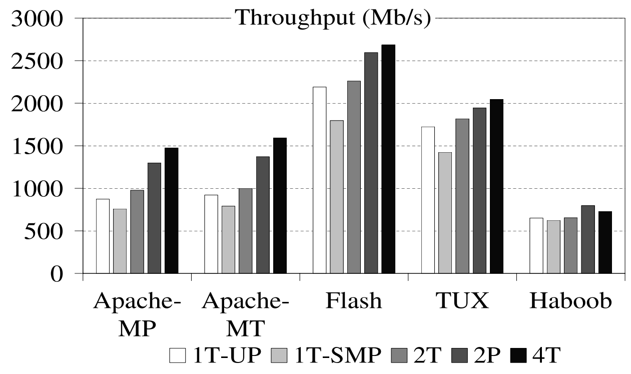


Figure 4.3: Throughput of Xeon 3.06GHz processor with 1MB L3 cache

consulted with its author to ensure that it was properly configured for maximum performance. We surmise that its performance is due to its emphasis on dynamic content, which is not exercised in this portion of our testing. Haboob's low performance can be attributed both to its use of Java as well as its lack of support for Linux's sendfile system call (and as a result, TCP checksum offload). For in-memory workloads, the CPU is at full utilization, so the extra copying, checksumming, and language-related overheads consume processor cycles that could otherwise be spent processing other requests.

4.3.1 SMP Overhead on Uniprocessor

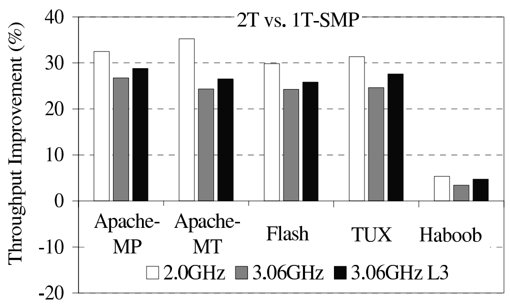
We can quantify the overhead of supporting an SMP-capable kernel by comparing the 1T-UP (one processor, uniprocessor kernel) value with the 1T-SMP (one processor, SMP kernel) value. The loss from uniprocessor kernel to SMP kernel on the base 3.06 GHz processor is 10% for Apache, and 13% for Flash and Tux. The losses on the L3-equipped processor and the 2.0 GHz processor are 14% for Apache and 18% for Flash and Tux, which are a little higher than our base system. The impact on Haboob is relatively low (4%-10%), because it performs the most non-kernel work. The magnitude of the overhead is fairly large, even though Linux has a reputation of being efficient for low-degree

SMP configurations. This result suggests that, for uniprocessors, the performance gained from selecting the uniprocessor kernel instead of SMP kernel can be significant for these applications.

The fact that the impacts are larger for both the slowest processor and the processor with L3 are also interesting. However, if we consider these results in context, it can be explained. The extra overheads of SMP are not only the extra instructions, but also the extra uncacheable data reads and writes for the locks. The fastest system gets its performance boost from its L3 cache, which makes the main memory seem closer to the processor. However, the L3 provides no benefit for synchronization traffic, so the performance loss is more pronounced. For the slowest processor, the extra instructions are an issue when the processor is running at only two-thirds the speed of the others.

4.3.2 Uniprocessor SMT Benefits

Understanding the benefits of SMT for uniprocessors is a little more complicated, because it must be compared against a base case. If we compare 1T-SMP to 2T (uniprocessor SMT), the resulting graphs would appear to make a great case for SMT, with speedups in the 25%-35% range for Apache, Flash and TUX, as shown in Figure 4.4. However, if we compare the 2T performance versus 1T-UP, then we see that the speedups are much more modest. These comparisons are shown in Figure 4.5, for all three processor types. In general, the relative gain decreases as processor becomes faster (via clock speed or cache). Apache-MT's gain on the 2.0 GHz processor is the highest at 15%, but this drops to the 10%-12% range for faster processors. The gains for Flash and TUX are less, dropping to the 3%-5% range for the faster processor. The Haboob numbers show the opposite trend from all other servers, showing a loss at 2.0 GHz improving to a small gain.



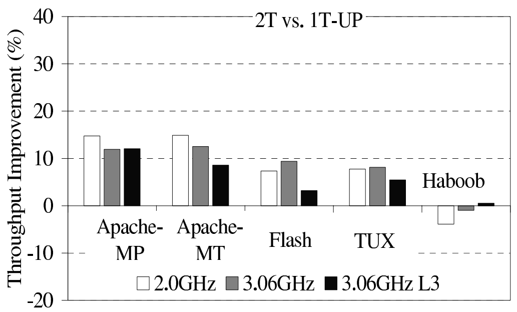


Figure 4.4: SMT speedup on uniprocessor system with SMP kernel

Figure 4.5: SMT speedup on uniprocessor system with different kernels

We believe that the correct comparison for evaluating uniprocessor SMT benefits is comparing the bandwidths with 1T-UP. Although the kernels are different, the SMP kernel needlessly hinders uniprocessor performance. For parallel algorithms, comparison with the best base case is also a standard speedup measurement technique. The performance of a parallel algorithm is compared to the performance of the best sequential algorithm. Simply put, the gain from choosing the appropriate kernel is comparable to the gain of upgrading hardware.

In comparing what is known about measured speedups from enabling the Xeon's SMT, our results are comparable to the 20%-24% gains that Tuck and Tullsen observed using other workloads [90]. Their speedup comparisons are performed using an SMP kernel for all measurements, which would be similar to comparing our 2T results to the 1T-SMP values. In fact, our observed speedups are slightly higher than theirs, if we discount Haboob. This result is in-line with the observation that SMT can potentially help server-style software more than other workloads [73]. The impact of using a uniprocessor kernel on the Tuck and Tullsen results is not clear – their workloads are not OS-intensive, so the performance loss of using an SMP kernel may be less than what we observed.

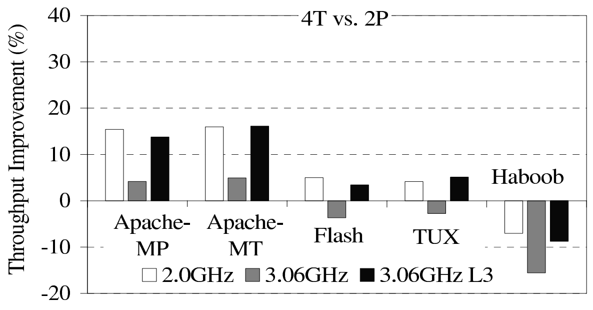


Figure 4.6: SMT speedup on dual-processor system

4.3.3 SMT in Dual-processor systems

The next reasonable point of comparison for SMT is in dual-processor systems, since these systems are particularly targeted to the server market and are rapidly approaching commodity status. Two factors responsible for this shift are the falling CPU prices and the support for low-degree multiprocessing built into some chips. The Xeon processors are available in two variants, the Xeon DP and the Xeon MP, with the distinguishing feature being the number of processors that can be used in one system. The DP ("dual-processor") line has on-chip support for building "glueless" 2-processor SMP systems that require no extra hardware to share the memory bus. The MP ("multiprocessor") line is intended for systems with more than 2 processors. In addition to the on-chip glue logic, the Xeon DP also drives commodification of dual-processor systems via pricing – as of this writing the Xeon DP is roughly one-tenth the cost of a Xeon MP at the same clock rate. Whether this difference stems from pricing strategies or economies of scale is unclear to us, but it does greatly magnify the price difference between dual-processor systems and larger multiprocessors.

We note that enabling SMT in a dual-processor configuration carries more risk than enabling it for uniprocessors. While the actual gains in uniprocessors may have been comparable to the loss in using an SMP kernel, the overall gains were still positive. As shown in Figure 4.6, enabling SMT in dual-processor configurations can cause a performance loss, even though the same kernel is being used in both cases. Haboob shows a 9%-15% loss on the three different processors, when comparing the 2P configuration to the 4T configuration. Flash and TUX show a loss in the base 3.06 GHz case, but show small gains for the other two processor types. The specifics of the performance curves lead us to believe that Flash and TUX are bottlenecked on the memory system – the base 3.06 GHz processor will have more memory traffic than its L3-equipped counterpart. Likewise, the 2 GHz processor has relatively faster memory, measured in CPU cycles, since the processor speed is slower. So, by taking a 2-processor system whose bottleneck is already memory, and increasing the memory demand, the overall performance will not improve. By the same reasoning, we can infer that Apache may be processor-bound, since it sees gains on all of the processors. The highest gain in this test, both in terms of absolute bandwidth and in percentage, is seen in the Apache-MT results for the L3-equipped 3.06 GHz processor. It gains 16% over the 2P configuration, jumping from 1371 Mb/s to 1593 Mb/s. The gain for Apache-MP on this processor is also significant, but smaller.

4.3.4 Understanding Relative Gains

These results are interesting because Apache is neither the best performer nor the worst – it appears to be in a “sweet spot” with respect to the benefits of SMT. This sweet spot may not be very large, in terms of the variety of configurations for which it works – Apache’s gain on the base 3.06 GHz is only 4%-5%. Going forward, it may be necessary to keep increasing cache sizes to prevent faster processors from being bottlenecked on memory.

If all of the contexts are waiting on memory, SMT may not be able to provide much benefit.

However, when using the relative gains, one should remember that they are compared only to the same server software, and may only reflect some artifact of that server. For instance, while Apache's relative gains are impressive, the absolute performance numbers may be more important for many people. Those show clearly that even the best Apache score for a given processor class never beats the worst Flash score, and almost never beats the worst TUX score. So, even with 2 SMT-enabled processors, Apache still does not perform as well as Flash (or TUX, generally) on a single processor.

4.3.5 Measuring the Memory Bottleneck

In the previous analysis, we have attempted to ascribe some of the performance characteristics of the various servers and configurations to their interaction with the memory system. To quantify these effects, we measure the cycles when the memory bus is occupied by any one of the threads, including both driving data onto or reading data from the bus. Even though the bus utilization figures do not differentiate "pointer chasing" styles of memory accessing from bulk data copying, by knowing the particular optimizations used by the servers, we can use this information to draw reasonable conclusions. To normalize the different processor clock speeds, bus utilization is calculated as follows:

$$Utilization = \frac{(CyclesBusOccupied) * (ClockSpeed)}{(NonHaltedCycles * BusSpeed)} \tag{4.1}$$

The bus utilization values, broken down by server software, configuration, and processor type, are shown in Figure 4.7. Several first-order trends are visible: bus utilization tends to increase as the number of contexts/processors is increased, is comparable for

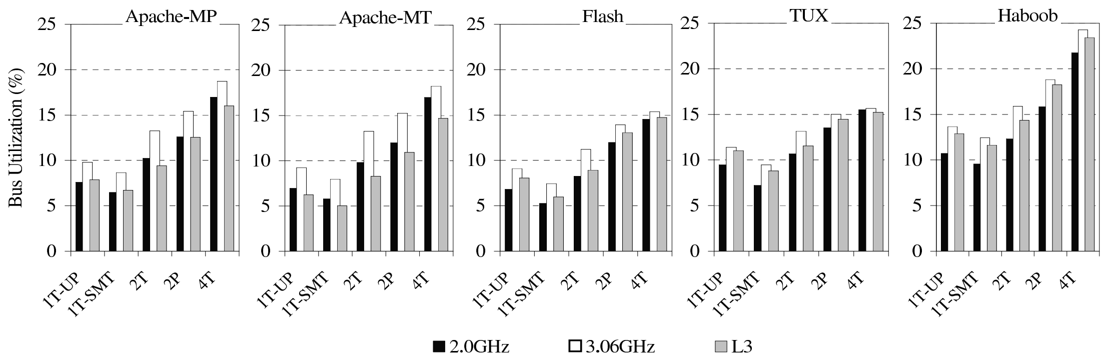


Figure 4.7: Bus utilization of three hardware configurations

all servers except Haboob, and is only slightly lower for L3-equipped processors. The trends can be explained using the observations from the bandwidth study, and provide strong evidence for our analysis about what causes bottlenecks.

The increased bus utilization for a given processor type as the number of processors and hardware contexts increase is not surprising, and is similar in pattern to the throughput behavior. Essentially, if the system is work-conserving, we expect bus utilization to be correlated with the throughput level. In fact, we see this pattern for the gain from the 2.0 GHz processor to 3.06 GHz – the coefficient of correlation between the throughput and the bus utilization is 0.95. The coefficient for the L3-equipped versus base 3.06 GHz Xeon is only 0.62, which is still high, and provides evidence that the L3 cache is definitely affecting the memory traffic. A more complete explanation of the L3 results are provided below.

The fact that Haboob's bus utilization looks different from others is explained by its lack of zero-copy support, and in turn explains its relatively odd behavior in Figures 4.5 and 4.6. The bulk data copying that occurs during file transfers will increase the bus utilization for Haboob, since the processor is involved in copying buffers and performing TCP checksums. However, the absolute utilization values mask a much larger difference

– while Haboob's bus utilization is roughly 50% higher than that of Flash or TUX, its throughput is one-half to one-third the value achieved by those servers. Combining those figures, we see that Haboob has a per-request bus utilization that is three to four times higher than the other servers.

The same explanation applies to the bus utilization for the L3-equipped processors, and to Apache's relative gain from SMT. The L3 cache absorbs memory traffic, reducing bus utilization, but for Flash and TUX, the L3 numbers are only slightly below the non-L3 numbers. However, the absolute throughput for the L3-equipped processors is as much as 50% higher, indicating that the per-request bus utilization has actually dropped. The differences in bus utilization then provide some insight into what is happening. For Flash and TUX, the L3 bus utilizations are very similar to the non-L3 values, suggesting that the request throughput increases until the memory system again becomes the bottleneck. For Apache, the L3 utilization is lower than the non-L3, suggesting that while the memory system is a bottleneck without the L3 cache, something else becomes the bottleneck with it. Since we know the memory system is capable of higher utilization, we can conclude that CPU processing is the bottleneck. The explanation addresses both Apache's benefit with a second processor as well as with SMT enabled. Since Apache has more non-memory operations than Flash or TUX, it can benefit from the additional CPU capacity.

4.4 Microarchitectural Analysis

To understand the underlying causes of the performance we observed, we use the hardware events available on the Xeon. These events can monitor various microarchitectural activities, including cache misses, TLB misses, pipeline stalls, etc. While these counts discover low level resource utilization, their effects are hard to quantify since the Xeon's long pipeline overlaps many of these events' occurrences. We examine cycles per instruction and track various causes of these cycles spent. At a high level, CPU cycles can be modeled by three factors: cycles required to graduate the given instruction, instruction related stalls caused by unavailable instructions or data, and stalls due to pipeline resource limits. For instruction-related stalls caused by cache or TLB misses, we are able to calculate the individual numbers of cycles for each event type based on the known miss penalties. For pipeline resources, the Xeon only provides the number of stalls caused by allocator buffer shortages. Stalls from other sources, such as lack of other buffers or decode-execute interlocks, are not available. Some major Xeon performance events used in this thesis and their names in Oprofile are listed in Appendix A. More detailed description of performance events can be found in [35] and [36].

We next describe a number of important metrics from a microarchitectural standpoint. We show measurements on our base 3.06 GHz processors since all of our processors have exactly the same resources such as cache size, TLB lines, buffers, etc., and we do not observe radical differences between them for the metrics we present here.

- **Cycles per instruction (CPI).** We measure the number of non-halted cycles and the number of instructions retired to calculate CPI. Since the Xeon decodes each instruction into multiple micro-ops (μops), we report CPI or CPμ where appropriate. The ratio of μops to instructions in our application ranges between 1.75 - 1.95. Figure 4.8 shows cycles per μop on each logical processor base for all of the servers. Network servers demonstrate much higher CPμ than other workloads, with the minimum value of 2.15, while the Xeon's optimal CPμ is 0.33, with three μops graduating per cycle.

While CPI is commonly used for indicating processor execution quality or efficiency, it is not a perfect metric for some parts of our study. Because of the varying code bases, the number of instructions used to deliver a single byte of content also differs. For this

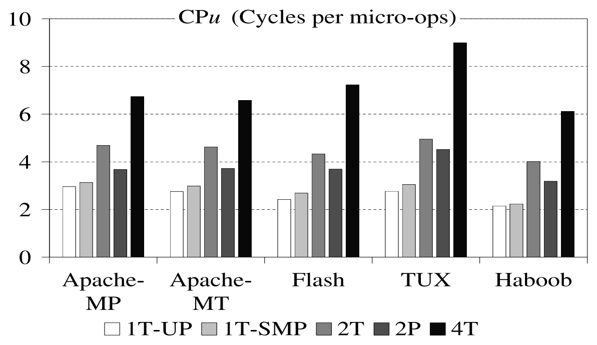


Figure 4.8: Cycles per micro-op (CPμ)

	Apache-MP	Apache-MT	Flash	Tux	Haboob
μPB	12.5	13.0	5.7	6.0	20.7
IPB	6.8	7.1	3.2	3.4	10.7

Table 4.4: Average Instructions and μops per byte for all servers

reason, we may also report counts in terms of application-level bytes transferred, shown in Table 4.4 as instructions and μops per byte (IPB and μPB). We discuss some of the event results below.

• **Cache behavior.** In SMTs, the multiple contexts share all of the cache resources. This sharing may cause extra cache pressure because of conflicts, but may also reinforce each other. By comparing miss rates, we are able to detect whether cache conflicts or reinforcement dominates. Figures 4.9, 4.10, and 4.11 show miss rates for the L1 instruction (trace) cache, the L1 data cache, and the combined L2 cache, respectively.

When SMT is enabled (in 2T and 4T), both the L1 instruction and data caches show significantly higher miss rates, indicating extra pressure, but the L2 miss rates improve, indicating benefits from sharing. In comparing Apache-MT to Apache-MP, we do see some reduction in the 4T L1 miss rate, but the miss rate is still higher than the 2P cases. Thus, while the multithreaded code helps reduce the pressure, the SMT ICache pressure is still significant. The L2 miss rate drops in all cases when SMT is enabled, indicating

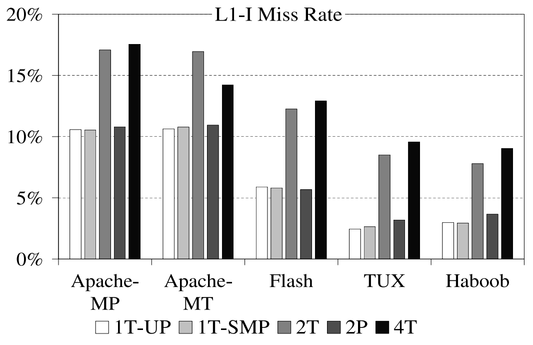


Figure 4.9: L1 instruction cache (Trace Cache) miss rate

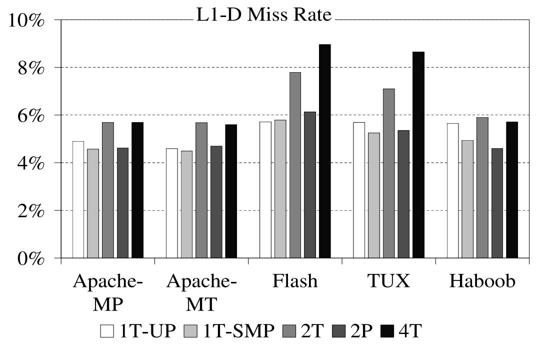


Figure 4.10: L1 data cache miss rate

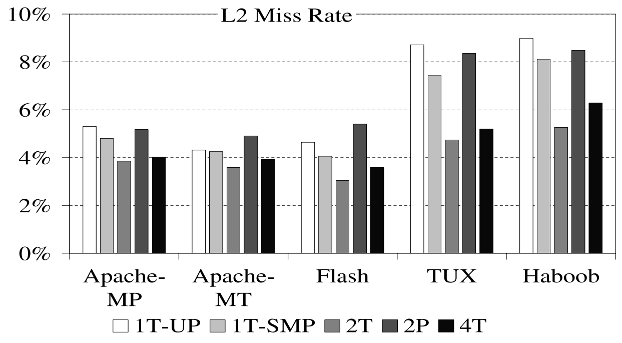


Figure 4.11: L2 cache miss rate, including both instruction and data

that the two contexts are reinforcing each other. The relatively high L2 miss rate for TUX is due to its lower L1 ICache miss rate – in absolute terms, TUX has a lower number of L2 accesses. The interactions on CPI are complex – the improved L2 miss rates can reduce the impact of main memory, but the much worse L1 miss rates can inflate the impact of L2 access times. We show the breakdowns later when calculating overall CPI values.

- **TLB misses.** In the current Xeon processor, the Instruction Translation Lookaside Buffer (ITLB) is duplicated and the shared DTLB is tagged with each logical processor's ID. Enabling SMT drops the ITLB miss rate (shown in Figure 4.12) while increasing the DTLB miss rate (shown in Figure 4.13). The DTLB miss rate is expected, since the threads may be operating in different regions of the code. We believe the drop in

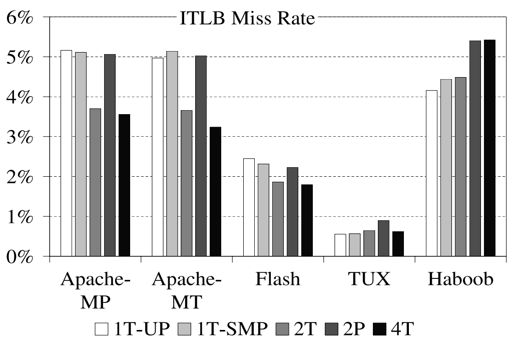


Figure 4.12: Instruction TLB miss rate

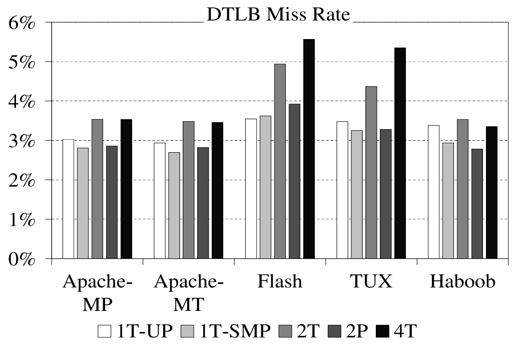


Figure 4.13: Data TLB miss rate

ITLB stems from the interrupt handling code executing only on the first logical processor, effectively halving its ITLB footprint.

• **Mispredicted branches.** Branches comprise 15% - 17% of instructions in our applications. Each mispredicted branch has a 20 cycle penalty. Even though all of the five servers show 50% higher misprediction rates with SMT, the overall cost is not significant compared to cache misses, as we show in the breakdowns later.

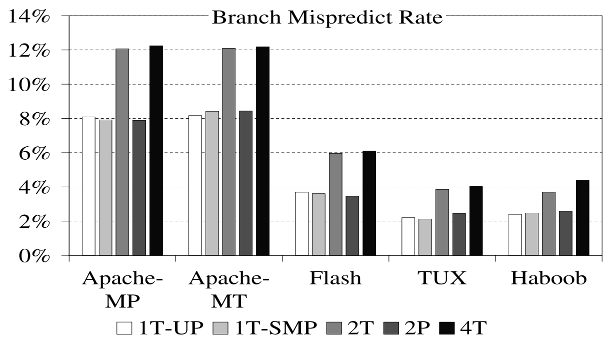


Figure 4.14: Branch misprediction rate

• **Instruction delivery stalls.** The cache misses and mispredicted branches result in instruction delivery stalls. This event measures the number of cycles halted when there are no instructions ready to issue. Figure 4.15 shows the average cycles stalled for each byte delivered. For each server, we observe a steady increase from 1T-UP to 4T, suggesting that with more hardware contexts, the number of cycles spent stalled increases.

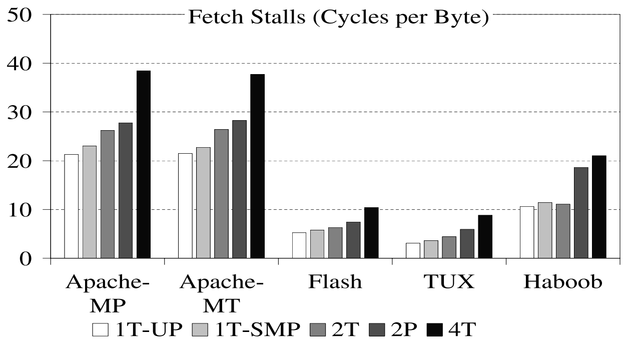


Figure 4.15: Trace delivery engine stalls

• **Resource Stalls.** While instruction delivery stalls happen in the front-end, stalls may also occur during pipeline execution stages. This event measures the occurrence of stalls in the allocator caused by store buffer restrictions. In the Xeon, buffers between major pipeline stages are partitioned when SMT is enabled. Figure 4.16 shows cycles stalled per byte due to lack of the store buffer in the allocator. Enabling SMT exhibits a doubling of the number of stall cycles for each byte transferred. Unfortunately, stalls due to other buffer conflicts, such as the renaming buffer, are not available on existing performance-monitoring counters. We expect similar pressure is also seen in other buffers.

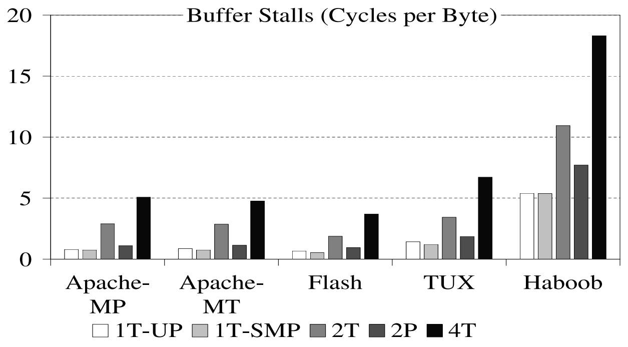


Figure 4.16: Stalls due to lack of store buffers

• **Pipeline clears.** Due to the Xeon's design, there are conditions in which all non-retiring stages of the pipeline need to be cancelled. This event measures the number of these flushes. When this happens, all of the execution resources are idle while the clear occurs. Figure 4.17 shows the average number of pipeline clears per byte of content. The SMT rate is a factor of 4 higher, suggesting that pipeline clears caused by one thread can affect other threads executing simultaneously. Profiling on this event indicates that more than 70% are caused by interrupts. Haboob's high clear rate in 4T mode may be responsible for some of its performance degradation.

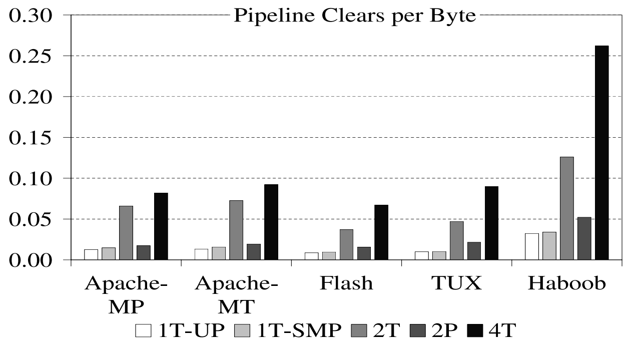


Figure 4.17: # of pipeline clears per byte

• **64K aliasing conflicts.** This event occurs when the address of a load or store conflicts with another reference which is in progress. When this happens, the second reference cannot begin until the first one is evicted from the cache. This type of conflict exists in the first-level cache and may incur significant penalties for loads that alias to preceding stores. The number of conflicts per byte is shown in Figure 4.18. All of the servers show fairly high number of conflicts, suggesting an effective direction for further optimization.

• **Putting cycles together.** We estimate the aggregated cycles per instruction of these negative events and compare them to the measured CPI. While it is possible to estimate the penalty of each event, some have aggregated effects and thus are not included. Figure 4.19 shows breakdowns of non-overlapped CPIs calculated from eight events, with

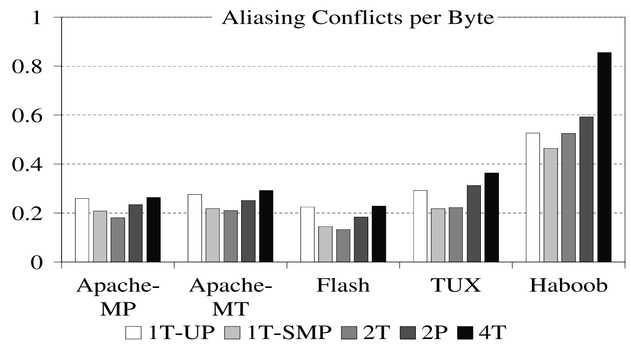


Figure 4.18: # of aliasing conflicts per byte

measured CPI shown as dashes. The breakdowns indicate that L1 and L2 misses are responsible for most of the cycles consumed. Pipeline clears and buffer stalls also have a significant portion when SMT is enabled, as shown in Flash, Tux and Haboob's 2T and 4T cases. Other events such as TLB misses and mispredicted branches are not major factors in our workloads.

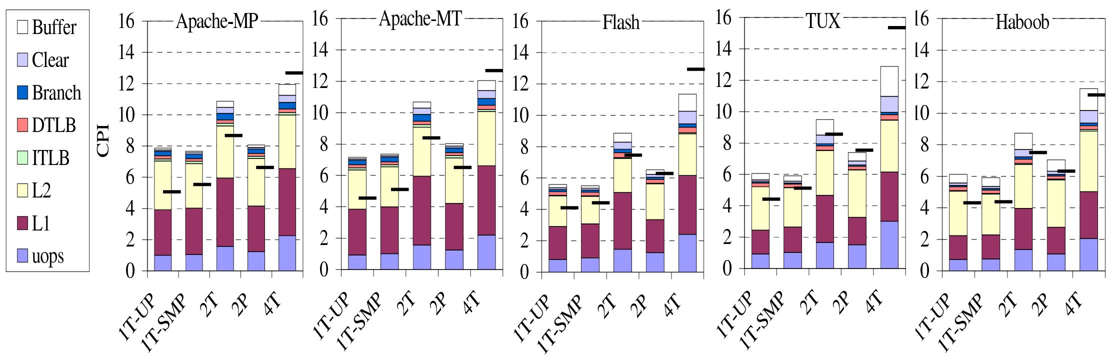


Figure 4.19: Non-overlapped CPI accumulated by cache miss, TLB miss, mispredicted branches and pipeline clear. Labels shown here such as L1, L2 etc. are misses, and components in each bar from top to bottom are in the same order as in the legend. Measured CPIs are shown as small dashes.

Our microarchitectural analysis provides quantitative explanations of the observed performance and discovers a number of SMT resource bottlenecks. Quantifying performance change based on processor events for our-of-order superscalar processors is not our goal, nor do we think it is feasible. However, by examining the aggregated CPI and measured CPI, we can estimate the pipeline overlapping if the measured CPI is lower than the calculated value, and how many cycles are taken by other sources if measured CPI is higher. With this microscopic information and observed performance improvement, we can compare to similar studies using simulation which we describe in the next section.

4.5 Evaluating the Simulations

Our measurements present a much less optimistic assessment of SMT performance benefits than many of the simulation-based studies – most of our gains are in the 5%-15% range for 2 threads, while the simulations show speedups in the 200%-400% range for 4-8 threads. Intuitively, the number of threads might be the cause of this speedup gap. However, studies also show that the first few threads usually exhibit more performance gain than the rest [91]. Thus, the simulated speedup for 2 threads would be in the range of 70-100%, which is still much higher than what we observe. While none of the published simulations modeled the Xeon, the significant disparity in the gains warrants analyzing their cause. We have not found any simulations specifically regarding multiprocessor systems, although such systems are popular in the network server market. None of the simulations appear to have considered the cost of using an SMP-enabled kernel instead of a uniprocessor kernel, and we have shown this cost to be significant. However, we believe the other significant differences are hardware-related, which we discuss below.

The most prominent area of difference between the Xeon and the simulated processors is the structure of the memory hierarchy, and the associated latencies. The Xeon has an 8KB L1 data cache and a 12K μops trace cache (TC), which is equivalent to an 8KB - 12KB conventional instruction cache. Detailed hardware parameters and latencies for

Type	sim	sim	sim	Xeon	Xeon
Year	1996	2000	2003	2003	2004
Clock rate (Mhz)	600*	800*	800*	2000	3060
# Contexts	8	8	8	2	2
# Stages	9	9	9	20	30
L1 ICache (KB)	32	128	64	8*	8*
L1 DCache (KB)	32	128	64	8	8
L2 size (KB)	256	16384	16384	512	512
L2 cycles	6	20	20	18	18
L3 size (KB)	2048	-	-	-	1024
L3 cycles	12	-	-	-	46
Memory (cycles)	62	90	90	225	344

Table 4.5: Processor parameters in simulation and current products. Values marked with an asterisk are approximate or derived.

our experimental platform are presented in Table 4.5. The 1996 study is a proposal for practical SMT processors [91], while the 2000 paper examines SMT OS and Web server performance [73], and the 2003 one examines SMT search engine performance [52]. The processor models were derived from Alpha, and have shorter pipelines and slower clock speeds than modern processors. While the 1996 design had caches that are comparable to what is available today, the others are much more aggressive than what is currently available.

The issue of cache size is significant, because of its direct impact on processor cycle time. Larger caches slow access times, and the Xeon's L1 caches are small in order to support its high clock frequencies. For comparison, if we triple the clock frequencies, stages, and main memory latencies for the 2000 and 2003 studies, then those values are in line with the current Xeons, but the L1 cache sizes are 8-16 times higher, and the L2 cache size is 32 times larger. If we assume the simulated L2 caches are really L3, then they are more than twice as fast as the Xeon's L3 latencies, while still being 4 times larger than the L3 caches of any Xeon in the market at the time this study was performed. Even if we compare with high-end processors, the simulated processors are still aggressive. For example, the IBM POWER5 has a 64KB instruction and 32 KB data L1 cache, 1.9MB L2 cache, 36MB of shared L3 cache per 2 processors, 2 SMT contexts, a 16-stage pipeline, and operates at 1.5 GHz [40]. Compare to the scaled version of the simulated processors, it has one-fourth the SMT contexts, and operates at half of the clock speed. We conclude that not only are the simulated memory hierarchies more aggressive than what current hardware can support, but the memory latencies are also much faster than what might be reasonably expected from their size. Given the sensitivity to cache sizes and memory speeds we have seen in our evaluations, it is not surprising that the simulations yield more optimistic speedup predictions.

While these differences indicate that SMT in the Xeon may not yield significant benefit on the workloads we studied, it does not imply that SMT in general is not useful. If a company were to design a processor specifically suitable for SMT, it may choose a larger number of contexts and larger caches while consciously sacrificing cycle time. Such a system may have poor performance for single-threaded applications, but would be suitable for highly-parallel tasks. Sun Microsystems has discussed their upcoming "Niagara" processor, which has 8 cores with 4 contexts per core [42]. The projected performance of this processor is about 15 times the performance of today's current processors, while a dual-core UltraSparc V microprocessor targeted for the same timeframe was expected to have 5 times the performance of today's processors. Using these numbers, each context on Niagara will perform at one-fifth the performance of one UltraSparc V context. This approach is similar to the Denelcor HEP [68] and the Tera MTA [83] which were designed for high throughput instead of high single-thread performance. Whether this approach will be more successful for a higher-volume processor remains an open question.

	Simulation		Measurement	
# Contexts	8		2	
Speedup	4-fold		5-15%	
	SMT	ST	SMT	ST
IPC	5.6	2.6	0.43	0.33
Branch Mispredict (%)	9.3	5.0	12.0	8.0
L1-I miss (%)	2.0	1.3	17.1	10.5
L1-D miss (%)	3.6	0.5	5.7	4.7
L2 miss (%)	1.4	1.8	3.9	5.1
ITLB miss (%)	0.0	0.0	3.7	5.1
DTLB miss (%)	0.6	0.05	3.5	2.9

Table 4.6: Results comparison between related simulation work and our measurements. We use ST (Single Threaded) to indicate the non-SMT performance.

In broader terms, though, the simulations identified a number of trends that we can confirm via our evaluation. Table 4.6 compares our measured results versus the simulation study of the same workload [73]. While the magnitude of the values between the simulated and actual processor is large due to the differences in cache sizes, etc., the direction of change is the same for each metric.

4.6 SMT on SPECweb99 benchmark

While static Web workloads are useful to compare with previous studies, dynamic content is an important part of current Web traffic, and is captured in the SPECweb99 benchmark. In order to compare with results discussed in previous sections, we run the full benchmark but also limit the data set size to 500 MB.

SPECweb99 introduces changes to both the workload and methodology of the SPECweb96 benchmark. The benchmark consists of 70% static and 30% dynamic requests. The dynamic requests attempt to model commercial Web servers performing ad rotation, customization, etc., and require some computation. Rather than reporting rates in requests

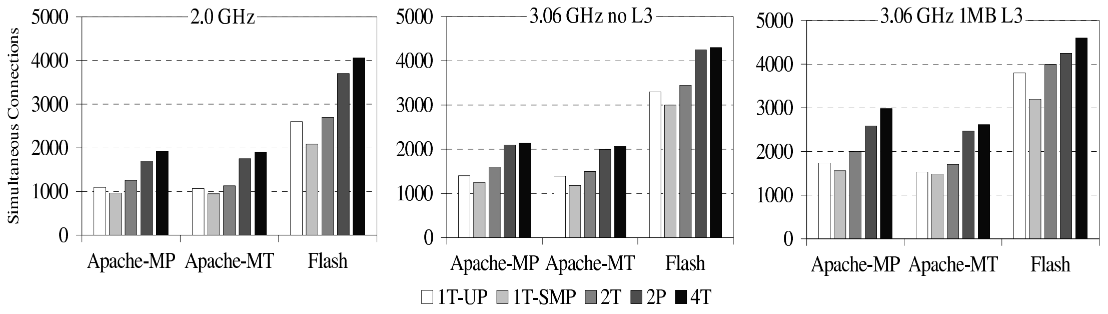


Figure 4.20: SPECweb99 scores of three servers. The metric is number of simultaneous connections.

per second, SPECweb99 reports the number of simultaneous connections the server can handle while meeting a specified latency requirement.

The dynamic portion of SPECweb99 consists of a specification which must be implemented for each server. Because we do not have versions of this specification for all of our servers, we can only evaluate it for three of our five systems. We run Flash as well as both versions of Apache on the three hardware configurations. While TUX has SPECweb99 scores, we have been unable to get the dynamic content support to work on the free version of Linux with SMT support. We are investigating its performance on the Red Hat Enterprise distribution since it is more stable and is the distribution for which most TUX results are reported. Haboob is not included because we did not find its dynamic API for SPECweb99. Since we focus on the differences stemming from SMT, we are not overly concerned about the missing servers. For Apache, we use the mod_specweb99 module which is available on the Apache Web site. Similarly, for Flash, we use its built-in SPECweb99 module that handles dynamic requests.

Figure 4.20 shows the SPECweb99 scores of the three servers on the three different hardware configurations, and Figure 4.21 calculates the speedups. The trends are generally consistent with what we observed in Section 4.3, but some interesting differences

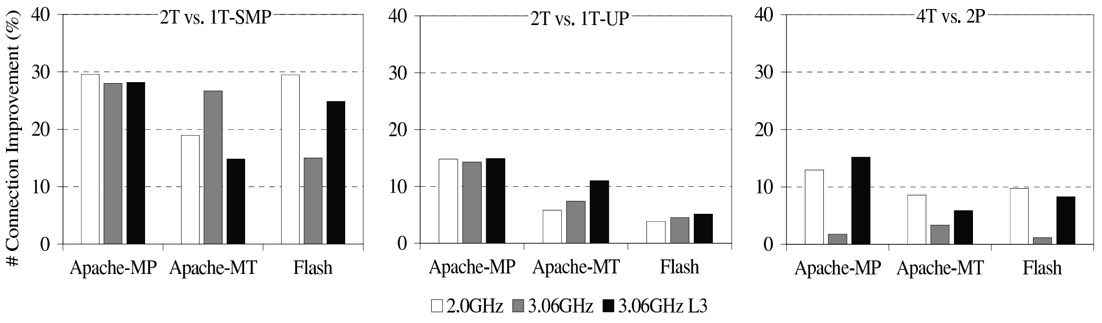


Figure 4.21: SMT speedups on SPECweb99 scores for the three servers.

emerge. On uniprocessor systems, SMT has a speedup of 4% to 15% when comparing to the uniprocessor kernel, and a speedup of 15% to 30% when comparing to the SMP kernel. On dual-processor systems, the improvements range from 1% to 15%. In comparing Figure 4.21 with Figures 4.5 and 4.6, we see some interesting differences. The gains for Apache on SPECweb99 are almost always worse than those on the static-only test. Flash's gains are worse at one processor, but better at two processors. The additional computation in SPECweb99 appears to help utilize the processor better when the memory system is very taxed, but seems to be causing more imbalance in Apache, which already does more computation than Flash.

The most noticeable differences are the comparison of 4T with 2P – previously, the 4T results for Flash showed degradation for the 3.06 GHz processor. On SPECweb99, this degradation disappears, and the relative improvement on the other two processors is larger. We believe that the dissimilar behavior between the static content and dynamic content allows better use of idle functional units. The differences are less apparent on Apache, which already was performing more processing per request than Flash.

This result also leads to another general observation. While SMT is intended to hide memory reference latency by overlapping the use of idle processor resources, application threads having similar resource utilization characteristics such as Web servers serving static content may have little to overlap.The difference between the effects of SMT on SPECweb96 and SPECweb99 is the compute-intensive dynamic content. This requires work for the processor than simple pointer-chasing, so it is possible that the dynamic content processing can be run during memory fetch stalls. Using dissimilar workloads to achieve better SMT utilization has been explored for the CINT (integer component of SPEC CPU) benchmark programs [87], but not for network server software. Even with this mix of programs, the benefits we see in the network server environment are lower than in the CPU-only tests.

4.7 Discussion

In the previous sections, we have shown that memory bottleneck prevents network servers from realizing significant benefits on SMT. Another interesting area for SMT systems is CPU-intensive applications, particularly in multiprogrammed systems. These programs may be more cache friendly, and may have very different resource utilization characteristics, so running two different CPU-intensive programs can potentially improve processor utilization. However, even this attempt of pairing mutually-beneficial programs has its caveats, and the associated problems have led a number of researchers to explore optimizing the scheduler for SMT [21, 84].

Rather than recreate this research, we use previously-published results but analyze them differently. In particular, two groups have examined the pairwise interference/benefit between the 26 programs in the SPEC CPU2000 [20, 90] benchmark suite. We use these as input to our analysis, which basically asks the following question: if all 26 program pairs ran for the same time, and you had to schedule them for the most benefit, how

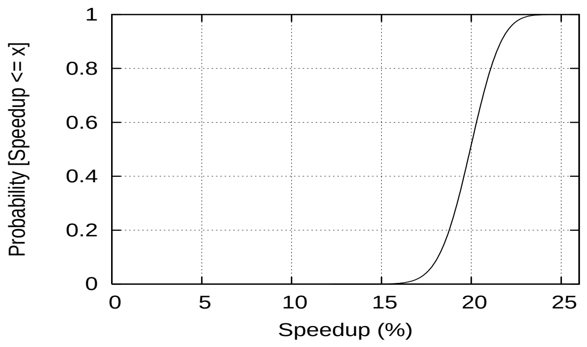


Figure 4.22: Speedup CDF of SPEC CPU2000 scheduling: Min is 12%, Max is 26%, and Mean is 20%.

would you do it? Since the possible permutations are far too large to analyze, we use randomized schedules to simplify the analysis.

We create schedules by randomly pairing program and calculating the schedule's total runtime using the pairwise interference measurements. By repeating this process for a large number of trials, we can determine the range of scheduling benefit. The CDF of 10 million trials, shown in Figure 4.22, indicates that the range of resulting speedups is fairly narrow, with a mean of 20% and an absolute maximum of 26%. A more typical high-end schedule achieves 23%.

So, even if we have perfect offline information, the ideal scheduler achieves only 3%-6% more speedup than a random scheduler. One may argue that a random scheduler could perform as poorly as only a 12% speedup. While true, the remedy is also simple. The scheduler could periodically re-randomize the pairings, bringing all schedules closer to the median as the number of randomizations increases. This scheduling policy is extremely simple, and performs almost as well as an ideal scheduler. For realistic schedulers, which will need to collect the pairwise data at runtime, the gap may be even narrower.

While we use randomized analysis, this argument is not about the law of large numbers – we assume that the ideal scheduler can achieve the best schedule, or something close to it. What we are merely observing is that given enough programs and the pairwise characteristics of the P4, the average speedup is simply not that bad. If people run very few simultaneous programs, then scheduling matters even less – there are very few choices for the scheduler to make. Some degenerate cases are possible, where there may be only two programs in the system, and they exhibit slowdown when run together. However, these cases are rare among the SPEC CPU2000 programs, and the ability to detect this scenario does not imply that a more complicated scheduler is needed.

4.8 Related Work

In this section we discuss related work not already covered. When investigating SMT performance, operating systems were not included in simulations until Redstone *et al*. [73] ported the SMTSIM simulator [92] into the SimOS framework [75]. They discovered that although ignoring operating systems behavior may not result in misleading predictions for SPEC CINT, it has significant impact on evaluation of server applications such as Apache. McDowell et al. [52] used the same simulator and studied memory allocation and synchronization strategies for a search engine application. Similarly, many studies focus on user-level and compute-intensive applications, including SPEC CINT and CFP [85, 91, 92], parallel ray-tracing applications [32], SPLASH-2 benchmarks [49], MPEG-2 decompression [23, 82], and other scientific application workloads [44, 84]. Lo *et al*. [48] analyzed SMT performance with database workloads, which spend 70% of their time in user space.

Our approach differs from previous performance evaluations in several ways. While direct measurement on real hardware gives accurate results and does not need model validation, as is required by simulation, it is limited by the available hardware configurations. By making small changes to the hardware and exploring kernel options, we obtain a reasonable space for comparison. More importantly, compared to the work which studies server applications, our evaluation has a broader range of server software, OS support and hardware configurations. Our results on uniprocessor kernels and dual-processor systems discover new SMT performance characteristics. In contrast to other works, we focus on server workloads since it is one of the biggest markets for SMT-enabled processors.

Performance evaluation and characterization on currently available SMT-enabled processors is still an ongoing research area. Tuck and Tullsen [90] evaluated the implementation and effectiveness of SMT in a Pentium 4 processor, particularly in the context of prior published research in SMT. They measured SPEC CPU2000 and other parallel benchmarks, and concluded that this processor implementation matches the promise of the published SMT research. Bulpin and Pratt [20] extended Tuck *et al.*'s evaluation by comparing an SMT system to a comparable SMP system, but did not investigate SMT on SMP systems as we do in this thesis. Chen *et al.* [23] also only evaluated performance of SMT and SMP individually. Vianney [94] measured a single Xeon processor and reported that Hyper-Threading on the Linux kernel can improve throughput of a multi-threaded application (namely, chat [47]) as much as 60%. Our study not only differs in target testbed and workloads, but also provides low level microarchitectural characteristics to reveal detailed resource utilization pertinent to SMT.

SMT's microarchitectural performance is always one of the main concerns in SMT design. In addition to work discussed earlier in this section, Grunwald and Ghiasi [30] examined the Xeon processor and discovered a possible microarchitectural denial of service attack for the SMT processor. Snavely *et al*. [84, 85] observed symbiotic features in SMT architectures and proposed a special scheduler to exploit it. Raasch and Reinhardt [69] studied the impact of resource partitioning on SMT processors. Our microarchitectural analysis using the performance counters focuses on the comparison between when SMT is enabled and disabled, instead of evaluating the performance of different SMT design options.

Performance analysis using hardware provided event counters has been an effective approach in previous studies. Bhandarkar *et al*. [14] and Keeton *et al*. [41] characterized performance of Pentium Pro systems and studied latency components of the CPI. More recently, Blackburn *et al*. [15] used some of the Pentium 4 performance counters to study impact of garbage collection. Given the complexity of Xeon microarchitecture, interpreting the performance-monitoring events on these systems is more difficult than with previous Pentium family processors or RISC processors. Moreover, we are unaware of any non-simulation work in this area that provides the breadth of event coverage that we do in this thesis.

Chapter 5

Conclusion & Future Work

5.1 Conclusion

This thesis presents network server performance analysis and improvement at various levels. At the kernel level, we develop a novel kernel profiling tool to provide applications with fine-grained kernel information at low cost, and contributed a set of kernel patches to improve networked file transfer. At the application level, we redesign a popular research server, Flash, and the widely-used Apache server, improving Flash's SPECweb99 score by a factor of four and reducing response time by one to two orders of magnitude on both servers. Using these servers as well as others, we then investigate the architectural aspects of server performance, conducting detailed analysis of delivered SMT systems. Some of the results have led to a better understanding of processor bottlenecks and server optimization.

At the beginning of this thesis, we present a study of server performance and the interactions with operating systems, and describes the design, implementation and evaluation of DeBox, an effective approach to provide more OS transparency, by exposing system call performance as a first-class result via in-band channels. DeBox provides direct performance feedback from the kernel on a per-call basis, enabling programmers to diagnose kernel and user interactions correlated with user-level events.

The case study using DeBox on the Flash Web Server demonstrates the power of this approach. Addressing the problematic interactions and optimization opportunities discovered using DeBox improves our experimental results an overall factor of four in SPECweb99 score, despite having a data set size nearly three times as large as our physical memory. Furthermore, our latency analysis demonstrates gains between a factor of 4 to 47 under various conditions. Further results show that fixing the bottlenecks identified using DeBox also mitigates most of the negative impact from excess parallelism in application design.

Then the thesis examines server latency under load and traces the root cause of server-induced latency. By experimenting with workloads of various sizes, we determine that when disk accesses occur, both mean and median latencies increase, though the median should be unaffected. We trace the roots of this problem to head-of-line blocking within filesystem-related kernel queues. This behavior, in turn, causes batching and burstiness, which has little impact on throughput, but severely degrades latency. By examining individual request latencies, we find that this blocking gives rise to a phenomenon we call *service inversion*, where requests are served unfairly.

By addressing the blocking issues both 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. This work also improves on our fundamental understanding of the interactions between the filesystem, application, and workloads. The results suggest that most server-induced latency is tied to blocking effects, rather than queuing. 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.

Finally, the thesis provides a performance analysis of simultaneous multithreading for server applications, using five software implementations and three hardware platforms. We find that the performance benefits of SMT are much more modest when compared to the uniprocessor kernel, suggesting non-negligible amounts of OS overhead when supporting SMT. This cost was mostly ignored in previous studies. Our evaluation in dual-processor configurations indicates that the benefits of SMT are harder to achieve in these systems, unless memory reference latency is shorter or a large L3 cache is used, a finding that may aid SMT design and purchasing.

Our microarchitectural analysis provides quantitative explanations of the observed performance and discovers a number of SMT resource bottlenecks. Using this information, future processor designers can understand how to better serve this important class of applications. With this detailed, low-level information and observed performance improvement, we are able to compare our results to similar studies performed using simulation. We believe that simulation correctly predicts the direction of change for processor resources, but yields much more optimistic estimates of resource contention and overall speedup.

5.2 Future Work

My future research interests include computer architecture, operating systems and server applications. With the rapid development of computer architecture, it is a persistent research topic for software designers to identify necessary changes to cope with new platforms. One of our original goals of the SMT study was to improve server performance for SMT processors. We have discussed that it is unlikely to achieve high performance by optimizing server application on the current Intel Xeon platforms. However, other architectures, such as the IBM POWER5 and other emerging dual-core chips, may present very different resource usage characteristics. We are investigating optimization opportunities using performance-counter based profiling approaches. By focusing on bottleneck components, we identify the most expensive activities and redesign the responsible software invocations. We intend to term this approach as *hardware-aware profile-guided optimizations*.

In this thesis, we focus on performance issues on traditional operating systems. With the increasing popularity of virtual machines, we believe that performance analysis and optimization in virtualized environment present new opportunities for researchers. Similar to operating systems, virtual machines also isolate hardware resource from being accessed directly by up-level applications. However, as what we have shown in this thesis, it is critical for applications to obtain performance knowledge from underlying layers. We plan to provide effective approaches under virtualized environment to aid in high-performance application design and optimization. Specifically, we would like to examine what information is critical to export giving a combination of virtual machine monitor, application and workloads. Then we study the effectiveness of existing communication channels between the different layers. Particularly, we are interested in knowing whether any in-band channel such as system call interface can be enhanced in providing feedback from the isolated layers. Finally, we would also like to investigate the necessity and possibility of providing processor-specific performance counters to each virtual machine.

Appendix A

Xeon Performance Events

Event Description	Oprofile Event Name	Oprofile unit mask
Processor active cycles	GLOBAL_POWER_EVENTS	0x01
Total instructions retired	INSTR_RETIRED	0x0F
Total Micro-ops retired	UOPS_RETIRED	0x03
Non-bogus instructions retired	INSTR_RETIRED	0x03
Non-bogus Micro-ops retired	UOPS_RETIRED	0x01
Branches retired	BRANCH_RETIRED	0x0F
Mispredicted branches	MISPRED_BRANCH_RETIRED	0x01
Memory loads*	FRONT_END_EVENT & UOP_TYPE	0x01 0x01
Memory stores*	FRONT_END_EVENT & UOP_TYPE	0x01 0x02
Requests from Branch Prediction unit	BPU_FETCH_REQUEST	0x01
Instruction TLB misses	ITLB_REFERENCE	0x02
Instruction TLB references	ITLB_REFERENCE	0x03
Data TLB misses*	DTLB_ALL_MISS_RETIRED	0x01
L1 load misses*	L1_LD_MISS_RETIRED	0x01
L2 cache read hits	BSQ_CACHE_REFERENCE	0x07
L2 cache read misses	BSQ_CACHE_REFERENCE	0x100
L3 cache read hits	BSQ_CACHE_REFERENCE	0x38
L3 cache read misses	BSQ_CACHE_REFERENCE	0x200
Micro-ops written in pipeline	UOP_QUEUE_WRITES	0x07
Trace cache delivery	TC_DELIVER_MODE	0x4F
Front-side bus busy	FSB_DATA_ACTIVITY	0x03
Pipeline flushes	MACHINE_CLEAR	0x45
Stalls in pipeline buffer	RESOURCE_STALLS	0x20

Table A.1: Xeon performance events and their names/masks in Oprofile. Events with * are not supported by default in Oprofile version 0.x. Measuring these events requires a patch developed by the author [76]

Bibliography

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative tasking without manual stack management. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[2] W. Akkerman. strace. http://www.wi.leidenuniv.nl/ wichert/strace/.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, June 1997.

[4] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 1–14, Saint-Malo, France, Oct. 1997.

[5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

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

[7] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 43–56, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[8] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proc. of the 18th ACM Symp. on Operating System Principles*, pages 90–105, Bolton Landing, NY, Oct. 2003.

[9] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[10] 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.

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

[12] P. Benmowski. Hyper-Threading Linux. *LinuxWorld*, Aug. 2003.

[13] 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.

[14] D. Bhandarkar and J. Ding. Performance characterization of the Pentium Pro processor. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA '97)*, pages 288–298, Feb. 1997.

[15] S. Blackburn, P. Cheng, and K. McKinley. Myths and realities: The performance impact of garbage collection. In *Proc. of the SIGMETRICS '04*, June 2004.

[16] C. Blake and S. Bauer. Simple and general statistical profiling with pct. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[17] 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.

[18] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *ACM SIGMETRICS Conference*, pages 214–224, Seattle, WA, June 1997.

[19] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[20] J. Bulpin and I. Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD04)*, June 2004.

[21] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX 2005 Annual Tech, To appear*, Anaheim, CA, April 2005.

[22] N. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[23] Y.-K. Chen, E. Debes, R. Lienhart, M. Holliman, and M. Yeung. Evaluating and improving performance of multimedia applications on simultaneous multi-threading. In *9th Intl. Conf. on Parallel and Distributed Systems*, Dec. 2002.

[24] E. Cota-Robles and J. P. Held. A comparison of windows driver model latency performance on windows NT and windows 98. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 159–172, New Orleans, LA, Feb. 1999.

[25] 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.

[26] 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.

[27] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–18, Sept. 1997.

[28] K. Elmeleegy, A. Chanda, A. Cox, and W. Zwaenepoel. A portable kernel abstraction for low-overhead ephemeral mapping management. In *USENIX 2005 Annual Technical Conference*, Anaheim, CA, April 2005.

[29] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, Massachusetts, June 1982.

[30] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In *Proc. of the 35th Intl. Symp. on Microarchitecture*, pages 409–418. IEEE Computer Society Press, 2002.

[31] 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.

[32] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proc. of the 19th Annual International Symp. on Computer Architecture*, pages 136–145. ACM Press, 1992.

[33] 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.

[34] Intel. Vtune Performance Analyzers Homepage. http://developer.intel.com/ software/products/ vtune/index.htm.

[35] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide. http://developer.intel.com/ design/pentium4/ manuals/253668.htm.

[36] Intel Corporation. Intel Pentium 4 and Intel Xeon Processor optimization reference manual. http://developer.intel.com/ design/pentium4/ manuals/index_new.htm.

[37] 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.

[38] M. B. Jones and J. Regehr. The problems you're having may not be the problems you think you're having: Results from a latency study of windows nt. In *7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[39] 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.

[40] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March 2004.

[41] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance charaterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proc. of the 25th Annual International Symp. on Computer Architecture*, pages 15–26, 1998.

[42] P. Kongetira. A 32-way Multithreaded SPARC(R) Processor. In *The 16th Symposium on High Performance Chips (HOTCHIPS' 16)*, 2004.

[43] G. Kuenning. Kitrace—precise interactive measurement of operating systems kernels. *SOFTWARE–PRACTICE AND EXPERIENCE*, 1(1):1–21, 1994.

[44] H. Kwak, B. Lee, A. Hurson, S.-H. Yoon, and W.-J. Hahn. Effects of multithreading on cache performance. *IEEE Trans. Comput.*, 48(2):176–184, 1999.

[45] 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.

[46] 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.

[47] Linux Benchmark Suite Homepage. A GPL'd chat room benchmark. http://lbs.sourceforge.net/.

[48] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. of the 25th Annual International Symp. on Computer Architecture*, pages 39–50, 1998.

[49] J. Lo, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[50] 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 (USITS'97)*, Boulder, CO, Oct. 1999.

[51] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, Feb. 2002.

[52] L. McDowell, S. Eggers, and S. Gribble. Improving server software support for simultaneous multithreaded processors. In *Proc. of the 2003 Symp. on Principles of Parallel Programming (PPoPP'03)*, pages 37–48, San Diego, CA, June 2003.

[53] 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.

[54] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[55] Mindcraft. Webstone: The benchmark for web servers. http://www.mindcraft.com/webstone/.

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

[57] I. Molnar. Method and apparatus for atomic file look-up. United States Patent Application #20020059330, May 16, 2002.

[58] E. Nahum. Deconstructing SPECweb99. In *7th International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, CO, Aug. 2002.

[59] 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.

[60] Open Market. FastCGI. http://www.fastcgi.com/.

[61] OProfile. http://oprofile.sourceforge.net/.

[62] 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.

[63] 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.

[64] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.

[65] 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.

[66] L. K. Puthiyedath, E. Cota-Robles, J. Keys, and J. P. H. Anil Aggarwal. The design and implementation of the intel real-time performance analyzer. In *Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, Sept. 2002.

[67] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[68] R. E. Hiromoto, O. M. Lubeck, and J. Moore. Experiences with the Denelcor HEP. In *Parallel Computing*, pages 197–206, 1984.

[69] S. Raasch and S. Reinhardt. The impact of resource partitioning on SMT processors. In *12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, Louisiana, Sept. 2003.

[70] 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.

[71] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[72] N. Ramanathan, E. Kohler, L. Girod, and D. Estrin. Sympathy: A debugging system for sensor networks. In *29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, Nov. 2004.

[73] J. Redstone, S. Eggers, and H. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–256, 2000.

[74] T. Romer, G. V. D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *USENIX Windows NT Workshop*, pages 1–8, 1997.

[75] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[76] Y. Ruan. Oprofile patch for xeon/p4. http://www.cs.princeton.edu/ yruan/XeonSMT/ patch/.

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

[78] Y. Ruan and V. Pai. The origins of network server latency & the myth of connection scheduling (extended abstract). In *Proceedings of the 2004 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, New York, NY, June 2004.

[79] Y. Ruan, V. Pai, E. Nahum, and J. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *Proceedings of the 2005 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*, Banff, AB, Canada, June 2005.

[80] 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.

[81] 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.

[82] U. Sigmund and T. Ungerer. Memory hierarchy studies of multimedia-enhanced simultaneous multithreaded processors for mpec-2 video decompression. In *Workshop on MultiThreaded Execution, Architecture and Compilation*, January 2000.

[83] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the tera mta. In *Proc. of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–8, 1998.

[84] A. Snavely and D. Tullsen. Symbiotic job scheduling for a simultaneous multi-threaded processor. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244. ACM Press, 2000.

[85] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proc. of the SIGMETRICS'02*, pages 66–76, June 2002.

[86] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[87] Standard Performance Evaluation Corporation. http://www.spec.org/bench marks.html.

[88] Standard Performance Evaluation Corporation. SPEC Web Benchmarks. http://www.spec.org/web99/ http://www.spec.org/web96.

[89] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 117–130, New Orleans, LA, Feb. 1999.

[90] N. Tuck and D. Tullsen. Initial observations of a simultaneous multithreading processor. In *12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, Sept. 2003.

[91] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of the 23rd Annual International Symp. on Computer Architecture*, pages 191–202, 1996.

[92] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22nd Annual International Symp. on Computer Architecture*, 1995.

[93] TUX Web Server. http://www.tux.org/.

[94] D. Vianney. Hyper-Threading speeds Linux. *IBM developerWorks*, Jan. 2003.

[95] 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.

[96] 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.

[97] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[98] C. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 15–26, Saint-Malo France, Oct. 1997.