Techniques for Developing and Measuring High-Performance Web Servers over ATM Networks

James C. Hu[†], Sumedh Mungee, Douglas C. Schmidt
{jxh,sumedh,schmidt}@cs.wustl.edu
TEL: (314) 935-4215 FAX: (314) 935-7302
Campus Box 1045/Bryan 509
Washington University
One Brookings Drive
St. Louis, MO 63130, USA*

This paper appeared in the INFOCOM '98 conference, San Francisco, March/April 1998.

Abstract

High-performance Web servers are essential to meet the growing demands of the Internet and large-scale intranets. Satisfying these demands requires a thorough understanding of key factors affecting Web server performance. This paper presents empirical analysis illustrating how dynamic and static adaptivity can enhance Web server performance. Two research contributions support this conclusion.

First, the paper presents results from a comprehensive empirical study of Web servers (such as Apache, Netscape Enterprise, PHTTPD, Zeus, and JAWS) over high-speed ATM networks. This study illustrates their relative performance and precisely pinpoints the server design choices that cause performance bottlenecks. We found that once network and disk I/O overheads are reduced to negligible constant factors, the main determinants of Web server performance are its protocol processing path and concurrency strategy. Moreover, no single strategy performs optimally for all load conditions and traffic types.

Second, we describe the design techniques and optimizations used to develop JAWS, our high-performance, adaptive Web server. JAWS is an object-oriented Web server that was explicitly designed to alleviate the performance bottlenecks we identified in existing Web servers. It consistently outperforms all other Web servers over ATM networks. The performance optimizations used in JAWS include adaptive pre-spawned threading, fixed headers, cached date processing, and file caching. In addition, JAWS uses a novel software architecture that substantially improves its portability and flexibility, relative to other Web servers. Our empirical results illustrate that highly efficient communication software is not antithetical to highly flexible software.

*This work was funded in part by NSF grant NCR-9628218, Object Technologies International, Eastman Kodak, and Siemens MED.

1 Introduction

During the past two years, the volume of traffic on the World Wide Web (Web) has grown dramatically. Traffic increases are due largely to the proliferation of inexpensive and ubiquitous Web browsers (such as NCSA Mosaic, Netscape Navigator, and Internet Explorer). Likewise, Web protocols and browsers are increasingly applied to specialized computationally expensive tasks, such as image processing servers used by Siemens [8] and Kodak [18] and database search engines (*e.g.*, AltaVista and Lexis Nexis).

To keep pace with increasing demand, it is essential to develop high-performance Web servers. Therefore, the central themes of this paper are:

• **High-performance Web servers must be adaptive:** To achieve optimal performance, Web servers must adapt to various conditions, such as machine load and network congestion, the type of incoming requests, and the number of simultaneous connections. While it is always possible to improve performance with more expensive hardware or a faster OS, our objective is to produce the fastest Web server possible for a given hardware/OS platform configuration.

• **Standard Web server benchmarking suites are inadequate over high-speed networks:** Our experience measuring Web server performance on ATM networks reveals that existing benchmarking tools (such as WebSTONE and SPECWeb) designed for low-speed Ethernet networks are inadequate to capture key performance determinants on high-speed networks.

To address these issues, this paper describes an adaptive Web server framework and a Web server/ATM testbed designed to empirically determine (1) the scalability of Web servers under varying load conditions, (2) the performance impact of different server design and implementation strategies, and (3) the pros and cons of alternative Web server designs.

1.1 Web Server Performance

Web servers are often considered synonymous with HTTP servers and the HTTP 1.0 and 1.1 protocols are relatively straightforward. HTTP requests typically name a file, which the server locates and returns to the client requesting it. On the surface, therefore, Web servers appear to have few opportunities for optimization. This may lead to the conclusion that optimization efforts should be directed elsewhere (such as transport protocol optimizations [14], specialized hardware [5], and client-side caching [27, 15]).

However, empirical analysis in [8] and in Section 2 reveals that Web server performance problems are complex and the solution space is quite diverse. For instance, our experimental results show that a heavily accessed Apache Web server (the most popular server on the Web today [23]) is unable to maintain satisfactory performance on a dual-CPU 180 Mhz UltraSPARC 2 over a 155 Mbps ATM network, due largely to its choice of process-level concurrency. Other studies [12, 8] have shown that the relative performance of different server designs depend heavily on server load characteristics (such as the number of simultaneous connections and file size).

The explosive growth of the Web, coupled with the larger role servers play on the Web, places increasingly larger demands on servers [3]. In particular, the high loads that servers like the NASA Pathfinder Web site and AltaVista already encounter handling millions of requests per day will be confounded with the deployment of high-speed networks, such as ATM or Gigabit Ethernet. Therefore, it is critical to understand how to improve server performance and predictability.

Server performance is already a critical issue for the Internet [1] and is becoming more important as Web protocols are applied to performance-sensitive intranet applications. For instance, electronic imaging systems based on HTTP (such as Siemens MED or Kodak Picture Net) require servers to perform computationally-intensive image filtering operations (such as smoothing, dithering, and gamma correction). Likewise, database applications based on Web protocols (such as AltaVista Search by Digital or the Lexis Nexis) support complex queries that may generate a higher number of disk accesses than a typical Web server.

1.2 Adaptive Web Servers

This paper presents empirical results that illustrate that no single Web server configuration is optimal for all circumstances. Based on these results, we conclude that optimal Web server performance requires both *static* and *dynamic* adaptive behavior.

Static adaptivity allows a Web server to bind common operations to high-performance mechanisms provided by the native OS (*e.g.*, Windows NT 4.0 support for asynchronous I/O and network/file transfer). Programming a Web server to use generic OS interfaces (such as synchronous POSIX threading) is insufficient to provide maximal performance across OS platforms. Therefore, asynchronous I/O mechanisms in Windows NT and POSIX must be studied, compared, and tested against traditional concurrent server programming paradigms that utilize synchronous event demultiplexing and threading [8].

Dynamic adaptivity allows a Web server to alter its run-time behavior "on-the-fly." This is useful when external conditions have changed to the point where the initial configuration no longer provides optimal performance. Such situations have been observed in [8] and [11].

The remainder of this paper is organized as follows: Section 2 outlines our Web server/ATM benchmarking testbed and analyzes our benchmark results; Section 3 describes the OO design and performance of JAWS, our high-performance Web server; Section 4 summarizes the Web server optimization techniques identified by our empirical studies; Section 5 compares our research with related work; and Section 6 presents concluding remarks.

2 Web Server Performance over ATM

This section describes our experimental methodology, benchmarking and analysis tools, the results of our experiments, and our analysis of the results. To study the primary determinants of Web server performance, we selected five Web server implementations and analyzed their performance through a series of blackbox and whitebox benchmarking experiments. Our analysis of these results identified the following key determinants of Web server performance:

- **Filesystem access overhead costs are high:** Most distributed applications benefit from caching and Web servers are no exception. In general, Web servers that implement file caching strategies (such as Enterprise, Zeus, and JAWS) perform substantially better than those that did not (such as Apache).
- **Concurrency overhead is significant:** A large portion of non-I/O related Web server overhead is due to the Web server's concurrency strategy. Key overheads include synchronization, thread/process creation, and context switching. Therefore, it

is crucial to choose the right concurrency strategies to ensure optimal performance.

• **Protocol processing overhead can be expensive:** Although the HTTP/1.0 protocol is relatively simple, a naive implementation can introduce a substantial amount of overhead. For instance, the dynamic creation of response headers and the use of multiple `write` system calls significantly inhibits performance.

These and related performance issues are described in Section 4.

2.1 Web Server Test Suite

The servers chosen for our tests were Apache v1.1, PHTTPD v0.99.76, Java Server 1.0, Netscape Enterprise v2.01 and Zeus Server v1.0. The choice of servers for our study were based on two factors. The first was *variation in Web server design*, to gauge the performance impact of alternative approaches to concurrency, event dispatching, and filesystem access. The second was *published performance*, as reported by benchmark results published by Jigsaw [2] and NCSA [13], as well as information available from WebCompare [23].

Apache and PHTTPD are implemented in C. The Java Server is implemented in Java. Netscape Enterprise and Zeus servers are commercial servers with only binaries available. Apache utilizes process level concurrency, PHTTPD spawns a thread per request, as does the Java Server from Sun.[1] Although source code for Netscape Enterprise and Zeus is unavailable, their concurrency model can be inferred using tools like `truss` and `TNF`. Our analysis indicates that Netscape Enterprise uses a Thread Pool implementation, whereas Zeus uses a Process Pool implementation.

2.2 Web Server/ATM Testbed

2.2.1 Hardware and Software Platforms

We studied Web server performance by observing how the servers in our test suite performed on high-speed networks under heavy workloads. To accomplish this, we constructed a hardware and software testbed consisting of the Web server being tested, and multiple clients connected to it via a high-speed ATM switch [26], as shown in Figure 1.[2]

The experiments in this paper were conducted using a Bay Networks LattisCell 10114 ATM switch connected to four dual-processor UltraSPARC-2s running SunOS 5.5.1. The

[1] The Java Server actually performs Thread-per-Connection. However, our testbed is based on HTTP 1.0 and does not use "keep-alives" (*i.e.*, a new connection is established for every new request). Therefore, Java Server effectively performs Thread-per-Request.

[2] We also performed measurements over 10 Mbps Ethernet, but due to a lack of performance variance, we omitted the discussion from this paper.

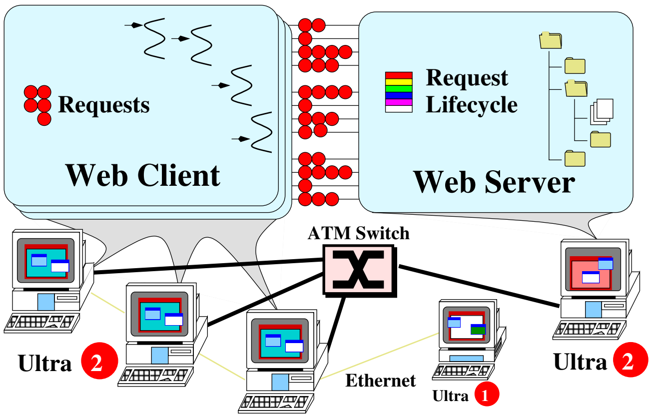

Figure 1: Web Server/ATM Testbed Environment

LattisCell 10114 is a 16 Port, OC3 155 Mbs/port switch. Each UltraSPARC-2 contains 2 168 MHz CPUs with a 1 Megabyte cache per-CPU, 256 Mbytes of RAM, and an ENI-155s-MF ATM adaptor card that supports 155 Megabits per-sec (Mbps) SONET multi-mode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card. This testbed is similar to the one used in [7].

2.2.2 Benchmarking Methodology

We used the WebSTONE [6] v2.0 benchmarking software to collect client- and server-side metrics. As described in Section 2.3, these metrics included *average server throughput*, *average client throughput*, *average number of connections-per-second*, and *average client latency*. The testbed comprised multiple concurrent *Web clients*, running on UNIX hosts depicted in Figure 1. Each Web client transmits a series of HTTP requests to download files from the server. The file access pattern used in the tests is shown in Table 1.

Document Size	Frequency
500 bytes	35%
5 Kbytes	50%
50 Kbytes	14%
5 Mbytes	1%

Table 1: File Access Patterns

This table represents actual load conditions on popular servers, based on a study of file access patterns conducted by SPEC [4].

We benchmarked each Web server on an UltraSPARC-2 host, while the Web clients ran on three other UltraSPARC-

2s. Web clients are controlled by a central *Webmaster*, which starts the Web clients simultaneously. The Webmaster also collects and combines the measurements made by individual clients. Since the Webmaster is less performance critical, it ran on an UltraSPARC-1 connected to the testbed machines with 10 Mbps Ethernet. The UltraSPARC-1 contained a 167 MHz CPU with 1 Megabyte cache and 128 MBytes of RAM.

2.2.3 Web Server Performance Analysis Techniques

We used two techniques to analyze the performance of Web servers: *blackbox* and *whitebox* benchmarks. The *blackbox* tests measure externally visible Web server performance metrics (such as throughput and average response time seen by clients). We accomplished this by controlling the Web clients to vary the load on the server (*i.e.*, the number of simultaneous connections). These clients computed several blackbox metrics, as explained in Section 2.3.

To precisely pinpoint the *source* of performance bottlenecks, we employed whitebox benchmarks. This involved the use of profiling tools, including the UNIX `truss(1)` tool, `TNF` [24], and `Quantify` [9]. These tools trace and log the activities of Web servers and measure the time spent on various tasks, as explained in Section 2.4.

2.2.4 Limitations of WebSTONE v2.0 over High-speed Networks

Although WebSTONE measures several key performance metrics, a study of its source code revealed that it uses process-based concurrency on UNIX, *i.e.*, multiple Web clients are spawned on a host using the `fork` system call. Context switching overhead between these processes is high. This overhead limits the number of concurrent clients on a single host to ∼10, which prevents the clients from heavily loading the Web server.[3] The standard WebSTONE model is acceptable for ordinary LANs (*e.g.*, 10 Mbps Ethernet). However, the high cost of ATM interfaces limits the number of available machines to use in our high-speed testbed (*i.e.*, 155 Mbps ATM). Thus, this model is unsuitable for our purposes.

To overcome these limitations, we modified WebSTONE's concurrency mechanism to use threads rather than processes. This modification alleviated the client-side limitations and enhanced the ability of the testbed to stress Web servers. As a result, our modified WebSTONE required just one process per host. In contrast, the original WebSTONE process-based concurrency model required *14 processes* per host. Therefore, the servers in our study could be subjected to a much higher number of simultaneous connections per Web client. The tests reported below use 42 concurrent connections on three hosts (*i.e.*, 14 concurrent connections per host).

[3] The number of processes can increase beyond 10, limited by the available memory of the machine. However, beyond 10 processes, the context switching overhead becomes very high, which causes a performance bottleneck on the *client* side and inhibits WebSTONE's ability to stress the Web server.

2.3 Blackbox Performance Analysis

The following WebSTONE blackbox metrics were measured in our Web server performance study. These metrics were obtained using a range of simultaneous connections from 1 to 42. Server file caches were pre-loaded by running a "dummy" client before doing the performance measurements. We present the blackbox results below. The whitebox results for each server are presented in Section 2.4.

Server throughput: This measures the number of bits the server writes onto the network per second. Figure 2 depicts the results. The process-based concurrency models of Apache exhibits the lowest overall throughput. The multi-threaded Netscape Enterprise server consistently outperforms the other servers; the Process Pool based Zeus server also performs quite well. Both sustained aggregate throughput higher than 35 Mbps over the 155 Mbps ATM network (for one concurrent connection to the server, the server throughput is low because the average size of the requested files are relatively small).

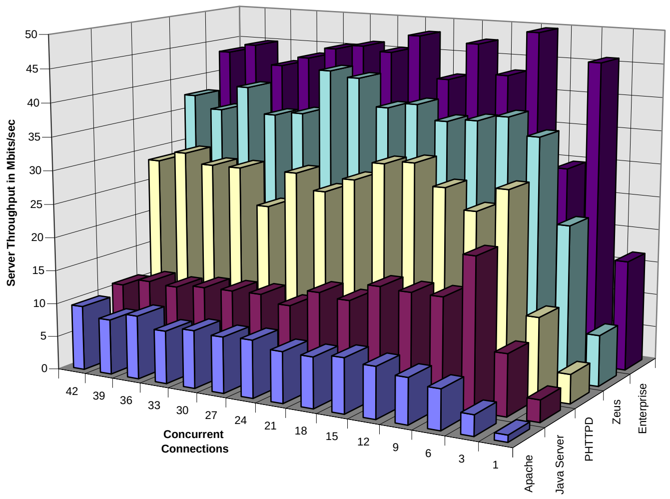

Figure 2: Average Server Throughput

Server connections/sec: This metric computes the number of connections the server completes per second. The results are shown in Figure 3. This figure depicts how many connections are completed per second by the servers, as we increase the number of simultaneous connections to the server. The Enterprise server completed more connections per second than other Web servers, followed by the Zeus server.

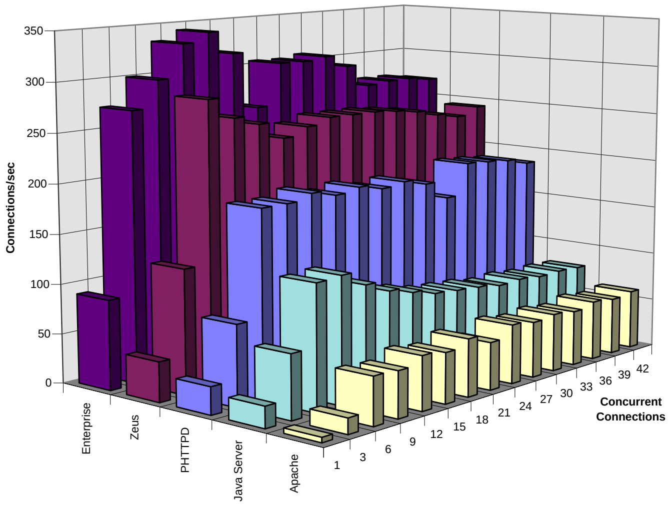

Figure 3: Average Connections/sec

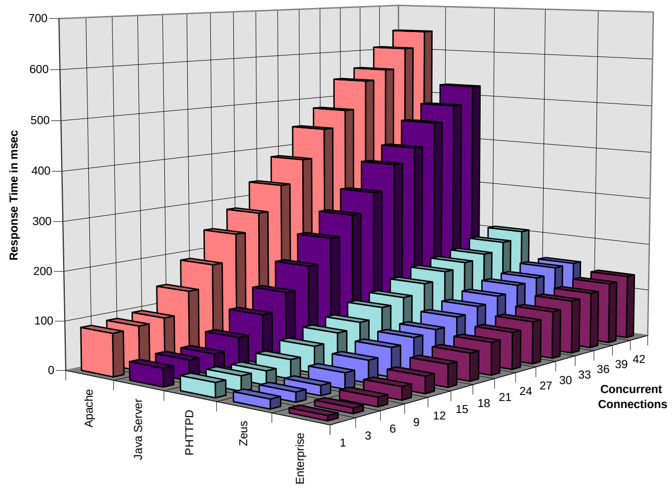

Figure 5: Average Client Latency

Client throughput: This is the average number of bits received per second by the client. The number of bits received includes the HTML headers sent by the server. The results are depicted in Figure 4. Clearly, as the number of concurrent

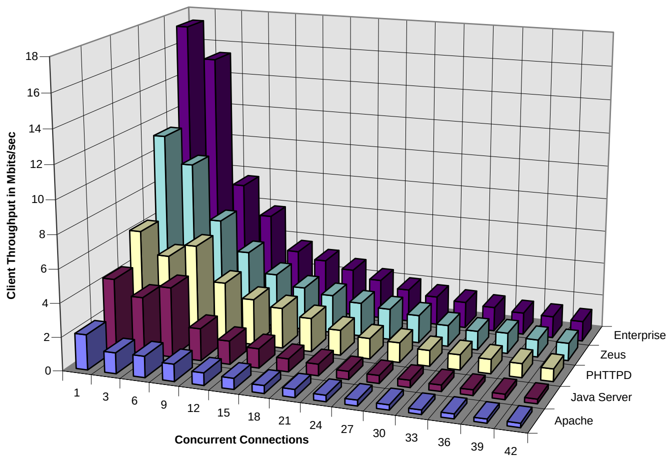

Figure 4: Average Client Throughput

clients increases, the server throughput is multiplexed amongst a larger number of connections, and hence the clients' average throughput drops. Therefore, those servers that exhibit high server throughput (*e.g.*, Enterprise and Zeus) also exhibit correspondingly high average client throughput.

Client latency: Latency is defined as the average amount of delay in milliseconds seen by the client from the time it sends the request to the time it completely receives the file. The results are shown in Figure 5. This graph is similar to the client throughput graph in Figure 4. It shows how the latency observed by individual clients increases, especially for the process-based concurrency mechanism employed by Apache.

2.4 Whitebox Performance Analysis

To determine the design and implementation issues that cause Web servers to exhibit the blackbox performance results in Section 2.3, we conducted the following whitebox experiment. Every Web server in our suite was subjected to an identical load. 15 simultaneous connections were set up to the Web server. Each connection made 1,000 requests (*i.e.*, 15,000 total requests). The access patterns of these requests were identical to those presented in Section 2.2.2.

While the Web server was serving these requests, we used the Solaris `truss` and `TNF` tools to count the number of system calls made and the time spent in each call. We then categorized these calls into the tasks shown in Table 2.

Task	System calls
File operations	`open, close, stat`, etc.
Writing files	`write, writev, ioctl`, etc.
Reading requests	`read, poll, getmsg`, etc.
Signal handling	`sigaction, sigsetmask`, etc.
Synchronization	`lwp_mutex_{lock,unlock}`, etc.
Process control	`fork, execve, waitid, exit`, etc.
Miscellaneous	`time, getuid`, etc.

Table 2: Categories of Web Server System Call Tasks

In addition, we used a software monitoring tool called `truss` to measure the amount of time the Web server spent at user-level (*i.e.*, when the server was not making a system call). This allowed us to estimate the amount of time each Web server spent in HTTP processing. The whitebox results for each Web server are presented below, ordered by increasing performance.

Note that the *ideal* Web server would spend most of its time performing network I/O, *i.e.*, reading HTTP requests and writing requested files to the network. In particular, it would have negligible overhead resulting from synchronization (due to efficient concurrency control and threading strategies) and filesystem operations (due to caching).

2.4.1 Apache Whitebox Analysis

The results of the whitebox analysis of Apache are illustrated in Figure 6. The key determinants of Apache performance are

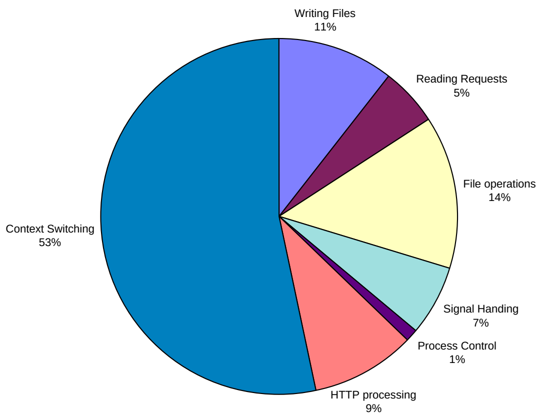

Figure 6: Apache Whitebox Analysis

outlined below:

Process-based concurrency: The primary performance bottleneck of Apache is its process-based concurrency model, where concurrent connections are handled by concurrent processes. Apache performs a number of optimizations to reduce the overhead of process creation. The server pre-forks a number of children into a process-pool to handle connections which are yet to be established. A new child is spawned if the pool is depleted. When a child finishes handling a connection, it returns to the pool. Thus, the cost of process creation is amortized by handling multiple requests *iteratively*, which has less overhead than a pure process-per-request model. Even with these optimizations, it is possible for the number of active processes to match the number of concurrent connections, which yields high context switching overhead.[4]

The `.htaccess` mechanism: A feature of the Apache Web server is its ability to configure Web server behavior on a per-directory basis (*e.g.*, different error messages for different directories). This is achieved by placing a configuration file (typically called `.htaccess`) in every directory that contains files being served by Apache. When a request arrives for a file, the server tries to open the `.htaccess` file in that directory, in an attempt to read the per-directory configuration. In addition, the Web server attempts to read all the `.htaccess` files in directories *above* the one containing the requested file. This causes a very large number of `open` system calls (∼100,000 calls for 15,000 requests). Moreover, the `open` calls fail if the `.htaccess` file is not present, which executes error handling code. Clearly, this feature increases latency since it causes the server to make several filesystem calls and handle their failure while a client waits for a response.

Lack of effective file-caching: Analysis of the system call trace reveals that the Apache server does not perform file caching. Instead, an `open` system call is made for each file requested and the file is read into memory before being sent to the client.

Memory-mapped files are not used: After the file has been opened, the server reads the file contents into a memory buffer via the `read` system call and then writes it out to the network. This introduces *data copying overhead*. This overhead can be avoided with *memory-mapped files*, via the UNIX `mmap` system call. The `mmap` call makes the contents of a file available to a process as a pointer to its address space. This technique is explained further in Section 4.

Signal handling: A significant fraction (7%) of the total time was consumed in signal handling activities. For example, `sigaction` installs signal handlers for the `SIGUSR1` signal and `sigprocmask` controls the responses to various signals. Apache employs signals to assist inter-process communication between the multiple server processes. For instance, the `SIGUSR1` signal is used by the parent process to inform child server processes that they should quit, so that new processes can be launched with fresh configuration information.

2.4.2 Java Server Whitebox Analysis

The results of the whitebox analysis of Java Server are illustrated in Figure 7. The key determinants of Java Server performance are outlined below:

Context switching: The Java Web Server uses the Java runtime system, which supports user-level threads. On Solaris, it uses the `getcontext` and `setcontext` system calls[5]

[4]The context switching overhead was calculated by using the UNIX `time` utility, which reports the total time spent by kernel on behalf of the Apache Web server (*i.e.*, in system calls) and the total user level time consumed by the Web server process. These times were subtracted from the "wall-clock" time reported by `time`, to yield the context-switching overhead. The overhead for the other servers were negligible and was not computed.

[5]Java Server uses the "green-threads" (*i.e.*, user-level threads) Java Virtual Machine implementation.

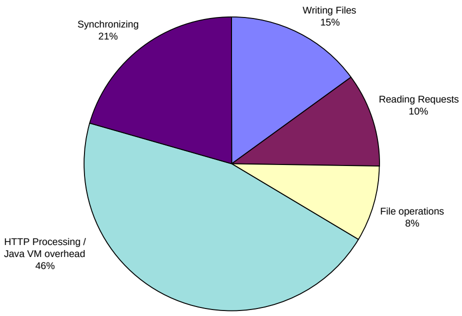

Figure 7: Java Server Whitebox Analysis

to context switch between the Java threads. The Java Server spends over 20% of its per-request processing performing user-level context switching. Moreover, context switching overhead increases as the number of concurrent clients grows. Hence, the average server throughput decreases and the client latency increases as the number of simultaneous connections increases, as shown in Figures 4 and 5.

HTTP processing and Java VM overhead: Apart from the user-level context switching mentioned above, a large fraction (46%) of the total time is consumed by user-level activity, *i.e.*, HTTP processing and Java VM processing. This overhead constitutes the main performance bottleneck of Java Server.

2.4.3 PHTTPD Whitebox Analysis

The results of the whitebox analysis of PHTTPD are illustrated in Figure 8. The key determinants of PHTTPD performance

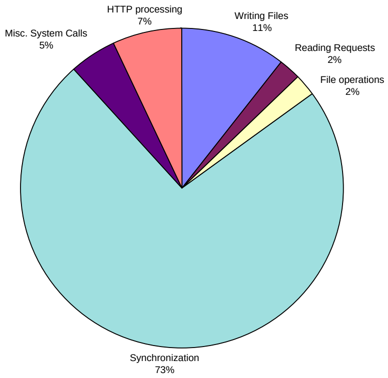

Figure 8: PHTTPD Whitebox Analysis

are outlined below:

Concurrency model: PHTTPD is a single-process multi-threaded server and employs the Thread-per-Request concurrency model. Thus, a new thread is created to handle each incoming request, which increases the latency seen by the client. It also causes the number of threads to grow rapidly as the number of outstanding requests increases, which increases context switching overhead.

Background hostname lookups: In addition to the threads mentioned above, PHTTPD spawns threads to perform hostname lookups in the background. Numeric IP addresses of HTTP clients are available locally using the `getsockname` system call. However, the full hostname is more useful for logging client hits.

Background `ident` lookups: PHTTPD also uses threads to perform `ident` [10] lookups in the background. The `ident` lookups obtain the name of the user who issued the HTTP request, which is typically used for logging purposes.

Synchronization: The threads spawned by PHTTPD use mutex locks and semaphores to serialize access to shared data (*e.g.*, the file cache table). Our whitebox analysis reveals that synchronization overhead (*i.e.*, the time spent by PHTTPD in acquiring and releasing locks) is the major performance bottleneck in PHTTPD, contributing over 70% of its total execution time. For instance, for 15,000 requests approximately 600,000 mutex locks were acquired by various threads, causing an average of 40 locks acquired *per request* (an equal number of mutex locks were released).

2.4.4 Zeus Whitebox Analysis

The results of the whitebox analysis of Zeus are illustrated in Figure 9. The key determinants of Zeus performance are

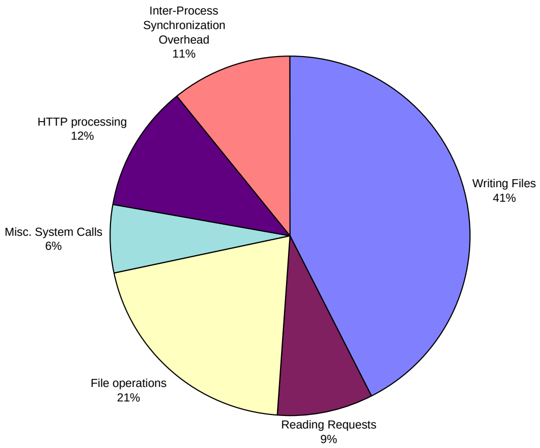

Figure 9: Zeus Whitebox Analysis

outlined below:

Concurrency model: Zeus employs a static process pool model of concurrency. The server initialization process creates a pool of processes, which is known as *pre-forking*. All processes in the pool run concurrently, though each process serves its requests using asynchronous I/O mechanisms. The use of a Process Pool avoids the overhead of dynamic process creation. The cache also limits the number of processes and the context switching overhead. This accounts for the superior performance of Zeus over Apache, which also employs a process-based concurrency model, as explained in Section 2.4.1. The Zeus documentation recommends that the number of the processes in the pool should equal the number of processors. Therefore, we used two processes in our tests since the server ran on a dual-processor machine, as explained in Section 2.2.1.

Memory-mapped files and file caching: Zeus uses memory-mapped files via the use of the `mmap` system call. Therefore, files need not be read into server memory before being written to the Network. This technique is explained further in Section 4. In addition, each file is memory-mapped the first time it is requested so that subsequent calls bypass the filesystem. This indicates that Zeus employs *file-caching*, which contributes to its relatively high performance.

Optimized file-writes: Zeus uses the `writev` system call, which allows multiple buffers to be written to a socket with a single system call. The `writev` call transmits the HTTP header and the requested file in a *single* system call, as explained in Section 4.

Synchronization overhead: Since all processes in the Zeus Process Pool serve connections arriving on the same socket, they use inter-process synchronization to synchronize access via the UNIX process-level advisory file locking APIs (*i.e.*, the `fcntl` system call). This overhead is significant (10%).

Other system calls: Zeus also spends appreciable time (11%) performing in miscellaneous operations and signal handling. The major contributors include obtaining the current time via the `time` system call and setting signal masks via the `sigprocmask` system call.

2.4.5 Enterprise Whitebox Analysis

The results of the whitebox analysis of Netscape Enterprise are illustrated in Figure 10. The key determinants of Netscape Enterprise performance are outlined below:

Concurrency model: Enterprise is a multi-threaded server that employs the thread pool concurrency model. These threads are created at server initialization time. This avoids the overhead of dynamic thread creation and thus improves latency.

Optimized file writes: Like Zeus, Netscape Enterprise employs the `writev` optimizations. Therefore, it sends the HTTP header and the requested file using a single system call.

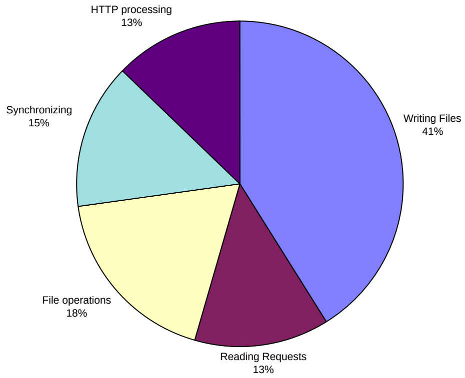

Figure 10: Enterprise Whitebox Analysis

Memory-mapped files and file caching: The Enterprise server employs memory-mapped files via the `mmap` system call and file-caching to minimize filesystem access. This is evident from the `truss` output, which reveals that files requested by clients are only mapped into memory the first time they are requested. Subsequent calls to the same file are serviced from the cache.

Optimized synchronization mechanisms: A system call trace of Enterprise reveals that although the threads acquire and release locks for accepting new connections, once a new request has been accepted *no locking overhead is incurred until after the request has been completely serviced.* This reduces client latency, as seen in Figure 5.

Thus, a combination of lightweight concurrency model, efficient synchronization mechanisms, and file-caching optimizations gives Enterprise superior performance over the other Web servers benchmarked in our ATM experiments.

3 Strategies for Developing High-Performance Web Servers

The analysis in Section 2 illustrates the superior performance of Netscape Enterprise and Zeus and identifies key factors that determine Web server performance. These factors include the *server concurrency strategy*, *synchronization overhead*, *protocol processing overhead*, and *caching strategy*. Applying whitebox measurement techniques in our ATM/Web Server testbed enabled us to determine precisely *why* Netscape Enterprise and Zeus perform much better than other Web servers over high-speed ATM networks.

After empirically determining the key Web server performance factors, our next objective was to develop an OO Web server development framework called JAWS. JAWS is designed to systematically develop and test the performance impact of different Web server design strategies and optimization techniques. This section outlines the object-oriented design of JAWS and presents the results of systematically applying techniques uncovered in the analysis in the previous section. We conclude this section by demonstrating how a highly optimized version of JAWS gives equivalent (and sometimes superior) performance compared with Netscape Enterprise and Zeus.

3.1 The Object-Oriented Architecture of JAWS

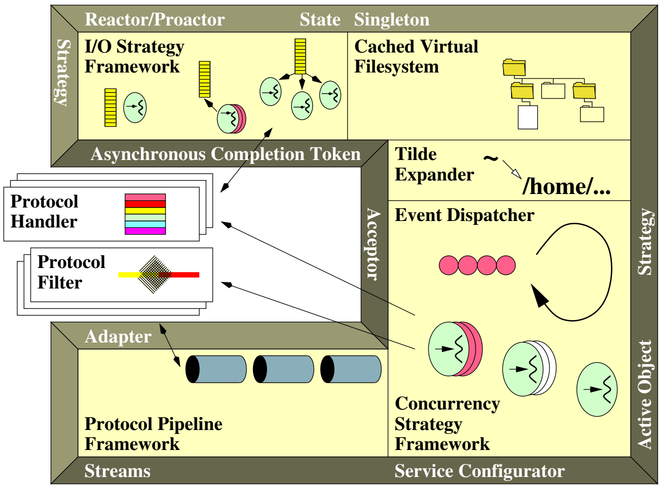

Figure 11: The Object-Oriented Architecture of JAWS

Figure 11 illustrates the OO software architecture of the JAWS Web server. As shown in Section 2, concurrency strategies, event dispatching, and caching are key determinants of Web server performance. Therefore, JAWS is designed to allow these Web server strategies to be customized according to key environmental factors. These factors include traffic patterns, workload characteristics, support for kernel-level threading and/or asynchronous I/O in the OS, and the number of available CPUs.

JAWS is structured as a framework [22] that contains the following components: an *Event Dispatcher*, *Concurrency Strategy*, *I/O Strategy*, *Protocol Pipeline*, *Protocol Handlers*, *Cached Virtual Filesystem*, and *Tilde Expander*. Each component is structured as a set of collaborating objects implemented with the ADAPTIVE Communication Environment (ACE) C++ communication framework [21]. Each component plays the following role in JAWS:

Event Dispatcher: This component is responsible for coordinating the *Concurrency Strategy* with the *I/O Strategy*. As events are processed, they are dispensed to the *Protocol Handler*, which is parameterized by a concurrency strategy and an I/O strategy, as discussed below.

Concurrency Strategy: This implements concurrency mechanisms (such as single-threaded, Thread-per-Request, or synchronous/asynchronous Thread Pool [8]) that can be selected adaptively at run-time or pre-determined at initialization-time. These strategies are discussed in Section 3.2.3.

I/O Strategy: This implements the I/O mechanisms (such as asynchronous, synchronous, and reactive). Multiple I/O mechanisms can be used simultaneously.

Protocol Handler: This component allows developers to apply the JAWS framework to create various Web server configurations. A Protocol Handler is parameterized by a concurrency strategy and an I/O strategy (though these remain opaque to the protocol handler). In JAWS, the Protocol Handler implements parsing and processing of HTTP request methods. The abstraction allows other protocols (*e.g.*, HTTP/1.1 and DICOM) to be incorporated easily into JAWS. To add a new protocol, developers simply implement a new Protocol Handler, which is then configured into the JAWS framework.

Protocol Pipeline: This component provides a framework to allow a set of filter operations (*e.g.*, compression, decompression, and parse HTML) to be incorporated easily into the data being processed by the Protocol Handler. This enables a server programmer to easily incorporate functional extensions (such as image filters or database operations) transparently into the Web server.

Cached Virtual Filesystem: This component improves Web server performance by reducing the overhead of filesystem access. The caching policy is strategized (*e.g.*, LRU, LFU, Hinted, and Structured). This allows different caching policies to be profiled for effectiveness and enables optimal strategies to be configured statically or dynamically. These strategies are discussed in Section 3.2.4.

Tilde Expander: This mechanism is another cache component that uses a perfect hash table [20] to map abbreviated user login names (*e.g.*, `~schmidt`) to user home directories (*e.g.*, `/home/cs/faculty/schmidt`). When personal Web pages are stored in user home directories (and user directories do not reside in one common root), this component substantially reduces the disk I/O overhead required to access a system user information file, such as `/etc/passwd`.

In general, the OO design of JAWS decouples the functionality of Web server components from their implementation strategies. For instance, the JAWS Concurrency Strategies can be decoupled from its Protocol Handlers. Thus, a wide range of strategies can be supported, configured, tested, and evaluated. As a result, JAWS can adapt to environments that may require different concurrency, I/O, and caching mechanisms. This additional flexibility is not antithetical to performance, as shown in Section 3.3.1 where JAWS demonstrates that decoupled and flexible Web server designs can achieve superior performance.

3.2 Performance Impacts of Web Server Strategies

The following subsection describes the concurrency, I/O, and, caching strategies supported by JAWS. The discussion focuses on the performance of the various strategies and how they interact with each other. The JAWS framework allows the Web server strategies to be changed easily, which facilitates controlled measurements of different server configurations. The results of this study are described below.

3.2.1 JAWS Baseline

Our study of the performance impact of different Web server strategies began with a version of JAWS that was not tuned with any optimizations. This *baseline* implementation of JAWS consists of its original default run-time configuration, running with a pool of 20 threads. Below, we illustrate the performance impacts in relation to the baseline implementation.

3.2.2 Protocol Processing Optimizations

Our initial optimizations for JAWS implemented techniques that reduced protocol processing overhead. These techniques included: caching the HTTP response header, lazy time header calculation, and the use of the `writev` system call to send multiple buffers of data in a single operation. Figure 12 shows the performance improvements when these enhancements were implemented. As shown in the figure, these optimizations resulted in a ~65% improvement in server throughput over the baseline version.

3.2.3 Concurrency Strategies

Our experiments in Section 2 suggest that the choice of concurrency and event dispatching strategies significantly impacts the performance of Web servers that are subject to changing load conditions. Carrying these results forward, we determined the quantitative performance impacts of using different concurrency strategies (*i.e.*, Thread Pool and Thread-per-Request), as well as varying parameters of a particular concurrency strategy (*e.g.*, the minimum and maximum number of active threads) in JAWS.

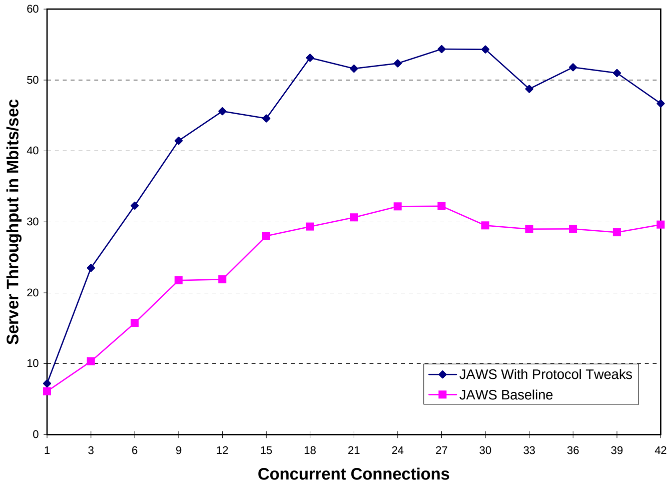

Figure 12: JAWS Performance After Protocol Optimizations

Thread Pool results: In the *Thread Pool* model, a group of threads are spawned at initialization time. All threads block in `accept`[6] waiting for connection requests to arrive from clients. This eliminates the overhead of waiting to create a new thread before a request is served. The Thread Pool model is used by the JAWS baseline implementation.

The performance graph in Figure 13 compares the performance of JAWS using the Thread Pool strategy on a dual-CPU UltraSPARC 2, while varying the number of threads in the Thread Pool. Note that the server throughput does not correlate clearly with the size of the Thread Pool. In addition, as we increase the size of the Thread Pool the variance in server throughput is not appreciable. Therefore, we conclude that a smaller Thread Pool (*e.g.*, 6 threads) performs just as well as a larger Thread Pool (*e.g.*, 42 threads). However, the traffic patterns used in our benchmarks (shown in Table 1) exhibit a large distribution of small files. Therefore, if the distribution shifted to larger files, a larger Thread Pool may behave more efficiently than a smaller Thread Pool because a smaller Thread Pool will be depleted with many long running requests. In this case, latency for new requests will increase, thereby decreasing the overall throughput.

To avoid underutilizing CPU resources, the number of threads in the Thread Pool should be no lower than the number of processors in the system. Thus, Figure 13 illustrates that server throughput is low when only one thread is in the Thread Pool, especially under higher loads (> 24 concurrent connections). This behavior is due to the absence of concurrency.

[6] Several operating systems (*e.g.*, Solaris 2.5) only allow one thread in a process to call `accept` on the same port at the same time. This restriction forces JAWS to use thread mutexes to serialize `accept` calls, which introduces additional synchronization overhead.

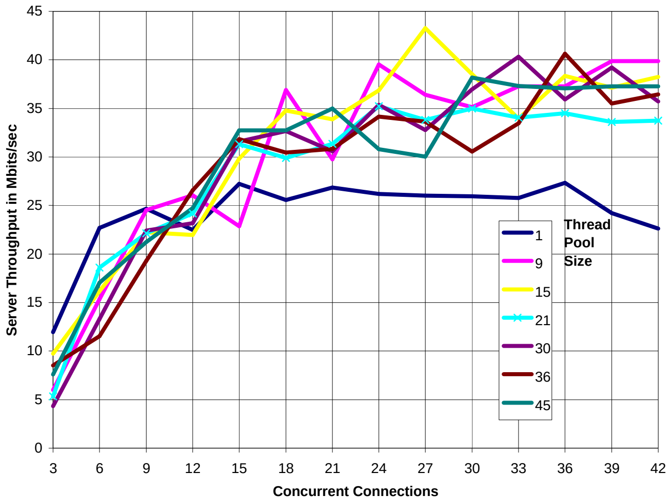

Figure 13: Performance of the JAWS Thread Pool

Thread-per-Request results: A common model of concurrency (*e.g.*, used by `inetd`) is to spawn a new process to handle each new incoming request. *Thread-per-Request* is similar, except that threads are used instead of processes. While a child process requires a (virtual) copy of the parent's address space, a thread shares its address space with other threads in the same process.

[Note to reviewers: We did not have enough time to complete Thread-per-Request results for this paper. If the paper is accepted, Thread-per-Request results will be included in the final submission.]

3.2.4 File Caching Strategies

Our analysis in Section 2 determined that accessing the filesystem is a significant performance inhibitor. This concurs with other Web server performance research [15, 27] that uses caching to achieve better performance. While the baseline version of JAWS does employ caching, it spends too much time *synchronizing* concurrent thread access to the Cached Virtual Filesystem (CVF).

To address this concern, the CVF was re-engineered. The new implementation accounted for the following factors:

- *Locking the entire cache can be avoided* – The original implementation locked on entry to every operation, because each operation was implemented to modify shared state. Once this requirement was removed, greater concurrency was achieved by only locking the hashed index entry of the cache.

- *The cache lock can be inherited by the file* – Acquiring a new lock for the cached object itself is unnecessary since all users of the cached file must acquire it through the CVF. Thus, within the CVF, operations on a cached file are serialized automatically.

Figure 14 shows a significant performance gain with the new CVF. In particular, with lower synchronization overhead the performance improves by as much as 40% when concurrent contention for files is high.

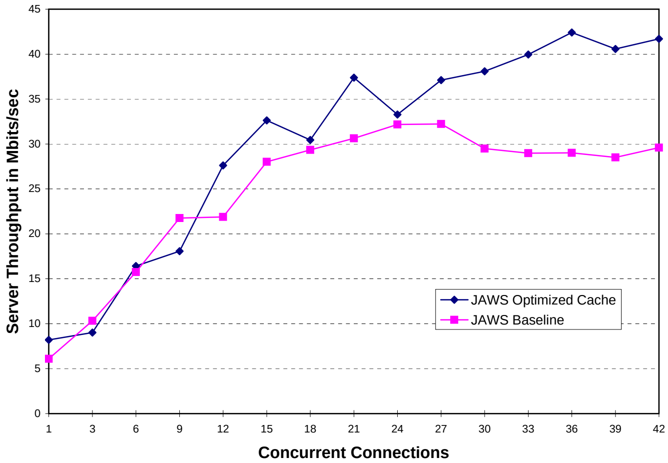

Figure 14: JAWS Performance after File Cache Optimizations

3.3 Performance Analysis

In the previous section, we demonstrated the impact that each design strategy had on the performance of JAWS compared to its baseline implementation. Below, we provide detailed performance analysis of the optimized version of JAWS. We first present whitebox performance results comparing the JAWS baseline with the optimized JAWS. We conclude with blackbox benchmarking results that demonstrate how the optimized JAWS outperforms Netscape Enterprise and Zeus, which have the best performance of the Web servers benchmarked in Section 2.3.

3.3.1 JAWS Whitebox Analysis

This section compares whitebox analysis of the JAWS baseline implementation against the optimized JAWS implementation. Figure 15 illustrates the percentage of time the JAWS baseline spends servicing HTTP requests.

In contrast, Figure 16 provides insight into how combining the optimization strategies analyzed in the previous section helped to improve the performance of JAWS. In particular, the synchronization time is reduced by 7%, and the network transfer time increased by 5%.

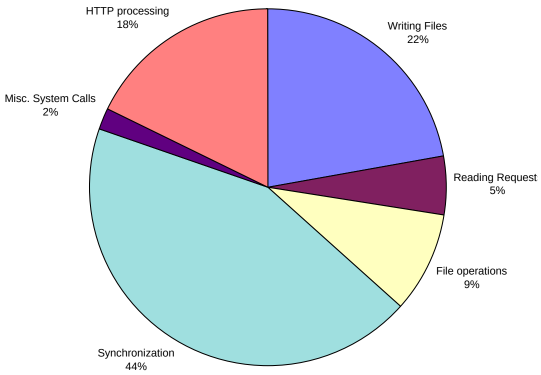

Figure 15: Whitebox Analysis of Baseline JAWS

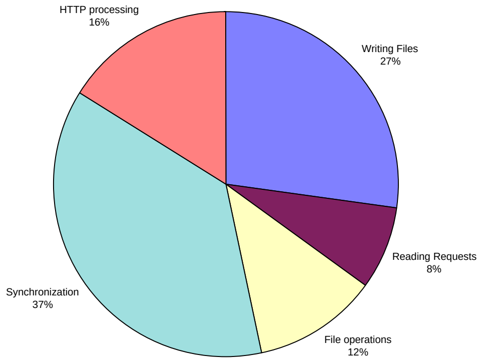

Figure 16: Whitebox Analysis of Optimized JAWS

Earlier in this section we described the individual impacts of applying the different design strategies to the baseline JAWS. Our whitebox results demonstrate that combining these techniques have yielded an optimized version of JAWS with much improved performance. The deployment of the protocol optimization strategies reduced HTTP processing time and the improved file cache implementation minimized synchronization overhead. The use of a tuned Thread Pool strategy removes thread creation overhead and minimizes the resource utilization of the server.

3.3.2 JAWS Blackbox Analysis

We conclude this section by comparing the benchmark results of the optimized JAWS against Netscape Enterprise and the Zeus Web servers. Figures 17-20 provide a new insight using the same metrics described in Section 2.3. These metrics reveal the following conclusion: *JAWS is capable of outperforming the best existing Web servers*.

This result confirms that a open flexible Web server framework is capable of providing equivalent performance to the best commercial Web servers. We believe this achievement is possible due to JAWS' adaptive framework that allowed us to systematically tune run-time parameters to optimize JAWS' performance. With automated adaptation, it should be possible for JAWS to dynamically adjust its behavior at run-time to handle different server load conditions than those encountered during our benchmarking tests.

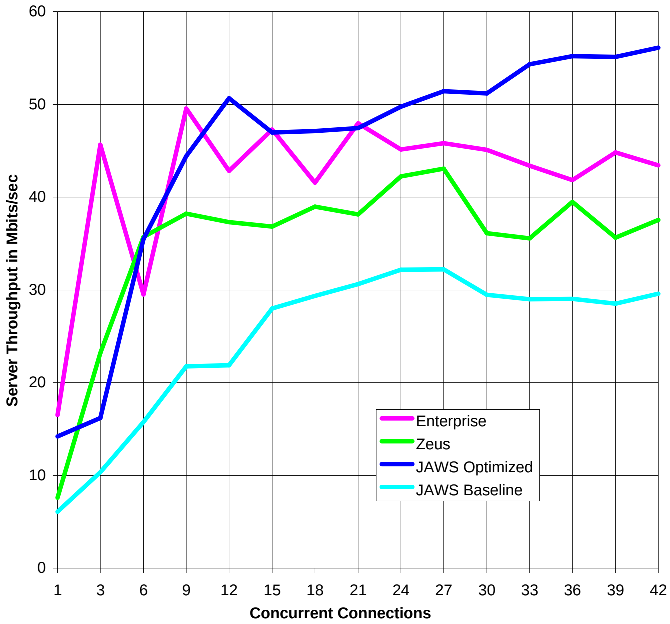

Figure 17: Average Server Throughput

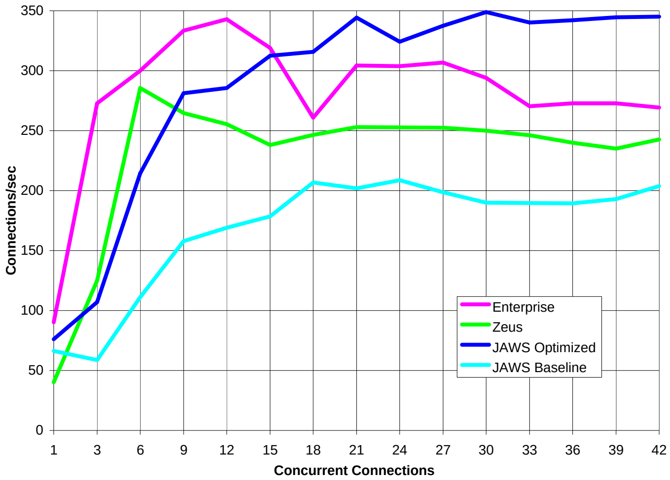

Figure 18: Average Connections-per-Second

Further evidence of the need for adaptivity is seen in the performance difference between JAWS and Netscape Enter-

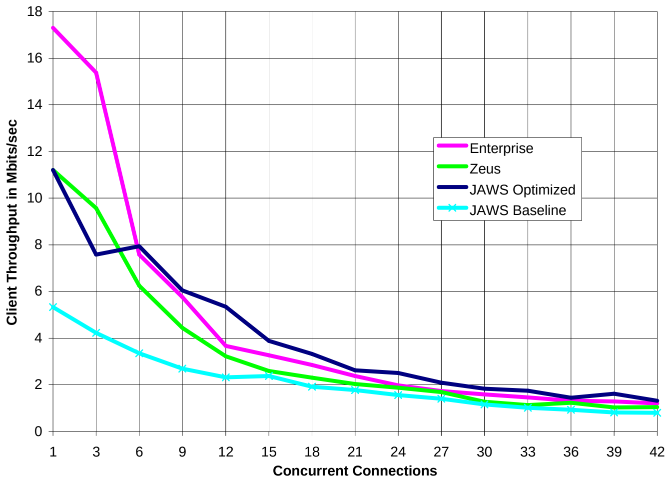

Figure 19: Average Client Throughput

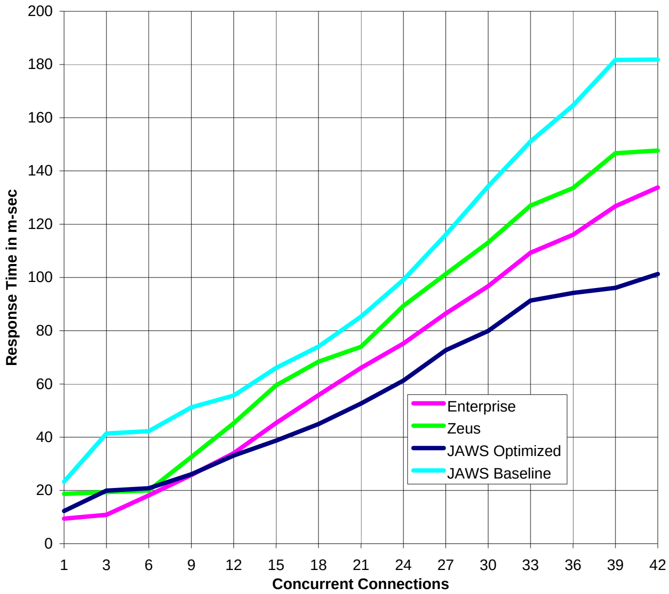

Figure 20: Average Client Latency

prise. Although JAWS consistently outperforms Enterprise under heavy loads, Enterprise consistently delivers higher server throughput during light loads. These results indicate the need to alter server behavior to handle light *vs.* heavy loads. Further research is necessary to reveal how Enterprise delivers this performance.

4 Summary of Web Server Optimization Techniques

Based on our study of existing Web server designs and implementation strategies, as well as our experience tuning JAWS, the following summarizes optimizations for developing high-performance Web servers.

Lightweight concurrency mechanism: Process-based concurrency mechanisms can yield poor performance, as is evident in the case of the Apache Web server. In multi-processor systems, a process-based concurrency mechanism might perform well, as in the case of Zeus, especially when the *number of processes are equal to the number of processors*. In this case, each processor can run a Web server process and context switching overhead is minimized.

In general, processes should be *pre-forked* to avoid the overhead of dynamic process creation. However, it is preferable to use lightweight concurrency mechanisms, (*e.g.*, using POSIX threads) to minimize context switching overhead. Similar to processes, threads should be created at server startup time and organized into a thread pool to avoid dynamic thread creation overhead.

Critical path system call overhead: The critical path in a Web server is defined as the sequence of instructions that must be executed by the server after it receives an HTTP request from the client and before it sends out the requested file. The time taken to execute the critical path of instructions directly impacts the latency observed by clients. Therefore, it is important to minimize system call overhead and other processing in the critical path. The remainder of this section describes various places in Web servers where such overhead can be reduced.

Synchronization mechanism: Process-based concurrency mechanisms often employ signals to assist synchronization. For instance, Apache uses the `SIGUSR1` signal to restart all the server processes. In such cases, the use of signal masking calls like `sigprocmask` should be minimized since they cause substantial overhead, as observed for Apache in Figure 6.

For thread-based concurrency mechanisms, synchronization using locks should be minimized. In particular, it is important to minimize the number of locks acquired (or released) on the critical path. Servers that average a lower number of lock operations per request (Enterprise performs ∼4) perform much better than servers that perform a high number of lock operations (PHTTPD averages ∼40 lock operations per request).

In some cases, acquiring and releasing locks can also result in *preemption*. Thus, if a thread reads in an HTTP request and then attempts to acquire a lock, it might be preempted, and may wait for a relatively long time before it is dispatched again. This increases the latency incurred by a Web client.

File Caching and the `mmap` mechanism: If the Web server does not perform file caching, the overhead of the `open` system call is typically incurred on the critical path. Servers that perform file caching (*e.g.*, Enterprise and Zeus) perform much better than servers that do not (*e.g.*, Apache). Caching can be effectively performed using memory-mapped files (*e.g.*, Solaris provides the `mmap` system call). Memory-mapped files have two advantages over the conventional `read`/`write` I/O cycle:

• **Reduced data copying overhead:** Since the file is directly mapped into memory, it need not be read into memory buffers before being transmitted to the client.

• **Reduced critical path length:** Since system calls like `open` and `read` are avoided, the critical path becomes shorter, thereby reducing latency.

The "gather-write" mechanism: The `writev` system call allows multiple buffers to be written to a device in a single system call. This is useful for Web servers since the typical server response to a valid client request is composed of the following two parts:

• **The HTTP header:** This header contains various HTTP-related information, such as the the HTTP success code, the document type, and the document date.

• **The requested file:** Using the `mmap` technique discussed earlier, this file is typically available in the server's address space.

If these two buffers are transmitted using a single `writev` call the overhead of switching between user-mode and kernel-mode will be reduced by a factor of $\sim$2.

Logging overhead: Most Web servers support features that allow administrators to log the number of hits on various pages they serve. Logging is often done to estimate the load on the server at various times during the day. It is also commonly performed for commercial reasons, *e.g.*, Web sites might base their advertising rates on page hit frequencies. However, logging HTTP requests causes a significant overhead for the following reasons:

• **Filesystem calls:** A heavily loaded Web server makes a significant number of I/O calls, which stresses the filesystem and underlying hardware. Writing data to log files increases this stress and thus contributes to lower performance. Keeping log files and the HTTP files on separate filesystems and, if possible, on separate physical devices can limit this overhead.

• **Increase in critical path complexity:** Logging requests in the critical path can cause severe performance penalties since typically file operations are required to write the log information to disk. This overhead can be reduced by *batching log file writes*. Batching allows several requests to be logged to a memory buffer, which is written to disk *lazily* (*e.g.*, when the server is not servicing requests or after the memory buffer grows beyond a certain threshold).

• **Synchronization overhead:** A typical Web server has multiple active threads or processes serving requests. If these threads/processes are required to log requests to a common shared log file, access to this log file needs to be synchronized, *i.e.*, at most one thread/process can write to the shared log file at any time. This synchronization introduces additional overhead and is thus detrimental to performance. This overhead can be reduced by keeping multiple independent log files. If memory buffers are used, these should be stored in *thread-specific storage* to eliminate locking contention.

• **Reverse hostname lookups:** The IP address of the client is available to a Web server locally. However, the hostname is typically more useful information in the log file. Thus, the IP address of the client needs to be converted into the corresponding host name. This is typically done using *reverse DNS lookups*. Since these lookups often involve network I/O, they are very costly. Therefore, they should be avoided or done in background threads (*e.g.*, as done by PHTTPD).

• **Ident lookups:** The Ident protocol [10] allows a Web server to obtain the user name for a given HTTP connection. This typically involves setting up a new TCP/IP connection to the user's machine and thus involves a round-trip delay. Also, the ident lookup must be performed while the HTTP connection is active and therefore cannot be performed lazily. To achieve high performance, such lookups must thus be avoided whenever possible.

Pre-computation of HTTP responses: Typical HTTP requests result in the server sending back the HTTP header, which contains the HTTP success code and the MIME type of the file requested, (*e.g.*, `text/plain`). Since such responses are part of the expected case they can be *pre-computed*. When a file enters the cache, the corresponding HTTP response can also be stored along with the file. When an HTTP request arrives, the header is thus directly available in the cache. This saves processing time on the critical path.

The `time` system call: Web servers are expected to send out HTTP responses that contain the current time. This time stamp can be used by clients to determine how current the information requested is if the last modification time is also forwarded. Invoking the `time` system call on receipt of every request causes the Web server to incur the overhead of switching between user-mode and kernel-mode. This overhead can be overcome by caching the time and only changing it when absolutely necessary (such as, when a file has a newer modification time than the current cached time). Netscape Enterprise uses this scheme.

Transport layer optimizations: The following transport layer options should be configured to improve Web server performance over high-speed networks:

• **The listen backlog:** Most TCP implementations buffer incoming HTTP connections on a kernel-resident "listen queue" so that servers can dequeue them for servicing using `accept`. If the TCP listen queue exceeds the "backlog" parameter to the `listen` call, new connections are refused by TCP. Thus, if the volume of incoming connections is expected to be high, the capacity of the kernel queue should be increased by giving a higher backlog parameter (which may require modifications to the OS kernel).

• **Socket send buffers:** Associated with every socket is a send buffer, which holds data sent by the server, while it is being transmitted across the network. For high performance, it should be set to the highest permissible limit (*i.e.*, large buffers). On Solaris, this limit is 64k.

• **Nagle's algorithm (RFC 896):** Some TCP/IP implementations implement Nagle's Algorithm to avoid *congestion*. This can often result in data getting delayed by the network layer before it is actually sent over the network. Several latency-critical applications (such as X-Windows) disable this algorithm, (*e.g.*, Solaris supports the `TCP_NO_DELAY` socket option). Disabling this algorithm can improve latency by forcing the network layer to send packets out as soon as possible.

5 Related Work

Measuring and analyzing the performance of Web servers is an increasingly popular research topic. Existing research on improving Web performance has focused largely on reducing network latency, primarily through caching techniques [15, 27] or protocol optimizations [14, 19, 17]. In addition, the workload of Web servers have been modeled analytically at the University of Saskatchewan [12].

SGI's WebSTONE is widely considered as the standard benchmarking system for measuring the performance of Web servers [6], and it is the basis of our own benchmarking methodology. A detailed analysis of the bottlenecks in a single Web server (*i.e.*, Apache) has been performed at Boston University [1]. Our work extends this work by benchmarking a wide range of Web servers to identify the impacts of alternative server designs.

Another way to improve Web performance is by removing overhead in the protocol itself. The W^{3}C has recently standardized HTTP/1.1, which enables multiple requests over a single connection. This "connection-caching" strategy can significantly enhance the performance over HTTP/1.0 [25, 19, 17]. The need for persistent connections to improve latency was noted by Mogul [14]. Latency can also be improved by using caching proxies and caching clients, although the removal policy needs to be carefully considered [27]. JAWS can be extended to become a caching proxy, allowing it to leverage directly from this work. Yeager and McGrath of NCSA discuss many of these issues in [16].

The analyses presented above have aided our work, which focuses on improving end-to-end Web performance by improving the efficiency of Web servers. The related work on caching is particularly rich, and our own results corroborate its importance. However, we extend the related work by studying the performance impacts of different combinations of concurrency, caching, and I/O dispatching strategies for Web servers that are subjected to varying loads. Workload studies provide evidence that Web server workloads can vary from server to server, motivating the need for adaptation. Work on removing protocol overhead will make Web server bottlenecks more prominent. We believe there is a need for comprehensive understanding of *when* to apply various combinations of optimization strategies in order to overcome these bottlenecks.

The JAWS framework facilitates the fundamental understanding of Web server performance by holding the development framework constant and systematically configuring and testing different strategies under varying load conditions on high-speed networks. This allows us to construct profiles that can predict the optimal combinations of strategies to use in different Web server conditions. By incorporating these profiles into the framework, JAWS can support automatic configuration of optimal combinations of concurrency, I/O, and caching strategies.

6 Concluding Remarks

The research presented in this paper was motivated by a desire to build high-performance Web servers. Naturally, it is always possible to improve performance with more expensive hardware (*e.g.*, additional memory and faster CPUs) and a more efficient operating system. However, our research objective is to produce the fastest server possible *for a given hardware/OS platform configuration*.

As shown in Section 2, we began by analyzing the performance of existing servers. The servers that performed poorly were studied to discover sources of bottlenecks. The servers that performed well were examined even more closely using whitebox techniques to examine what they did right. We found that checking and opening files creates significant overhead, which can be alleviated by applying perfect hashing and other caching techniques.

When network and file I/O are held constant, however, the largest portion of the HTTP request lifecycle is spent dispatching the `GET` request to the Protocol Handler that processes the request. The time spent in dispatching depends largely on the choice of the concurrency strategy. Our results show that no single concurrency strategy provides optimal performance in all circumstances.

In general, research on adaptive software has not been pursued deeply in the context of Web systems. Current research on Web *server* performance has emphasized caching [15, 27], concurrency [8], and I/O [16, 1]. While our results corroborate that caching is vital to high performance, non-adaptive caching strategies do not provide optimal performance in Web servers [11]. Moreover, current server implementations and experiments rely on *statically* configured concurrency and I/O strategies.

As a result of our empirical studies, we observed that servers relying on static, fixed strategies cannot behave optimally in many high load circumstances. Therefore, we conclude that high-performance Web servers must be *adaptive*, *i.e.*, be customizable to utilize the most beneficial strategy for particular traffic characteristics, workload, and hardware/OS platforms.

JAWS supports Web server adaptivity by providing a framework built using an adaptive communication environment (ACE) [21]. Future versions of JAWS will support prioritized request handling (to promote requests for smaller objects of requests for larger objects), dynamic protocol pipelines (to support optimal end-to-end data filtering operations, such as compression), as well as automatic configuration for concurrency, I/O dispatching and caching strategies. We believe that combining these techniques will produce a Web server that exhibits extremely low latency and high throughput.

The complete source code for JAWS is available at `www.cs.wustl.edu/~schmidt/ACE.html`.

References

[1] Jussara Almeida, Virgílio Almeida, and David J. Yates. Measuring the Behavior of a World-Wide Web Server. Technical Report TR-CS-96-025, Department of Computer Science, Boston University, October 29 1996.

[2] Anselm Baird-Smith. Jigsaw performance evaluation. www.w3.org/, October 1996.

[3] Kenneth P. Birman and Robbert van Renesse. Software for Reliable Networks. *Scientific American*, May 1996.

[4] Alexander Carlton. An Explanation of the SPECweb96 Benchmark. Standard Performance Evaluation Corporation whitepaper, 1996. www.specbench.org/.

[5] Zubin D. Dittia, Guru M. Parulkar, and Jr. Jerome R. Cox. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of INFOCOM '97*, Kobe, Japan, April 1997. IEEE.

[6] Gene Trent and Mark Sake. WebSTONE: The First Generation in HTTP Server Benchmarking. Silicon Graphics, Inc. whitepaper, February 1995. ww.sgi.com/.

[7] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, pages 306–317, Stanford, CA, August 1996. ACM.

[8] James Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks. In *Proceedings of the 2^{nd} Global Internet Conference*. IEEE, November 1997.

[9] PureAtria Software Inc. *Quantify User's Guide*. PureAtria Software Inc., 1996.

[10] M. St. Johns. Identification Protocol. *Network Information Center RFC 1413*, February 1993.

[11] Evangelos P. Markatos. Main memory caching of web documents. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[12] Martin F. Arlitt and Carey L. Williamson. Intenet Web Servers: Workload Characterization and Performance Implications. Technical report, University of Saskatchewan, December 1996. A shorter version of this paper appeared in ACM SIGMETRICS '96.

[13] Robert E. McGrath. Performance of Several HTTP Demons on an HP 735 Workstation. Avaiable from http://www.ncsa.uiuc.edu/, April 25 1995.

[14] Jeffrey C. Mogul. The Case for Persistent-connection HTTP. In *Proceedings of ACM SIGCOMM '95 Conference in Computer Communication Review*, pages 299–314, Boston, MA, USA, August 1995. ACM Press.

[15] Jeffrey C. Mogul. Hinted caching in the Web. In *Proceedings of the Seventh SIGOPS European Workshop: Systems Support for Worldwide Applications*, 1996.

[16] Nancy J. Yeager and Robert E. McGrath. *Web Server Technology: The Advanced Guide for World Wide Web Information Providers*. Morgan Kaufmann, 1996.

[17] Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *To appear in Proceedings of ACM SIGCOMM '97*, 1997.

[18] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. *USENIX Computing Systems*, 9(4), November/December 1996.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Standards Track RFC 2068, Network Working Group, January 1997. www.w3.org/.

[20] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Proceedings of the 2^{nd} C++ Conference*, pages 87–102, San Francisco, California, April 1990. USENIX.

[21] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the 6^{th} USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[22] Douglas C. Schmidt. Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software. In Peter Salus, editor, *Handbook of Programming Languages*. MacMillan Computer Publishing, 1997.

[23] David Strom. Web Compare. webcompare.iworld.com/, 1997.

[24] SunSoft Inc. *TNFtools:Programming Utilities Guide*. 2550 Garcia Avenue, Mountain View CA 94043, May 1995.

[25] T. Berners-Lee, R. T. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. Informational RFC 1945, Network Working Group, May 1996. www.w3.org/.

[26] J.S. Turner. An optimal nonblocking multicast virtual circuit switch. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 298–305, June 1994.

[27] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghalleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World Wide Web Documents. In *Proceedings of SIGCOMM '96*, pages 293–305, Stanford, CA, August 1996. ACM.