Techniques for Developing and Measuring High Performance Web Servers over High Speed 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*

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 prespawned 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 [7] and Kodak [13] 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.

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 [7].

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 [7] and [9].

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; and Section 5 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 [10], as well as information available from WebCompare [17].

2.2 Web Server/ATM Testbed

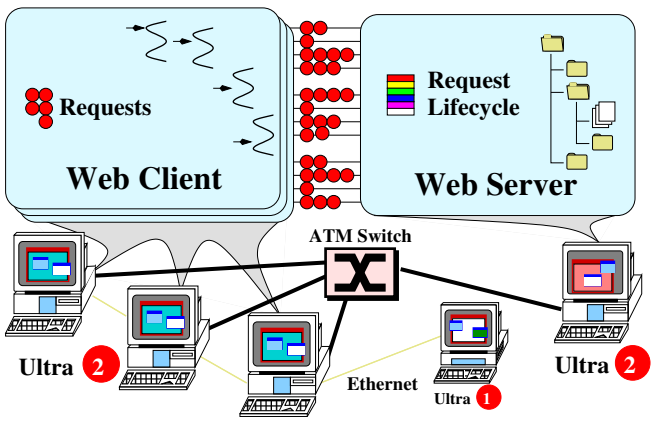

Figure 1: Web Server/ATM Testbed Environment

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 [19], as shown in Figure 1.[1]

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

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 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 [5].

2.2.2 Benchmarking Methodology

We used the WebSTONE [4] 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 (bytes)	500	5 K	50 K	5 M
Frequency	35%	50%	14%	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 [3].

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` [18], and `Quantify` [8]. These tools trace and log the activities of Web servers and measure the time spent on various tasks, as explained in Section 2.4.

2.3 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).

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.

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 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.

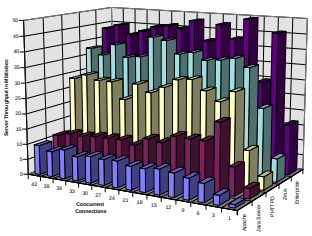

Figure 2: Server Throughput

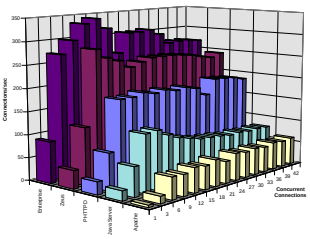

Figure 3: Connections/sec

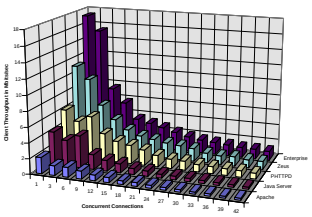

Figure 4: Client Throughput

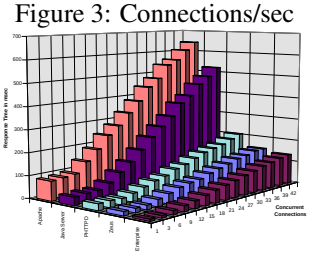

Figure 5: Client Latency

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).

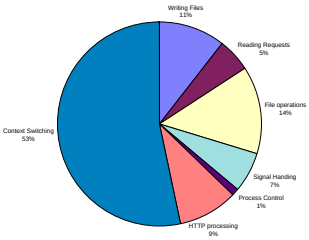

Figure 6: Apache

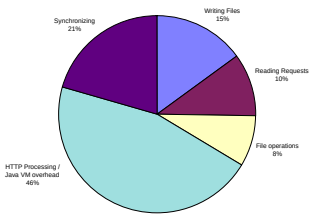

Figure 7: Java Server

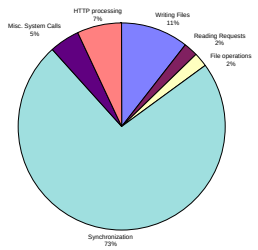

Figure 8: PHTTPD

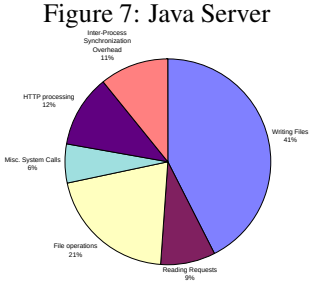

Figure 9: Zeus

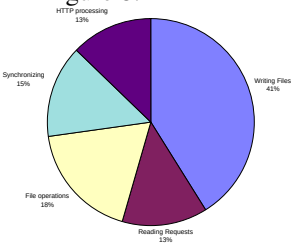

Figure 10: Enterprise

Whitebox Analysis Results

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*, *pro-*

tocol 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

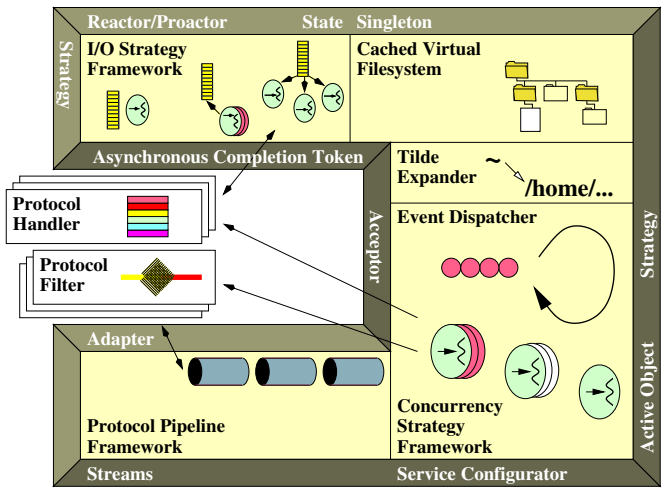

Figure 11: The Object-Oriented Architecture of JAWS

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 [16] 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 [15]. 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 [7]) 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 [14] 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 $\sim$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.

Thread Pool results: In the *Thread Pool* model, a group of threads are spawned at initialization time. All threads block in `accept`[2] 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.

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.

Figure 14 compares the performance of applying the Thread-per-Request strategy in JAWS with the baseline implementation. The graph depicts how performance varies with changing server loads, in comparison with the JAWS baseline

[2]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.

performance. This result illustrates the need for dynamic adaptivity. For low loads, (*i.e.*, up to 9 concurrent connections) the baseline Thread Pool implementation performs well. However, at higher loads, the Thread-per-Request model offers superior performance. This is because the Thread Pool model on Solaris requires additional synchronization while accepting new connections, as explained above.

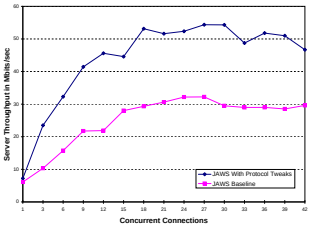

Figure 12: Protocol Optimizations

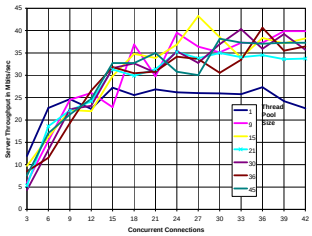

Figure 13: Thread Pool Sizes

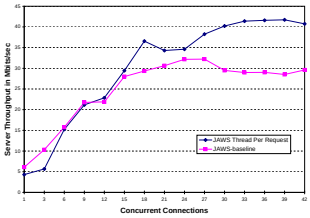

Figure 14: Thread-per-Request Model

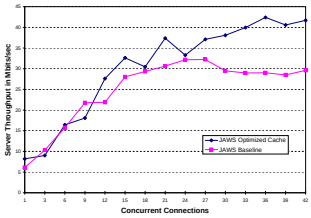

Figure 15: File Cache Optimizations

JAWS Performance Comparisons

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 [11, 20] 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 15 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.

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 16 illustrates the percentage of time the JAWS baseline spends servicing HTTP requests.

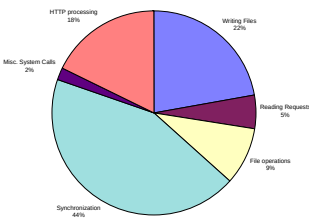

Figure 16: Baseline JAWS

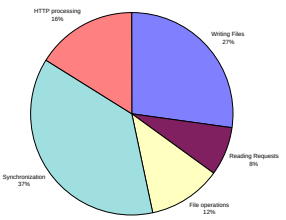

Figure 17: Optimized JAWS

JAWS Whitebox Analysis

In contrast, Figure 17 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%.

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 18-21 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.

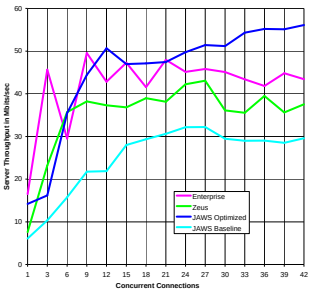

Figure 18: Server Throughput

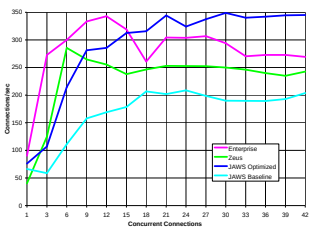

Figure 19: Connections/sec

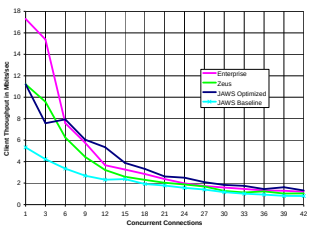

Figure 20: Client Throughput

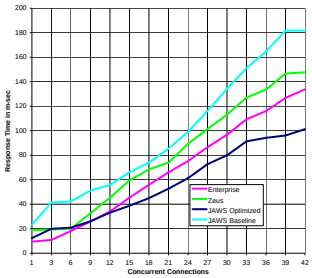

Figure 21: Client Latency

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

This section summarizes the most significant determinants of Web server performance. These observations are based on our studies of existing Web server designs and implementation strategies, as well as our experience tuning JAWS. These studies reveal the primary targets for optimizations to develop high performance Web servers.

Lightweight concurrency: Process-based concurrency mechanisms can yield poor performance, as seen in [6]. In multi-processor systems, a process-based concurrency mechanism might perform well, 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. As with processes, dynamic thread creation overhead can be avoided by *pre-spawning* threads into a pool at server start-up.

Specialized OS features: Often times, OS vendors will provide specialized programming interfaces which may give better performance. For example, Windows NT 4.0 provides the `TransmitFile` function, which uses the Windows NT virtual memory cache manager to retrieve the file data. `TransmitFile` allows data to be prepended and appended before and after the file data, respectively. This is particularly well-suited for Web servers since they typically send HTTP header data with the requested file. Hence, all the data to the client can be sent in a single system call, which minimizes mode switching overhead.

Usually, these interfaces must be benchmarked carefully against standard APIs to understand the conditions for which the special interface will give better performance. In the case of `TransmitFile`, our empirical data indicate that the asynchronous form of `TransmitFile` is the most efficient mechanism for transferring large files over sockets on Windows NT, as shown in [7].

Request lifecycle system call overhead: The request lifecycle 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 request lifecycle directly impacts the latency observed by clients. Therefore, it is important to minimize system call overhead and other processing in this path. The following describes various places in Web servers where such overhead can be reduced.

• **Reducing synchronization:** When dealing with concurrency, synchronization is often needed to serialize access to shared resources (such as the Cached Virtual Filesystem). However, the use synchronization penalizes performance. Thus, it is important to minimize the number of locks acquired (or released) during the request lifecycle. In [6], it is shown that servers that average a lower number of lock operations per request perform much better than servers that perform a high number of lock operations.

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.

• **Caching files:** If the Web server does not perform file caching, at least two sources of overhead are incurred. First, there is overhead from the `open` system call. Second, there is accumulated overhead from iterative use of the `read` and `write` system calls to access the filesystem, unless the file is small enough to be retrieved or saved in a single call. Caching can be effectively performed using memory-mapped files, available in most forms of UNIX and on Windows NT.

• **Using "gather-write":** On UNIX systems, 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 contains a number of header lines in addition to the requested file. By using "gather-write", header lines need not be concatenated into a single buffer before being sent, avoiding unnecessary data-copying.

• **Pre-computing 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.

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 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 [11, 20], concurrency [7], and I/O [12, 1]. While our results corroborate that caching is vital to high performance, non-adaptive caching strategies do not provide optimal performance in Web servers [9]. 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) [15]. 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. Available from http://www.w3.org/, October 1996.

[3] Alexander Carlton. An Explanation of the SPECweb96 Benchmark. Standard Performance Evaluation Corporation whitepaper, 1996. Available from http://www.specbench.org/.

[4] Gene Trent and Mark Sake. WebSTONE: The First Generation in HTTP Server Benchmarking. Silicon Graphics, Inc. whitepaper, February 1995. Available from http://www.sgi.com/.

[5] 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.

[6] James Hu, Sumedh Mungee, and Douglas C. Schmidt. Principles for Developing and Measuring High-performance Web Servers over ATM. In *Submitted to INFOCOM '97 (Washington University Technical Report #WUCS-97-10)*, February 1997.

[7] 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.

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

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] David Strom. Web Compare. Available from http://webcompare.iworld.com/, 1997.

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

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

[20] 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.