Principles for Developing and Measuring High-performance Web Servers over ATM

James C. Hu[1], Sumedh Mungee, Douglas C. Schmidt

{jxh,sumedh,schmidt}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

This paper has been submitted for publication. It my be referenced as Technical Report WUCS-97-09.

Abstract

High-performance Web servers are essential to meet the growing demands of the Internet. Satisfying these demands requires a thorough understanding of the key factors that affect Web server performance. This paper provides three contributions to the design, implementation, and evaluation of high-performance Web servers.

First, we report the results of a comprehensive empirical study of popular high-performance Web servers (such as Apache, Netscape Enterprise, PHTTPD, and Zeus) over high-speed ATM networks. This study illustrates their relative performance and identifies their performance bottlenecks. To measure performance accurately, we developed a new benchmarking technique that subjects Web servers to varying connection frequencies. We found that once network and disk I/O overheads are reduced to negligible constant factors, the main determinant of Web server performance is the concurrency strategy employed by the server. Moreover, no single strategy performs optimally for all load conditions and traffic types.

Second, we describe the performance optimizations used to develop our a high-performance Web server: JAWS. 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 every server in our test suite. The performance optimizations used in JAWS include adaptive pre-spawned threading, intelligent caching, and prioritized request processing.

Third, we outline the design principles and concurrency patterns of JAWS, focusing on its novel object-oriented software architecture. This architecture substantially improves the portability and flexibility of JAWS, relative to other Web servers. For instance, JAWS can configure itself automatically at run-time or installation-time to use the most efficient concurrency strategy for a given OS platform. In addition, JAWS can be configured to support multiple transport protocols (such as HTTP/1.0 and HTTP/1.1). Our work illustrates that highly efficient communication software is not antithetical to highly flexible software.

1 Introduction

During the past two years, the volume of traffic on the World Wide Web (Web) has grown dramatically. The increased traffic is due largely to the proliferation of inexpensive and ubiquitous Web browsers (such as NCSA Mosaic, Netscape Navigator, and Internet Explorer). To keep pace with this demand, it is essential to develop high-performance Web servers.

This paper makes the following contributions to the measurement and development of high-performance Web servers:

- **Empirical analysis of existing Web server performance:** We present results from our systematic performance analysis of existing Web server over a high-speed ATM network. Our study concentrates on the Zeus, PHTTPD, Netscape Enterprise, Apache, Roxen and Jigsaw Web servers. These servers were selected since they represent a wide range of points in the Web server design space.

  All Web servers were benchmarked using black-box performance metrics (such as throughput and latency). In addition, white-box metrics (such as thread creation and synchronization overhead) were collected for those Web servers with freely available source code. Our analysis revealed that once network and disk I/O overheads are reduced to negligible constant factors (*e.g.,* ATM networks and a very large memory cache), the primary determinant of Web server performance is its *concurrency strategy*.

- **Design principles for building high-performance Web servers:** We describe the design and implementation of our own high-performance Web server, JAWS. JAWS is a multi-threaded Web server that was explicitly designed to alleviate the performance bottlenecks we identified in existing Web servers. JAWS consistently outperforms every server in our test suite and can service hundreds of requests per second during peak usage over a high-speed ATM network.

  This paper presents the object-oriented design of JAWS, focusing on the general design principles and concurrency

[1] Funding provided by *Object Technologies International* and by *Eastman Kodak Company, Rochester, N.Y.*.

patterns for building high-performance Web servers. We describe the optimizations used to reduce the sources of overhead identified by our benchmarks and empirical analysis. In addition, we outline the adaptive concurrency design of JAWS, which can adjust its run-time concurrency behavior to provide optimal performance for particular traffic characteristics and workloads.

Existing research on improving Web performance has focused largely on reducing network latency, primarily through caching techniques [17] or protocol optimizations [11, 13, 3]. Moreover, measurements of server performance have focused largely on low-speed networks [1, 9]. However, this is the first paper that we are aware of that describes the design principles and techniques necessary to develop high-performance Web servers.

The remainder of this paper is organized as follows: Section 2 describes our ATM testbed for benchmarking Web servers and presents the results of these benchmarks and our analysis; Section 3 outlines the object-oriented design of JAWS and compares the performance of JAWS with the Web servers evaluated in Section 2; Section 4 describes related work; and Section 5 presents concluding remarks and outlines additional optimizations to improve Web server performance.

2 Experimental Results of Web Server Performance over ATM

2.1 Web Server/ATM Testbed Environment

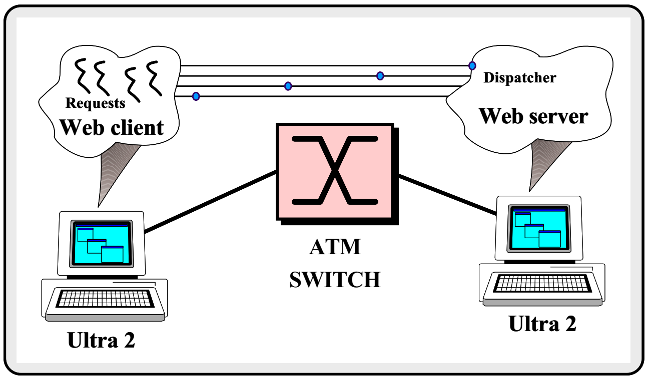

Figure 1: Web Server/ATM Testbed Environment

2.1.1 Hardware and Software Platforms

We began our study of Web servers performance by observing how available servers performed on high-speed networks under high workloads. To accomplish this, we constructed a hardware and software testbed consisting of a high-performance server and client connected by a high-speed ATM switch [22] (shown in Figure 1).[2]

The experiments in this paper were conducted using a Bay Networks LattisCell 10114 ATM switch connected to two 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 two 168 MHz Ultra SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the `STREAMS` communication framework.

Each UltraSparc-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode 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].

2.1.2 Client Traffic Generators

To provide a thorough and accurate understanding of server performance bottlenecks, we designed and implemented a new benchmarking technique. In this technique, a *client traffic generator* sends HTTP `GET` requests to a Web server at a particular *rate*. The results of the tests are collected and reported by the client after all the requests have completed.

We had two motivations for this design:

• **Predictability:** Our client traffic generator permits us to control the request rate per second to within 3 significant digits. For example, 2.15 requests per second would correspond to 129 requests per minute. Conventional benchmarking tools [6, 4] also issue requests at various rates. They do not permit fine grain control on the request rate, however, since their designs use separate client processes that issue requests iteratively using a *flooding* traffic generation model.

• **Controlling external factors:** We wanted to examine the point at which the HTTP request rate dramatically alters Web server performance. We achieved this by running our rate-based client traffic generator over ATM. By hold network and file I/O constant, allowed us to precisely pinpoint the request rates where a server starts to overload. These results help to create an accurate behavioral model for Web servers.

Our client traffic generator is based on the ACE network programming toolkit [15], which is an object-oriented framework composed of strategic and tactical design patterns that simplify the development of high-performance, concurrent communication software. ACE has been ported to a wide range of uni-processor and multi-process OS platforms including Win32 (*i.e.*, WinNT and Win95), most versions of UNIX (*e.g.*, SunOS 4.x and 5.x, SGI IRIX, HP-UX, OSF/1,

[2] We also performed measurements over 10 Mb/s ethernet, but due to a lack of performance variance, we ommitted the discussion from this paper.

AIX, Linux, and SCO), VxWorks, and MVS OpenEdition. The client traffic generator used the Sun SC3.0.1 C++ compiler suite, along with version 4.1.2 of the Adaptive Communication Environment (ACE).

2.1.3 Performance Metrics

Two *black-box* metrics, *throughput* and *latency*, were included in our Web server performance study. Throughput is defined as the average number of bits received per second by the client. A high-resolution timer for throughput measurement was started before the client benchmarking software sent the HTTP request. The high-resolution timer stops just after the connection is closed at the client end. The number of bits received includes the HTML headers sent by the server.

Latency is defined as the average amount of time that elapses before the client traffic generator receives the first packet from the server. It measures how long an end user must wait after sending an HTTP `GET` request to a Web server, and before the content begins to arrive at the client. The timer for latency measurement is started just before the client benchmarking software sends the HTTP request and stops just after the client receives the first response from the server.

Several *white-box* methods (such as thread creation and synchronization overhead) were collected using Quantify v2.2 [16]. Quantify analyzes server performance by identifying the functions and system calls that contribute significantly to server performance degradation at high-frequency of requests. It reports these measures both as a count of the number of accesses made to functions and system calls, as well as the amount of time spent in each function and system call.

The black-box analysis is presented in Section 2.2 and white-box analysis is presented in Section 3.

2.1.4 Web Servers

The choice of the servers for our test suite were based upon three factors. The first was *market penetration*, in order to gauge the performance of servers that are widely used. The second was *reported performance*, as reported by benchmark results published by Jigsaw and NCSA, as well as information available from WebCompare [2, 9, 20]. The third was *source/object code availability*, which is essential for white-box analysis. Based on these criteria, we selected a mixture of commercial and free server implementations, with source available for all of the free servers.

The servers chosen for our tests were Apache, PHTTPD, Roxen, Jigsaw, Netscape Enterprise and Zeus. In addition, our own Web server, JAWS, was benchmarked in the experiment, as well. Section 3 outlines the design principles and patterns used to develop JAWS.

Apache and PHTTPD were implemented in C. Roxen is an extensible interpreted server written in Pike (which is an interpreted language). Jigsaw is implemented in Java (which is also an interpreted language). The Netscape Enterprise and Zeus servers are commercial servers with only binaries available.

2.2 Performance Results

2.2.1 Methodology

Each experiment consisted of several *rounds*, one round for each server in our test suite. Each round was conducted as a series of benchmarking *sessions*. Each session consisted of having the benchmarking client issue a number of requests (N) for a designated file of a fixed *size*, at a particular frequency (ϕ). Each successive session increased the frequency by some fixed step value (δ) to a maximum frequency (Φ).

The client only requests a single file, and the same single file is used for every round of every session within a single experiment. The use of a single file is motivated by our goal of minimizing filesystem and ATM network overhead in order to isolate the impact of Web server design on performance.

We conducted the following three experiments:

1. Large file transfer: This black-box experiment subjected each server in our test suite to a series of requests for a single large file. This is typical of popular sites that contain a few popular large files, such as a new release of X windows. The content requested by clients was a 5 megabyte GIF file, which is the equivalent of a large desktop-sized 256-color picture.

2. Small file transfer: This black-box experiment consisted of requests for a single small file. This design is motivated by previous workload studies [8, 4]. These studies show that more than 80% of all requested files from a typical Web server are 10 kB or smaller, and that 85% of the requested files are from 15% of the files available on the Web server. Thus, it is reasonable that a high-performance Web server should perform well with large number of requests for a small number of files.

3. Profile analysis: This white-box experiment was conducted in two rounds on a single server that was instrumented for profile analysis by Quantify, discussed in Section 2.1.3. Each round corresponded to the *large file* and *small file* experiments described above. The purpose of the white-box experiment was to precisely identify the key sources of overhead that determine the black-box performance.

2.2.2 Web Server Performance for Large Files

• **Description:** The parameters for this experiment are outlined in Table 1. The results reported for this experiment consist of the average measurement over each of the 10 requests sent. The throughput and latency performance of all the Web servers in the suite are illustrated in Figures 2 and 3.

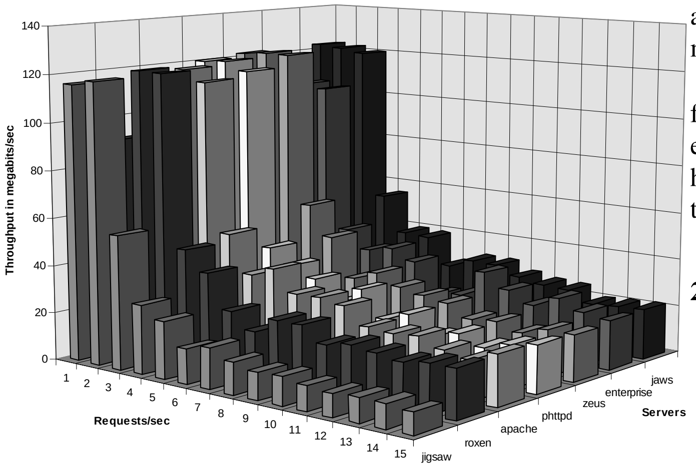

Figure 2: Throughput for Large File Transfers

File size	N	ϕ	Φ	δ
5 Mbyte	10	1 req/s	15 req/s	1 req/s

Table 1: Large File Experiment Parameters

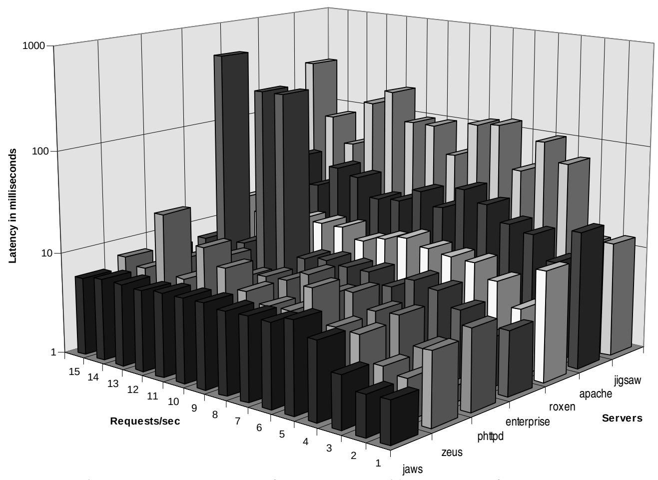

Figure 3: Latency for Large Files Transfers

• **Analysis:** Most of the servers perform well at low loads and can achieve high throughput. However, as we increase the request rate, throughput for all servers degrades rapidly and latency also rises significantly (the graph uses a logarithmic scale for the Z-axis).

These results illustrate that JAWS consistently outperforms the other Web servers. In addition, it provides the lowest latency to client requests, and also maintains consistently high throughput as workload increases. Section 3 explains the optimizations we used to achieve this performance.

2.2.3 Web Server Performance for Small Files

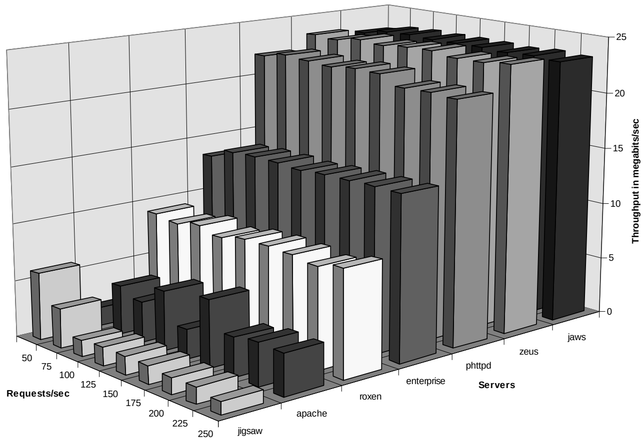

Figure 4: Throughput for Small File Transfers

• **Description:** The parameters for this experiment are outlined in Table 2. Once again, the throughput and latency were measured for each request for each request rate, and averaged over the 200 requests. The performance results for all Web servers appear in Figures 4 and 5.

File size	N	ϕ	Φ	δ
10 Kbyte	200	50 req/s	250 req/s	25 req/s

Table 2: Small File Experiment Parameters

Server Activity	Large file	Small file	Optimization
Dispatching	5%	9%	Adaptive concurrency
Read/parse request	1%	4%	
Data Logging*	–	–	Batch logging
Check file access	2%	4%	Caching
File open operations	15%-20%	25%-30%	Virtual filesystem
Read/send file	70%-75%	50%-55%	Virtual filesystem
Cleanup operations	2%	2%	

Table 3: White-box Analysis of the Apache Web server

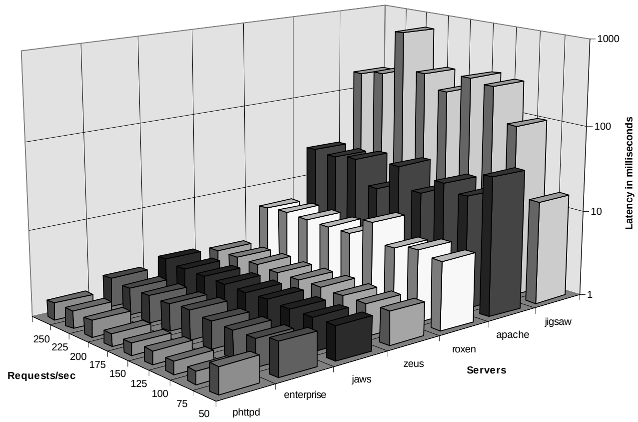

Figure 5: Latency for Small Files Transfers

• **Analysis:** Surprisingly, a large number of popular Web servers perform poorly while transferring small files. Notably, the Netscape Enterprise server attained only about half the throughput of the JAWS and Zeus servers.[3]

2.2.4 White-box Analysis of Web Server Performance

• **Description:** In this experiment, the Apache server was instrumented with Quantify [16]. Apache server was then subjected to workloads identical to those described in Section 2.2.2 and Section 2.2.3. For the purposes of this paper, only the Apache server was analyzed using white-box analysis. Since Apache was neither the best nor the worst performing server in our test suite, it serves as a representative implementation. It is also used to illustrate the life-cycle of an HTTP request.

Table 3 summarizes the Quantify results. The first column represents the typical life-cycle of an HTTP request and response from the perspective of a server. The second and third columns describes the percentage of time the server spent at each stage of the life-cycle. The fourth column suggests optimization techniques that can alleviate the performance bottlenecks, some of which have been implemented in JAWS.

• **Overview of an HTTP request life-cycle:** When an HTTP request arrives, it is dispatched to the code in the server that processes the request. Once the request is read and parsed, servers typically write out the information about the request into a logfile.[4] Once the request Uniform Resource Identifier (URI) has been converted into a filename, the file is opened and its contents read from the disk and delivered to the network interface. After the file is transferred to the client, any temporary data structures are cleaned up. Figure 6 illustrates this cycle.

• **Key sources of performance overhead:** As expected, Apache spends most of its time doing network and file I/O

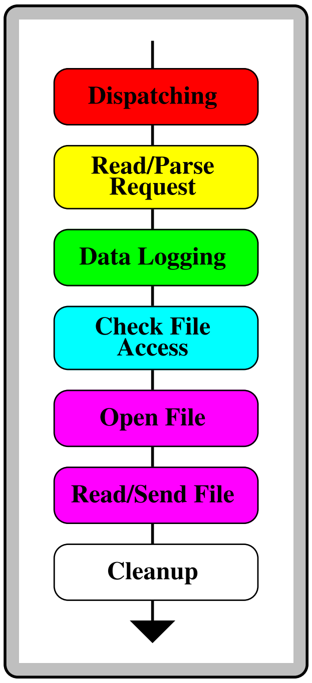

Figure 6: Typical HTTP Request Life-cycle

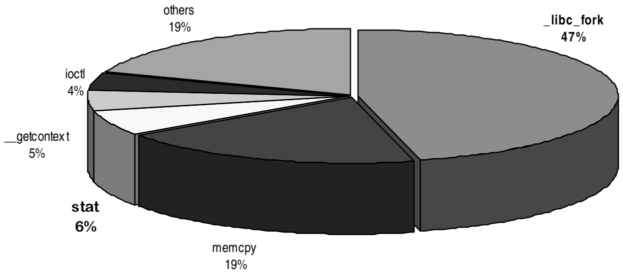

Figure 7: Performance Overhead in Apache, Excluding I/O

[3] Since the source code for Netscape Enterprise is not freely available, we were unable to identify its bottlenecks using white-box analysis.

[4] Logging was disabled for our experiments to reduce extraneous filesystem overhead.

operations. Including file open operations, the time it spends on network and file I/O ranges between 80% to 90%. However, removing I/O from the picture, dispatching comprises nearly 50% of the remaining sources of overhead, which can be seen in Figure 7.

Since dispatching is controlled by the concurrency strategy of the system, we concluded that the choice of concurrency strategy significantly impacts the performance of Web servers.

2.2.5 Analysis Summary

• **Compiled vs. interpreted programming languages:** The results presented above show that Web servers that use interpreted languages (such as Jigsaw and Roxen) cannot service very high request rates. However, the implementors chose to use interpreted approaches to achieve flexibility and modularity.

• **Concurrency strategies:** For single CPU machines, single-threaded solutions are acceptable and perform well.[5] However, they do not scale for multi-processor platforms, as shown by the performance of Roxen compared with the concurrent Web servers like Apache (which is process-based), PHTTP and JAWS (which is multi-threaded). Therefore, many high performance servers use multi-threaded concurrency strategies, such as Netscape Enterprise and Zeus.

Process-based concurrency implementations perform reasonably well when the network is the bottleneck. However, on high-speed networks like ATM, the cost of spawning a new process per request is relatively high. Apache implements an optimization by pre-forking the processes before requests arrive. This improves performance over ATM, but does not perform as well as threads on Solaris.

PHTTPD, a multi-threaded implementation, performs nearly as well as the top commercial server, Zeus. PHTTPD was carefully coded in C using Solaris threads and great care was taken to remove unnecessary latencies within the I/O system. Still, a new thread is spawned for each incoming request.

Netscape Enterprise and Zeus also perform well. Presumably, they are multi-threaded[6], but detailed information about their concurrency strategies are not available since they are only available in binary form.

3 Principles for Developing Extensible, High-performance Web Servers

As shown in Section 2.2, JAWS consistently outperforms the other servers in our test suite. During our study, we analyzed the results of our experiments to discover key Web server bottlenecks. We identified the following two key determinants of Web server performance:

[5] By single-threaded, we mean a single-threaded concurrent server based on the Reactor pattern.

[6] This is stated in their own product information.

• **Concurrency strategy:** Since dispatching occupies a large portion of non-I/O related Web server overhead, choosing the right concurrency strategy is a major determinant of performance.

• **Avoiding the filesystem:** We discovered that Web servers, that implemented sophisticated caching strategies performed much better than those that did not.

In this section, we outline the design principles and optimizations we applied when implementing JAWS.

3.1 The Object-Oriented Architecture of JAWS

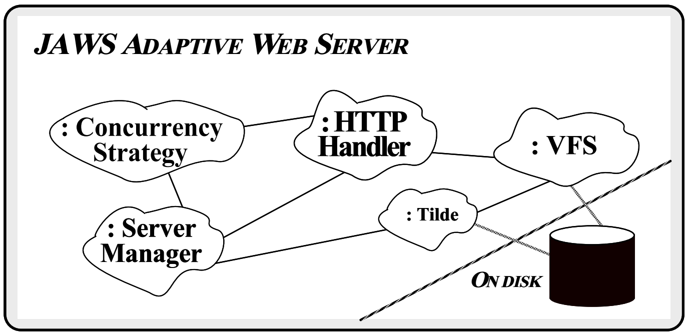

Figure 8: The Object-Oriented Software Architecture of JAWS

Figure 8 illustrates the object-oriented software architecture of the JAWS Web server. The black-box and white-box measurements in Section 2.2 indicated that the concurrency strategy used by a Web server largely determine its performance. Therefore, JAWS is designed to allow the concurrency strategy to be customized in accordance with key environmental factors such as user-level vs. kernel-level threading in the OS and the number of CPUs.

JAWS is organized into three decoupled components: an HTTP Handler, Server Manager, and Concurrency Strategy. Each component contains a collection of collaborating objects. The following summarizes the role each component plays in JAWS:

• **HTTP handler:** This implements the parsing of requests, and the operations for performing HTTP request methods. The server is designed to allow for other protocols (*e.g.*, HTTP/1.1 and DICOM) to be incorporated easily into the system. A new protocol would be implemented with its own handler.

• **Concurrency strategy:** This implements the concurrency mechanism, such as single-threaded, thread-per-request, or thread pool.

• **Server manager:** This component associates the HTTP handler (or any other protocol handler) with the Concurrency strategy. It also implements a small protocol which can direct the manager to dynamically re-link portions of the server code which have been updated.

By decoupling the concurrency strategy from the HTTP Handler, a wide range of strategies can be supported in JAWS. As a result, JAWS can adapt to environments that may require different concurrency mechanisms.

In addition to these three core components, JAWS contains two other important objects. One is a dynamically loaded perfect hash table of home directories, called `Tilde`. Another is a virtual filesystem, called CVF (Caching Virtual Filesystem, that caches information about the files so that future requests for the same file will not be needlessly looked up again.

The HTTP handler interacts with CVF. CVF interacts with `Tilde`. The Server Manager may reload `Tilde` if a new user is added to the system password file.

3.2 Principles for High-performance Web Server Development

3.2.1 Adaptive Concurrency Strategies

• **Iterative:** The easiest way to implement a server is to not have any concurrency at all. Such a server simply receives each request, and iterates through them one at a time, handling them in FIFO order. This kind of server is called an *iterative* server. This kind of server is not likely to perform well except in very low load servers. None of the servers in our test suite uses an iterative design.

• **Process-per-request:** A common concurrency strategy for Web servers is to `fork` a new process to handle each new request, Unfortunately, this strategy does not scale well on most OS implementations. Our measurements in our Solaris testbed indicated that a `fork` system call requires 5 ms to execute. This overhead yields poor performance during peak loads experienced by popular Web servers.

There are various techniques for alleviating this overhead. For instance, the Apache Web server eliminates the overhead of `fork` by pre-forking a *process pool*. To implement this scheme, Apache `fork`'s a certain number of processes at initialization time, before requests are received. As HTTP requests arrive, a dedicated request handling process is scheduled by the operating system to handle the request.

The process pool strategy achieves good results for moderately high loads. However, during heavy loads, the rate of requests eventually overrun the cache of handlers, and the method reverts to *process-per-request* strategy.

• **Thread-per-request:** *Thread-per-request* is similar, except that OS threads are used instead of processes. For each incoming request, a new thread is spawned to handle the incoming request. 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.

If the OS platform supports threads (and most modern operating systems do), thread-per-request is a more efficient concurrency strategy than process-per-request. In our testbed, the cost of issuing a thread create (`thr_create`) is less than 260 μs, which is approximately 20 times faster than `fork`. PHTTPD uses a thread-per-request strategy and it performs very well (as shown in Section 2.2).

• **Thread pool:** Even higher levels of performance can be achieved by modifying the Apache process pool model and *pre-spawning* a pool of threads. In this *thread pool* model, a number of threads are spawned at initialization time. All threads block in `accept` waiting for connection requests to arrive from clients. This eliminates the overhead of waiting for a new thread to be created before a request is served. This is the approach used by JAWS, although its object-oriented design permits for the integration of other concurrency strategies.

Since threads are created at initialization time, the thread pool concurrency strategy provides Web server administrators with the ability to control the threading policy. For example, in Solaris, threads can be created *bound* to an LWP or *unbound*.[7] In scenarios where processing are long running (*e.g.,* transferring large image files) and there are multiple CPUs, a bound thread may perform better than an unbound thread. However, bound threads are more expensive to create [5]. This cost becomes negligible, however, when pre-spawned thread pools are used.

• **Single-threaded concurrent:** Threads are a good choice overall, since they scale well to multiple CPU platforms. However, for single CPU platforms, the use of threads increase context switching and synchronization overhead. Hence, on a uni-processor platform it is usually more desirable to implement a *single-threaded concurrent* server. A single-threaded server is different from an iterative server (which also only occupies one thread) in that it handles multiple requests concurrently. This strategy is somewhat more complicated to implement, since it involves asynchronous I/O programming. The potential payoff, however, is a faster server for uni-processors.

3.2.2 Avoiding the Filesystem

• **Virtual file system:** Many Web servers transmit file via HTTP by reading a buffer of bytes from the file and writing those bytes to the socket descriptor connected with a client. This process is repeated until the file is transferred. While this technique is straightforward to implement, it is not very efficient [18]. The problem is that the data path of the transfer copies the bytes twice: once from the filesystem into main memory and once again from main memory to the network adapter.

In operating systems that support memory-mapped file I/O, a more efficient way to transmit data is to memory map the requested file. Once mapped, the entire file can be sent to the client in a single `write` operation. This technique minimizes the overhead of accessing the filesystem and leverages off of virtual memory management hardware.

[7] POSIX threads have similar attributes, PTHREAD_SCOPE_PROCESS, and PTHREAD_SCOPE_SYSTEM, which describes the scope of contention for the thread.

Although memory-mapped file I/O appears ideal, problems arise for highly loaded multi-threaded Web servers. Problems can arise when many clients attempt to access the same large file, or many different files. The operating system's virtual memory manager can easily be exhausted if too many map requests are issued since virtual memory addresses (which are finite resources) are allocated for each mapping.

To avoid this problem, JAWS implements a *virtual file system* called CVF. The CVF maintains reference count to cached information about each opened file. This allows multiple request handlers to utilize the same memory-mapped pointer to certain highly accessed files. To improve overall Web server virtual memory utilization, JAWS only memory maps for "large" files. Small files are read entirely into memory once and stored in JAWS VFS cache. A pointer to this memory is then used instead of memory mapping to prevent smaller files from occupying entire pages of memory.

• **Home directory expansion:** Many Web servers are configured so that file content is retrieved from a users home directory. The files are customarily referenced with `/~user/`, which denotes the home directory of a `user`. *Home directory expansion* is the process of converting the logical name (*i.e.,* `/~user/`) into an actual path (*i.e.,* `/home/cs/faculty/user/`).

Based on our `Quantify` measurements in Section 2.2.4, home directory expansion is a relatively expensive operation. The overhead stems from the need to perform a lookup in the system password file, which is often located on a remote file server. Experiments on our OS platform and network found that this lookup can take over 300 milliseconds if the user name has not been cached by the OS and network file system. Thus, performing a lookup for every HTTP `GET` request severely degrades end-to-end performance.[8]

The solution used in JAWS is to implement a dynamically loadable object instance that maintains a cache of user home directories. This cache is constructed externally to the Web server and utilizes perfect hashing, which provides $O(1)$ performance in the average- and worst-case. Perfect hashing is feasible since the password file is relatively static and changes infrequently, thereby amortizing the cost of reconstructing the perfect hash. Moreover, since the perfect hash function can be computed "off-line," sophisticated tools [14] can be used to generate the perfect hash table.

A JAWS server can be notified on-demand as the password file changes, in order to reload the cache. This dramatically improves performance over Web servers that do not cache home directory information.

• **File status information:** To build the response headers required for HTTP/1.0 and HTTP/1.1, certain status information about a file is required. This information includes the date of last modification and the file size. In addition, file permissions must be checked to ensure a user has sufficient privileges to retrieve the file.

On UNIX platforms, this information is available via the `stat` family of system calls. In early versions of UNIX, `stat` was an expensive system call because each invocation of `stat` resulted in a linear search through the names of the directory structure. To speed up common operations like directory listings, newer versions of UNIX implement `stat` more efficiently (*i.e.,* requiring less than 50 μs on average) by leaving the pointer at the name of the last successful `stat` call [10].

The information returned in a `struct stat` buffer must still be parsed, however. For instance, once `stat` returns, a Web server must immediately test to see if the requested file is a directory. If it is, another call to `stat` is usually performed to find the index page, *i.e.,* `index.html`.

Web server performance can be improved significantly if the status information about frequently accessed files is cached. In JAWS, this information is cached as part of its VFS component.

4 Related Work

Measuring and analyzing the performance of Web servers is an increasing popular research topic. Existing research on improving Web performance has focused largely on reducing network latency, primarily through caching techniques or protocol optimizations. In addition, the workload of Web servers have been modeled analytically at the University of Saskatchewan [8].

SGI's WebStone is widely considered as the standard benchmarking system for measuring the performance of Web servers [6]. A detailed analysis of a Pentium-based Apache Web server and its bottlenecks has been done at Boston University [1].

Another way to improve performance is by removing overhead in the protocol itself. The W^3C is currently standardizing HTTP/1.1 that enables multiple requests over a single connection. This "connection-caching" strategy can significantly enhance the performance over HTTP/1.0 [21, 13]. The need for persistent connections to improve latency was noted by Mogul in [11]. Latency also can be improved by using caching proxies and caching clients, although the removal policy needs to be carefully considered [17]. Yeager and McGrath of NCSA discuss many of these issues in [12].

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 was to produce the fastest server possible *for a given*

[8] Solaris provides `nscd(1M)`, a name service cache daemon. When this has cached the password file information, the request takes only around 100 μs. However, the first look up is still expensive.

Strategy	Advantages	Disadvantages
Single Threaded	No context switching overhead. Highly portable.	Does not scale for multiprocessor systems.
Process-per-request	More portable for machines without threads.	Creation cost high. Resource intensive.
Process pool	Avoids creation cost.	Requires mutual exclusion in some operating systems.
Thread-per-request	Much faster than fork.	May require mutual exclusion. Not as portable.
Thread pool	Avoids creation cost.	Requires mutual exclusion in some operating systems.

Table 4: Summary of Concurrency Strategies

hardware/OS platform configuration.

In the first stage of gaining this understanding, we analyzed the performance of existing servers. The servers that performed poorly were analyzed to discover sources of bottlenecks. The servers that performed well were analyzed more closely using white-box 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 in dispatching the request to the protocol handler that processes the request. The time spend dispatching is heavily dependent on the choice of the concurrency strategy.

Each concurrent strategy has positive and negative aspects, which are summarized in Table 4.[9] Thus, to optimize performance, Web servers should be *adaptive*, *i.e.,* be customizable to utilize the most beneficial strategy for particular traffic characteristics, workload, and hardware/OS platforms. In addition, workload studies indicate that the majority of the requests are for small files. Thus, Web servers should adaptively optimize themselves to provide higher priorities for smaller requests. These techniques combined could potentially produce a server capable of being highly responsive and maximizes throughput. The next generation of the JAWS server plans to implement the prioritized strategy.

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] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of CIKM'95: The Fourth ACM International Conference on Information and Knowledge Management*, November 1995.

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

[5] J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing... Multithreading the SunOS Kernel. In *Proceedings of the Summer USENIX Conference*, San Antonio, Texas, June 1992.

[6] 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/.

[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] 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.

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

[10] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison Wesley, 1996.

[11] Jeffrey C. Mogul. The Case for Persistent-Connection HTTP. Technical Report 95/4, Digital Equipment Corporation Western Research Laboratory, May 1995. Available online at http://www.research.digital.com/.

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

[13] 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. Available from http://www.w3.org/.

[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.

[9] The disadvantage of the pool approaches have to do with `accept`. In Solaris 2.5, `accept` is not a system call, and is not atomic. In BSD 4.4, `accept` is a system call. More information is available in [19].

[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] Pure Software. *Quantify User's Guide*, 1996.

[17] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdhulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of ACM SIGCOMM'96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26, October 1996.

[18] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, Reading, Massachusetts, 1992.

[19] W. Richard Stevens. *UNIX Network Programming, Second Edition*. Prentice Hall, Englewood Cliffs, NJ, 1997.

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

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

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