Creating a Framework for Developing High-performance Web Servers over ATM

Id: aa-main.tex,v 1.3 1997/02/20 20:14:44 jxh Exp

James C. Hu

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

1 Introduction

1.1 Motivation

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. The central thesis of this proposal is that to achieve optimal performance, Web servers must be able to dynamically adapt to various conditions, such as machine load, the type of incoming requests, and the rate of the requests. This claim is substantiated by the following results:

• **Concurrency strategy affects performance:** This proposal presents results from systematic performance analysis of existing Web server over a high-speed ATM network. The 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. The analysis revealed that if 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 the *concurrency strategy* used to process client requests.

• **JAWS is faster:** The proposal describes the design and implementation of my own high-performance Web server, JAWS. JAWS is a multi-threaded Web server that was explicitly designed to alleviate the performance bottlenecks identified in existing Web servers. JAWS consistently outperforms every server in the test suite and can service hundreds of requests per second during peak usage over a high-speed ATM network.

The outstanding performance of JAWS is due to several factors. First, by analyzing the performance measurements from our benchmarks, and understanding why some servers performed better than others, this knowledge was then applied to the development of JAWS. The techniques learned were: caching home directory information and caching file information. In other cases, strategies were combined, such as pre-spawning from Apache and multi-threading from PHTTPD. These techniques have been codified as general design principles and concurrency patterns for building high-performance Web servers. Second, the flexible object-oriented design of JAWS allowed these strategies to be incorporated easily.

The central research problem I am addressing is how to build Web servers that provide the best possible performance over high-speed networks. High-speed networks remove the network as a source of performance overhead. My research has thus far provided the following contributions:

- The identification of *concurrency strategy* as a factor to performance, and which concurrency strategies are most effective for common use cases.
- A methodology of *measurement driven refinement* that allows a competitive prototype to be developed quickly.
- Design principles and concurrency patterns for building high-performance Web servers.
- An adaptive concurrent server design, which can adjust its run-time concurrency behavior to provide optimal performance for particular traffic characteristics and workloads.

1.2 Research Proposal

While the preliminary results were obtained by developing a high-performance Web server, the goal of my research is to enable the construction of high-performance Web systems that are tailored for high-speed networks. A Web system denotes a broad class of server applications, including simple protocols, such as ftp, or more complex applications, such as a video-on-demand server, or meta applications, such as a super-server (*i.e.*, `inetd`). The goal of my research is to enable server developers to build high-performance servers by only implementing the protocol definition, such as a grammar. The framework will provide the appropriate I/O and concurrency mechanisms, and dynamically adapt to offer optimal performance on the given architecture. Thus, my work will be to provide a basis for this framework, which involves an abstraction for concurrency strategies and an abstraction for protocol construction and configuration.

As shown in Section 2, existing research on improving Web performance has focused largely on reducing network latency, primarily through caching techniques [32] or protocol optimizations [17, 20, 4]. Moreover, measurements of server performance have focused largely on low-speed networks [1, 15]. However, I am unaware of any publication, aside from unofficial accounts [22], that describes the design principles and techniques necessary to develop high-performance Web servers. To my knowledge, there is no related work on achieving Web server performance over high speed networks.

The approach to performance I will pursue relies on the ability of the Web server to adapt its concurrency strategy and protocol handling to changing conditions. The conditions to

which adaptation may be applied include workload, memory availability, bandwidth and CPU utilization. The efficacy of this approach will be demonstrated by instrumenting adaptation into the JAWS server, and by conducting benchmarks, comparing JAWS to other high-performance servers.

The remainder of this proposal is organized as follows: Section 2 discusses related work; Section 3 describes the existing ATM testbed used to benchmark Web servers and presents the results of these benchmarks and my analysis; Section 4 outlines the object-oriented design of JAWS and compares the performance of JAWS with the Web servers evaluated in Section 3; Section 5 lists the objectives to be accomplished for fulfillment of the doctoral degree, and a timetable by which to guide progress made towards the degree; and Section 6 presents concluding remarks.

2 Related Work

2.1 Background

Before the Internet, there was ARPANET, a network formed during the late 1960's comprising of a small number of CS departments. The networks then were slow, less than 57.6 Kbps. The TCP/IP layered protocol provided the functionality necessary for interconnections and communication between hosts. The three services provided were FTP for file transfer, SMTP for electronic mail, and TELNET for remote login [37]. These services were layered on top of TCP/IP, but the slow networks hid the costs of pushing data through every protocol layer. As the speed of networks has increased, due to the advent of fiber optics and high bandwidth switches [38], it became clear that layered protocol handling was a bottleneck, due to the "data copying" problem.

Solutions to this problem have included protocol delayering, protocol parallelizing, and using specialized hardware. Protocol delayering is the process where several layers of a protocol stack are implemented as a single layer (or eliminated completely as in [2]), thus avoiding some data copying overhead. Protocol parallelizing implements protocol handling by processing multiple layers of the protocol concurrently [30], *e.g.*, marshalling and demarshalling (presentation layer), connection multiplexing (transport layer), and directory lookup (application layer). Specialized hardware, which understands the nature of the protocol, can be used to avoid protocol processing by the CPU altogether, and handle the protocol without passing any data to memory in user space (zero data copies) [7].

Each of these solutions can improve overall performance. It may be impractical, however, to substantially alter the network device drivers or create new hardware for every new protocol that may arise, such as HTTP/99.3. Further, the techniques do not directly address the issue of adaptive protocol processing. However, the general principles of delayering and parallelizing may be applied to optimize the processing of application level protocols by the application itself.

2.2 Performance Evaluation

Issues concerning server performance have been dicussed in other server applications, the most relevant being NFS. The issues surrounding NFS such as reliability and cache co-

herency are discussed in detail in the literature [19, 23]. To measure the performance of filesystems, the Standard Performance Evaluation Corporation (SPEC) designed the LADDIS benchmark [40].

The need for *scalable* Web servers, *i.e.* servers which can be scaled up to handle increasingly larger capacities, was observed even in the early stages of the World Wide Web. The need was noted in [13], where they demonstrate a scalable decentralized distributed server. The need for high-capacity servers spurred commercial sector activity, and many server implementations are available on the market [35].

The growing number of available servers prompted the need for assessing their relative performance. The current standard benchmarks available are WebStone (by SGI) and SPECweb96 (by SPEC), both heavily influenced by the design of LADDIS [10, 5]. Both WebStone and SPECweb96 attempt to measure overall performance, and rate the performance of a server with a single number, where a higher number indicates better performance.

In addition to gauging the relative performance of Web servers, studies have been done to characterize the Web server workload and performance overheads. Workload has been analytically modeled at the University of Saskatchewan [14], but similar results were observed by Trent [10]. Additional performance tools have been developed at Boston University to analyze the Web server performance overheads of Apache on a Pentium-based machine [1].

2.3 Optimization Strategies

One of the purposes for performance measurements is to discover areas for optimization in order to improve performance. One way to improve performance is by removing overhead in the protocol itself. The W^3C is currently standardizing HTTP/1.1, which multiplexes multiple requests over a single connection. This "connection-caching" strategy can significantly enhance the performance over HTTP/1.0 by reducing unnecessary connection set up and termination [36, 20]. The need for persistent connections to improve latency was noted by Mogul in [17]. Latency can also be improved by using caching proxies and caching clients, although the removal policy must be considered carefully [32].

2.4 Web Server Design and Implementation

Design patterns are re-usable software abstractions that have been observed to occur many times in actual solutions [9]. A design pattern is intended to solve a general design problem for a specific context. Many patterns have been observed in the context of concurrent, parallel and distributed systems [29, 6, 26]. Many of these ideas are applicable to Web server design.

Katz presents an NCSA prototype of a scalable Web server design [13]. This design was prompted by the growing number of requests being made to the NCSA server machine. Many commercial server implementations arose to meet the demand for high-performance Web servers. Higher end implementations, such as Netscape Enterprise and Zeus, use multithreading to scale for high-end architectures with multiple CPUs. Other implementations (*e.g.*, Roxen, BOA and thttpd) use a single thread of control to optimize performance on single CPU architectures. An informal discussion about Web server performance issues can be found in [22]. Yeager and McGrath of NCSA discuss the tradeoffs of some Web server designs and their performance in [18].

2.5 Concurrency Strategies

A concurrency strategy specifies the policy and mecahnism for allowing multiple tasks to execute simultaneously. In the case of a server, as task is an object which encapsulates the handling of server requests. There are four general classifications for a concurrency strategy: iterative (or sequential), single-threaded concurrent, thread-per-request, and thread pool.

- An iterative server is a server that has no application-level concurrency: each task that begins runs to completion.
- A single-threaded concurrent server has one thread of control, but only partially serves each unfinished task until a pre-determined amount of work is done, or until a timer has expired.
- The thread-per-request strategy allows each task to execute in its own thread of control.
- A thread pool approach fixes the number of executing threads, and assigns tasks to available threads.

The use of these strategies are prevalent enough to have been identified as patterns. For example, the solution to designing a single-threaded concurrent server is to apply the reactor pattern [27]. Surprisingly, the literature lack any discussion of the relative performance between concurrency strategies. This may be due to the difficulty of decoupling the concurrency strategy from the rest of server implementation. Various sources discuss the tradeoffs between multi-threaded and single-threaded solutions [12, 28]. These are understood to be simplicity of design and implementation and overlapped I/O versus synchronization and context switching overhead.

3 Experimental Results of Web Server Performance over ATM

3.1 Web Server/ATM Testbed Environment

The performance measurements were done in collaboration with Sumedh Mungee. Sumedh implemented the client used for generating the Web traffic and conducted the benchmarking experiments over ATM.

3.1.1 Hardware and Software Platforms

The performance study of Web servers began 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 [39] (shown in Figure 1).[1]

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

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

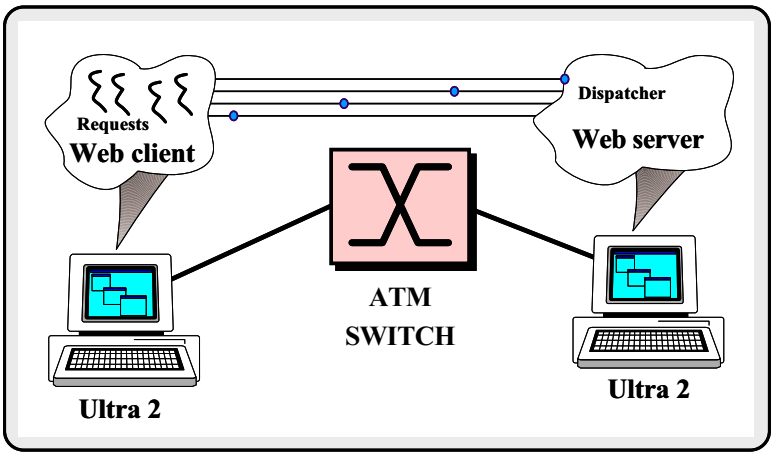

Figure 1: Web Server/ATM Testbed Environment

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 [21].

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 [11].

3.1.2 Client Traffic Generators

To provide a thorough and accurate understanding of server performance bottlenecks, I designed 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.

There were two motivations for this design:

• **Predictability:** The client traffic generator permits us to control the request rate per second to within 3 significant digits. For example, 2.15 requests per second corresponds to 129 requests per minute. Conventional benchmarking tools [10, 5] 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:** I wanted to examine the point at which the HTTP request rate dramatically alters Web server performance. This is achieved by running our rate-based client traffic generator over ATM. Holding network and file I/O constant (by using memory-cached files) allowed me to precisely pinpoint the request rates where a server starts to overload. These results help to create an accurate behavioral model for Web servers.

The client traffic generator is based on the ACE network programming toolkit [25], 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, 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 ACE.

3.1.3 Performance Metrics

Two *black-box* metrics, *throughput* and *latency*, were included in the 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 [31]. 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 Sections 3.2.2 and 3.2.3 and white-box analysis is presented in Section 3.2.4.

3.1.4 Web Servers

The choice of Web servers for the test suite was 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 [3, 15, 35]. 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 testing were Apache, PHTTPD, Roxen, Jigsaw, Netscape Enterprise and Zeus. In addition, my own Web server, JAWS, was benchmarked in the experiment, as well. Section 4 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.

3.2 Performance Results

3.2.1 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 my 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 [14, 5]. 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 a 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 3.1.3. 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.

3.2.2 Web Server Performance for Large Files

• **Description:** The parameters for this experiment are outlined in Table 1. The results

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

Table 1: Large File Experiment Parameters

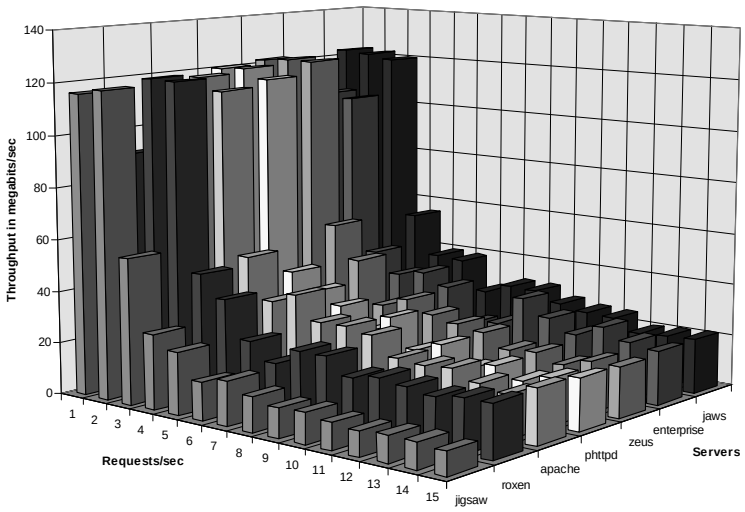

Figure 2: Throughput for Large File Transfers

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.

• **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 exponentially (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 4 explains the optimizations we used to achieve this performance.

3.2.3 Web Server Performance for Small Files

• **Description:** The parameters for this experiment are outlined in Table 2. Once again,

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

Table 2: Small File Experiment Parameters

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.

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

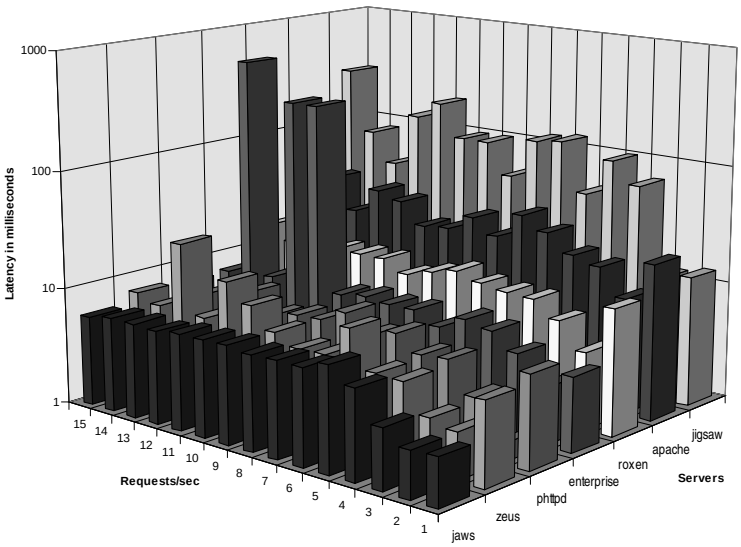

Figure 3: Latency for Large Files Transfers

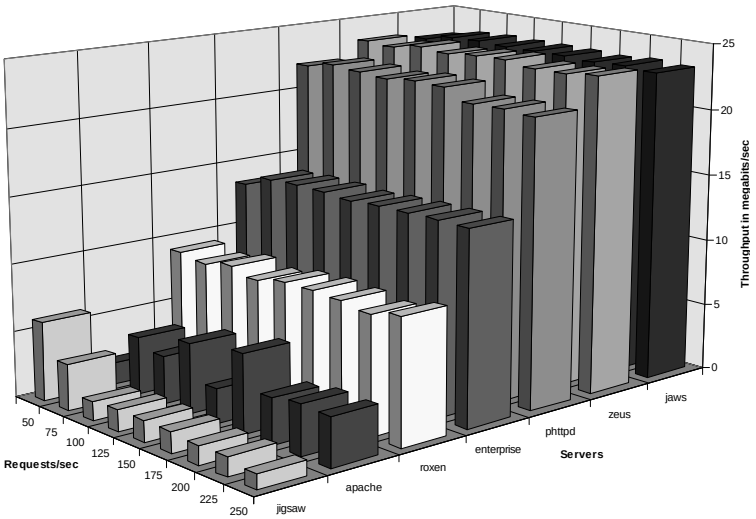

Figure 4: Throughput for Small File Transfers

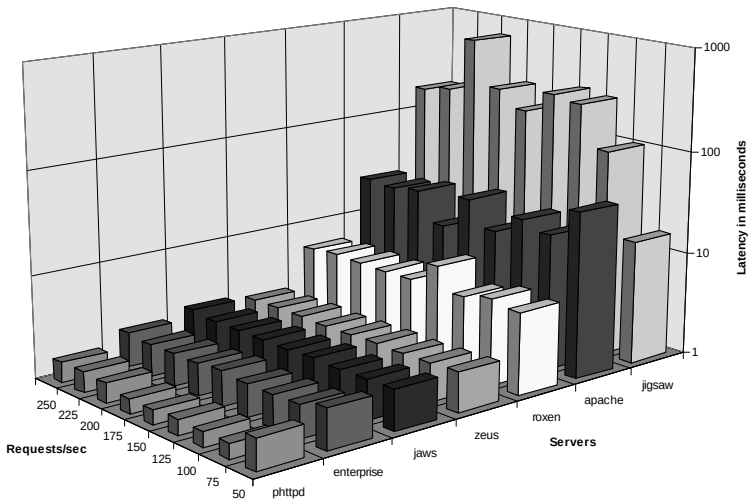

Figure 5: Latency for Small Files Transfers

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

the throughput of the JAWS and Zeus servers.[2]

3.2.4 White-box Analysis of Web Server Performance

• **Description:** In this experiment, the Apache server was instrumented with Quantify [31]. Apache was then subjected to workloads identical to those described in Section 3.2.2 and Section 3.2.3. For the purposes of this presentation, only Apache 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.

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

The fourth column suggests optimization techniques that can alleviate the performance bottlenecks, some of which have been implemented in JAWS.

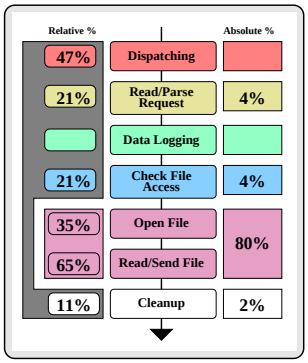

Figure 6: Typical HTTP Request Life-cycle

• **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 information about the request into a logfile.[3] 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.

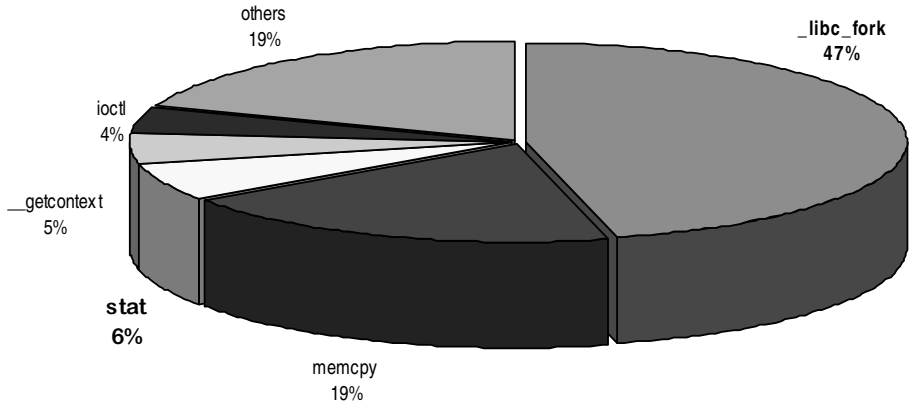

Figure 7: Performance Overhead in Apache, Excluding I/O

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

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

picture, dispatching comprises nearly 50% of the remaining sources of overhead, as shown in Figure 7.

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

3.2.5 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 software flexibility and modularity.

• **I/O bound processing:** To achieve maximum utilization of network resources, servers typically need only attempt to fill the bandwidth capacity of the network interface. However, if the network is not the bottleneck, as is the case with the ATM testbed, then time should not be wasted in opening and reading the file on the disk. Figure 8 shows that Apache spends 30% of its I/O processing in opening and reading the file into memory. This overhead could be eliminated if the entire file is cached.

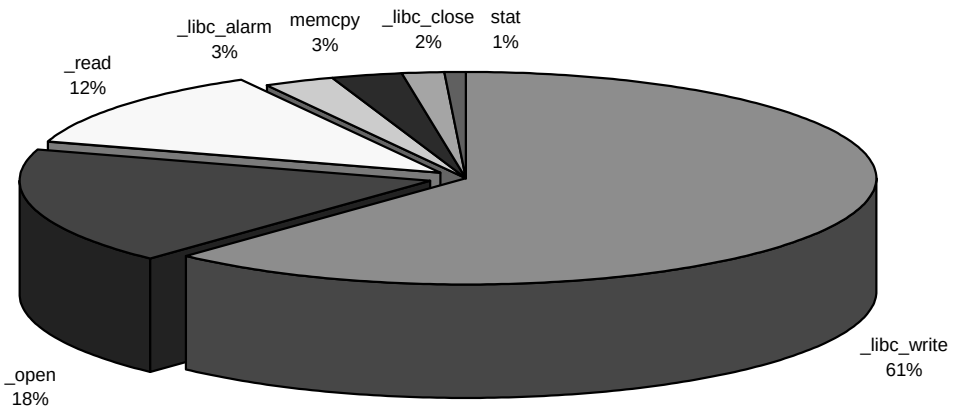

Figure 8: I/O Performance Overhead in Apache

• **Concurrency strategies:** For single CPU machines, single-threaded solutions are acceptable and perform well.[4] 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

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

the processes before requests arrive. This improves performance over ATM, but does not perform as well as threads on Solaris.

PHTTPD, a freely available 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[5], but detailed information about their concurrency strategies are not available since they are only available in binary form.

4 Principles for Developing Extensible, High-performance Web Servers

As shown in Section 3.2, JAWS consistently outperforms the other servers in our test suite. During the 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:

• **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, such as as PHTTPD and JAWS, that implemented sophisticated caching strategies performed much better than those that did not, such as Apache and Jigsaw.

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

4.1 The Object-Oriented Architecture of JAWS

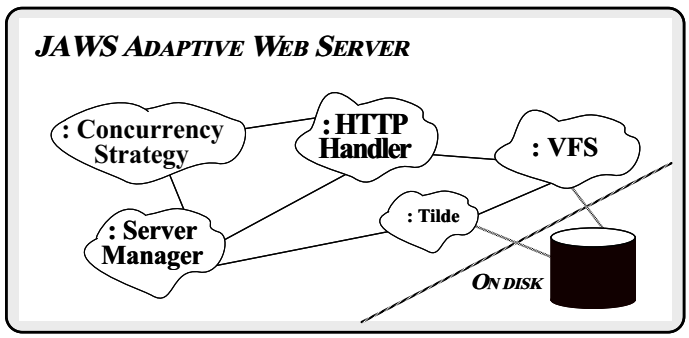

Figure 9: The Object-Oriented Software Architecture of JAWS

Figure 9 illustrates the object-oriented software architecture of the JAWS Web server. The black-box and white-box measurements in Section 3.2 indicated that the concurrency strategy used by a Web server largely determine its performance. Therefore, JAWS is designed

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

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, workload characteristics, and the number of available 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 JAWS. 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. These strategies are discussed in Section 4.2.1.

• **Server manager:** This component associates the HTTP handler (or any other protocol handler) with the Concurrency strategy. It also implements a small protocol that directs the manager to dynamically re-link portions of the server code when they are 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 incur filesystem overhead.

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.

4.2 Principles for High-performance Web Server Development

4.2.1 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 type of server does not perform well except in very low load servers [34]. 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

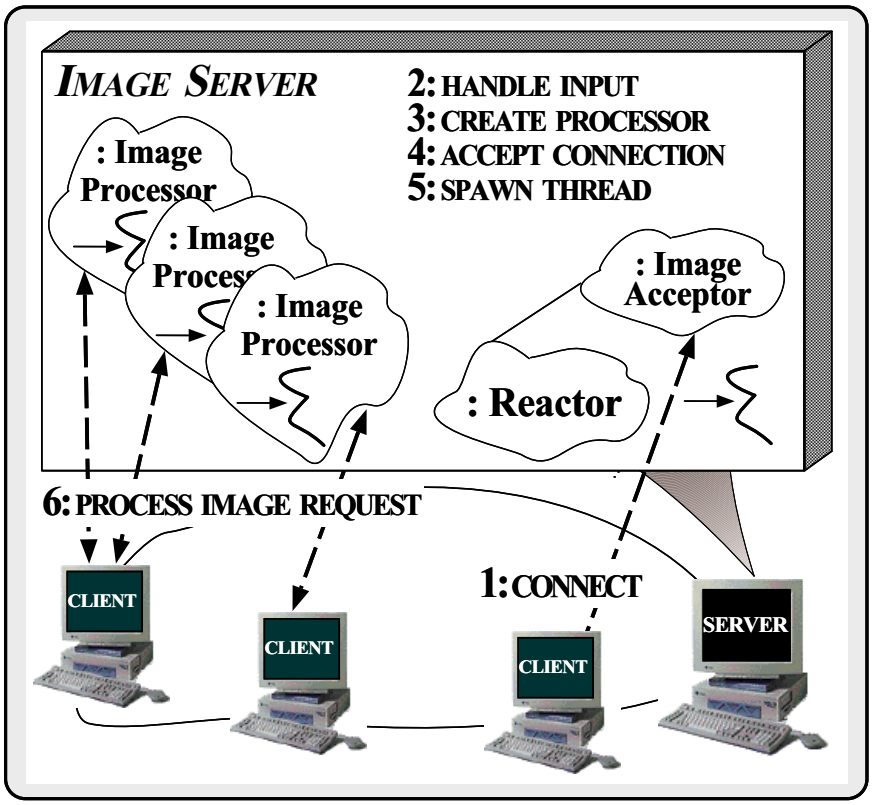

Figure 10: Illustration of the thread-per-request strategy

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 3.2). Figure 10 illustrates this strategy in the context of an image server.

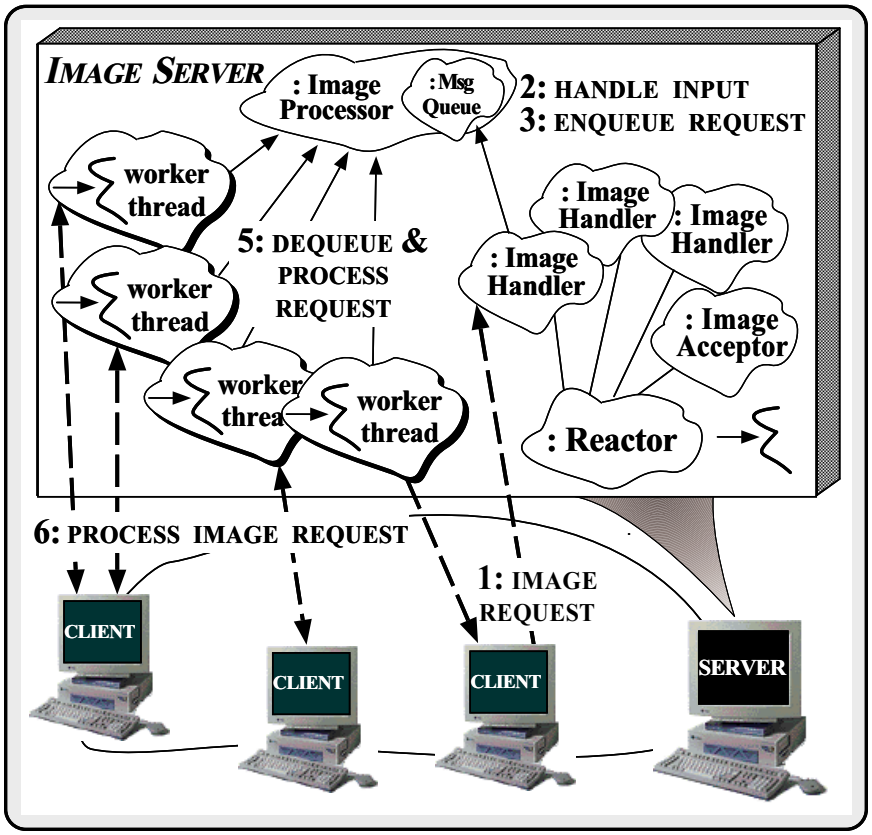

Figure 11: Illustration of the thread pool strategy

• **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*.[6] 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 [8]. This cost becomes negligible, however, when pre-spawned thread pools are used.

Figure 11 provides an illustration of this strategy. In this case, an image server queues requests into a message queue, from which free threads from the pool can dequeue a request and begin processing.

• **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 programming with error-prone I/O interfaces (*e.g.* `select`) and/or asynchronous I/O programming. The potential payoff, however, is a faster server for uni-processors.

4.2.2 Avoiding the Filesystem

• **Virtual file system:** Many Web servers transmit files 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 [33]. 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 virtual memory management hardware.

Although memory-mapped file I/O appears an ideal, problems arise for highly loaded multi-threaded Web servers when many clients 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.

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

To avoid this problem, JAWS implements a *Caching Virtual Filesystem*, 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 user's home directory. Web clients typically request these files with a URI which prefixes the file with `/~user/`, to denote the home directory of `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 the `Quantify` measurements in Section 3.2.4, 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.[7]

The solution used in JAWS is 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 [24] 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 [16].

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`.

[7] 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.

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.

5 Objectives

The demonstration of JAWS' performance is evidence that applying design principles and concurrency patterns can yield a high-performance Web servers. The research goal is to provide a framework which would take the difficulty of discovering these principles out of the development of a high-performance server. Instead, only the protocol would need to be implemented, and the framework would dynamically adopt the optimum concurrency strategy, and apply any needed caching to avoid potential sources of overhead, such as the filesystem.

In order to achieve this goal, a number of tasks are required:

1. Further decouple concurrency from protocol: The decoupling exists, but is not very high. The current interface depends on the protocol handler to understand whether the I/O model of the strategy is synchronous (*e.g.* has its own thread of control), or is asynchronous (*e.g.* must be willing to surrender control at any moment).

This task requires the design of an interface between the concurrency strategy and protocol which remains invariant to the I/O policy (synchronous or asynchronous).

After this decoupling is complete, I could begin to implement the other concurrency strategies, and enhance the current HTTP implementation.

2. Seek out new design principles: The principles we have discovered helped create JAWS, but white-box analysis was only applied to Apache. To discover other potential design principles, additional white-box analysis is required on a wider range of servers.

Also, further benchmarking is needed, *i.e.*, benchmark new servers, measure other performance parameters, realistic workload cases and use standard benchmarkers. Benchmarking new Web servers would provide additional insight into the server design versus performance issue. Additional performance parameters, server throughput and client jitter, would provide additional information about over server performance. Server throughput would be an indicator of server efficiency, and client jitter would predict if the server is suitable for use for applications with QoS guarantees, such as a video stream. Applying the benchmarks on more realistic workload cases would serve as a more practical indication of Web server performance. Applying standard benchmarks wil verify the soundness of my own benchmarking techniquem, and make JAWS' performance more accessible to a broad technical and non-technical audience.

3. Apply additional optimizations: There are additional strategies that have not yet been incorporated into JAWS. These include prioritized request handling, and parallelized protocol processing.

- Workload studies have indicated that the most frequently requested files from a Web server are small (less than 10 Kbytes) [14, 5]. 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.

- It is possible to parallelize some of the processing of the HTTP protocol. For instance, if logging is enabled, the this could be done in a separate thread in parallel with the reading and sending of the requested file.

Each optimization will be separately benchmarked to demonstrate performance gains.

4. Create protocol abstractions: In order to more easily incorporate new protocols into the server, I intend to create a protocol abstraction which would more easily allow the server writer utilize the I/O models and concurrency strategies of the framework by making them more transparent.

To optimize overall performance, it may be necessary for the client to request the server to apply computations to the requested file before delivering it to the client, such as file decompression. In order to be able to negotiate protocol processing between client and the server, the server protocol abstraction should be accompanied with a client protocol abstraction which can engage in negotiation.

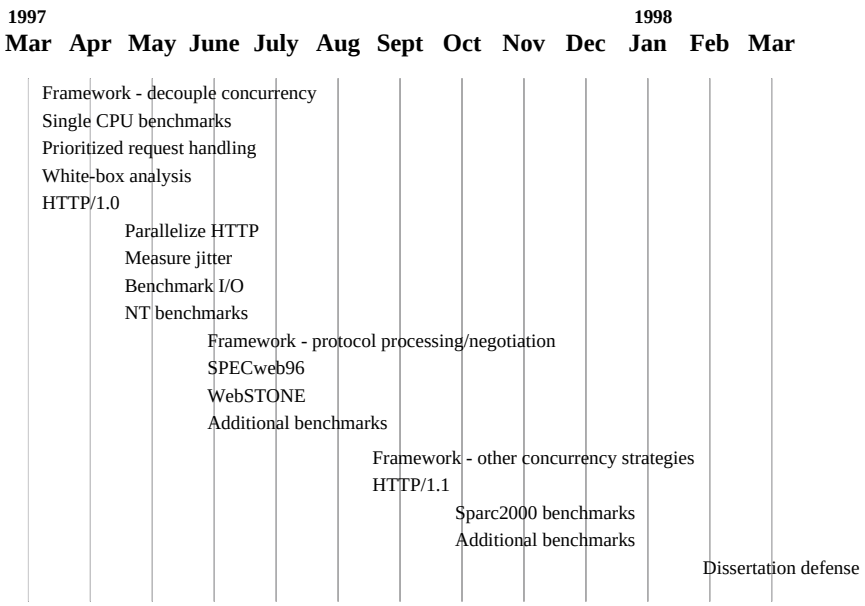

Figure 12: Objectives timeline for progress toward dissertation

The intended delivery dates for these tasks have been summarized in Figure 12.

6 Concluding Remarks

The research presented in this proposal 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, the research goal is to produce the fastest server possible *for a given hardware/OS platform configuration*

In the first stage of gaining this understanding, I 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 spent dispatching is heavily dependent on the choice of the concurrency strategy.

These results were utilized in the creation of JAWS. Each revision of JAWS was subjected to a new round of benchmarks until the resulting version began to consistently outperform the other Web servers in the test suite. I have found the measurement driven refinement to be a valuable tool in producing a competitive implementation prototype. The ideas used in the development of JAWS resulted in the formulation of design principles and concurrency patterns for building high-performance Web servers.

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

Each concurrent strategy has positive and negative aspects, which are summarized in Table 4.[8] 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 order to help server designers and implementors create adaptive servers, I plan to refine the JAWS server with abstractions for concurrency and for protocol specification. Hopefully, these abstractions would provide a solid foundation for a general *adaptive server framework*, which would not only be able to choose the right concurrency mechanism for a given circumstance, but also be able to choose the right protocol processing mechanism as well.

[8]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 [34].

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] M. Stella Atkins, Samuel T. Chanson, and James B. Robinson. LNTP – An Efficient Transport Protocol for Local Area Networks. In *Proceedings of the Conference on Global Communications (GLOBECOM)*, pages 705–710, 1988.

[3] Anselm Baird-Smith. Jigsaw performance evaluation. Available from http://www.w3.org/, October 1996.

[4] 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.

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

[6] Steve Crane, Jeff Magee, and Nat Pryce. Design Patterns for Binding in Distributed Systems. In *The OOPSLA '95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*, Austin, TX, October 1995. ACM.

[7] Zubin D. Dittia, Jr. Jerome R. Cox, and Guru M. Parulkar. Design of the APIC: A High Performance ATM Host-Network Interface Chip. In *IEEE INFOCOM '95*, pages 179–187, Boston, USA, April 1995. IEEE Computer Society Press.

[8] 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.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[10] 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/.

[11] 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.

[12] Tim Harrison and Douglas C. Schmidt. Thread-Specific Storage: A Pattern for Reducing Locking Overhead in Concurrent Programs. In *OOPSLA Workshop on Design Patterns for Concurrent, Parallel, and Distributed Systems*. ACM, October 1995.

[13] Eric Dean Katz, Michelle Butler, and Robert McGrath. A Scalable HTTP Server: The NCSA Prototype. In *Proceedings of the First International Conference on the World-Wide Web*, May 1994.

[14] 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.

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

[16] 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.

[17] 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/.

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

[19] Bill Nowicki. Transport Issues in the Network File System. *ACM Computer Communication Review*, 19(2):16–20, April 1989.

[20] 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/.

[21] Dennis Ritchie. A Stream Input–Output System. *AT&T Bell Labs Technical Journal*, 63(8):311–324, October 1984.

[22] James Rubarth-Lay. Keeping the 400lb. Gorilla at Bay: Optimizing Web Performance, May 9 1996. Class paper for UT Austin LIS385T.6, Spring 96.

[23] R. Sandberg. The Sun Network Filesystem: Design, Implementation, and Experience. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

[24] 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.

[25] 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.

[26] Douglas C. Schmidt. Flexible configuration of high-performance object-oriented distributed communication systems. In 9^{th} OOPSLA Conference, Workshop on Flexibility in Systems Software, Portland, Oregon, October 1994. ACM.

[27] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching. In *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, pages 1–10, Monticello, Illinois, August 1994.

[28] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

[29] Douglas C. Schmidt and Charles D. Cranor. Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O. In James O. Coplien, John Vlissides, and Norm Kerth, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1996.

[30] Douglas C. Schmidt and Tatsuya Suda. The Performance of Alternative Threading Architectures for Parallel Communication Subsystems. *Journal of Parallel and Distributed Computing*, 1997. to appear.

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

[32] 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.

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

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

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

[36] 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/.

[37] Andrew S. Tanenbaum. *Computer Networks (Second Edition)*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[38] Jonathan Turner. Design of a Broadcast Packet Switching Network. *IEEE Transactions on Communication*, 4(8), June 1988.

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

[40] Mark Wittle and Bruce E. Keith. LADDIS: The Next Generation in NFS File Server Benchmarking. In *USENIX Association Conference Proceedings '93*, April 1993.