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

James C. Hu and Douglas C. Schmidt

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

1 Introduction

During the past two years, the volume of traffic on the World Wide Web (Web) has grown dramatically. Traffic increases are due largely to the proliferation of inexpensive and ubiquitous Web browsers (such as NCSA Mosaic, Netscape Navigator, and Internet Explorer). Likewise, Web protocols and browsers are increasingly being applied to specialized computationally expensive tasks, such as imaging servers (*e.g.*, Siemens MED and Kodak Picture Net) and database search engines (*e.g.*, AltaVista and Lexus Nexus).

To keep pace with increasing 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 and network congestion, the type of incoming requests, and the rate of requests. Therefore, we propose to develop an adaptive Web server framework and use it to empirically determine (1) the performance impact of different server design and implementation strategies, (2) the scalability of servers under high-load conditions, and (3) the pros and cons of different design alternatives.

1.1 Motivation

Web servers are synonymous with HTTP servers and the HTTP 1.0 and 1.1 protocols are relatively straightforward. HTTP requests typically name a file and the server locates the file and returns it to the client requesting it. On the surface, therefore, Web servers appear to have few opportunities for optimization. This may lead to the conclusion that optimization efforts should be directed elsewhere (such as transport protocol optimizations [11], specialized hardware [5], and client-side caching [21, 12]).

Empirical analysis reveals that the problem is more complex and the solution space is much richer. For instance, our experimental results show that a heavily accessed Apache Web server (the most popular server on the Web today [20]) is unable to maintain satisfactory performance on a dual-CPU 180 Mhz UltraSPARC 2 over a 155 Mbps ATM network, due largely to its choice of process-level concurrency [7]. Other studies [8] have shown that the relative performance of different server designs depends heavily on server load characteristics (such as hit rate and file size).

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

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

This proposal is motivated by our empirical results [8, 7] that illustrate why no single Web server configuration is optimal for all circumstances. Based on these results, we hypothesize that optimal Web server performance requires both *static* and *dynamic* adaptive behavior.

Static adaptivity allows a Web server to bind common operations to high performance mechanisms provided by the native OS (*e.g.*, Windows NT 4.0 support for asynchronous I/O and network/file transfer). Programming the server to use generic

OS interfaces and APIs is insufficient to provide maximal performance across OS platforms. Therefore, asynchronous I/O mechanisms in Windows NT and POSIX must be studied, compared, and tested against traditional concurrent server programming paradigms that utilize synchronous event demultiplexing and threading.

Dynamic adaptivity allows a Web server to alter its run-time behavior "on-the-fly." This is useful when external conditions have changed to a point where the configured behavior no longer provides optimal performance. Such situations have been observed in [8] and [9].

Research on adaptive software has not been pursued deeply in the context of Web systems. Current research on Web *server* performance has emphasized caching [12, 21], concurrency, and I/O [13]. While our results indicate that caching is vital to high performance, non-adaptive caching strategies do not provide optimal performance in Web servers [9]. Moreover, current server implementations and experiments rely on *statically* configured concurrency and I/O strategies. As a result of our empirical studies, we conclude that servers relying on static, fixed strategies cannot behave optimally in many high load circumstances.

1.2 Preliminary Results

To understand the advantages and disadvantages of different server optimization strategies in the context of Web servers the following research questions must be addressed:

- What types of server load conditions (*e.g.*, machine load, disk I/O, context switching and memory paging) affect server performance?
- What techniques can be employed to improve Web server performance, and what are the impact on performance and predictability?
- How are optimization techniques affected by changing server load conditions and what combinations of techniques yield better performance under various server load conditions?
- How can optimal Web server performance be achieved in the face of changing server load conditions?

To address these research questions, we have developed a prototype high-performance Web server called JAWS.[1] We have compared JAWS' performance with existing Web servers. Thus far, our research has provided the following contributions:

- The identification of *concurrency* and *event dispatching* strategies as key determinants of performance, and which strategies are most effective for common use cases.
- Empirical data showing the extent to which the performance of concurrency, event dispatching, and I/O strategy combinations depend heavily on *network and server load conditions*.
- A methodology of *measurement driven refinement* that allows high performance prototypes 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.

The experimental JAWS Web server software platform performs as well as (and in many cases outperforms) existing Web server implementations. These results are discussed in more detail in [7] and [8].

1.3 Research Proposal Synopsis

Our preliminary results were obtained by developing a high-performance Web server prototype. However, the ultimate goal of our research is to determine how to construct and configure Web systems that can dynamically adapt to changing workloads to provide the best possible performance over high-speed networks. Naturally, it is always possible to improve performance with more expensive hardware (*e.g.*, additional memory and faster CPUs) and a more efficient OS. However, our research aims to produce the fastest server possible *for a given hardware/OS platform configuration*.

Moreover, a web system is representative of large class of server applications. These range from simple protocols (such as ftp) to complex distributed applications (such as electronic medical imaging systems [14], video-on-demand servers [5], or databases). Therefore, the results we obtain in our work will generalize to many other high-performance server use-cases.

We propose to apply and measure a broad range of optimization techniques. Empirical analysis will determine the impact of each technique under different end-system and network loads. From the analysis, we will develop *profiles* that indicate which strategies are most effective for various conditions, including workload, memory availability, bandwidth, and CPU utilization.

The goal of our research is to demonstrate that optimal performance can be achieved by developing Web servers that can adapt their concurrency strategies, I/O, caching, and protocol handling to changing conditions. To validate our hypotheses, we will enhance the prototype JAWS server to support adaptive strategies and conduct benchmarks that compare its performance to other high-performance Web servers.

[1] JAWS is available at `www.cs.wustl.edu/~schmidt/ACE.html`.

To facilitate technology transfer, we will develop an application framework that enables server developers to build high-performance servers by implementing only the protocol definition. This framework will provide the appropriate I/O and concurrency mechanisms and adapt statically and dynamically to offer optimal performance on the given architecture, by utilizing the empirically determined performance profiles.

The remainder of this proposal is organized as follows: Section 2 outlines the object-oriented design of JAWS, our high-performance Web server prototype; Section 3 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 4 presents concluding remarks.

2 Strategies for Developing High-performance Web Servers

As shown in [7], JAWS consistently outperforms the other servers in our test suite. These servers included Apache, PHTTPD, Roxen, Netscape Enterprise Server, Zeus, W^{3}C Jigsaw, and JavaServer. 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 and event dispatching strategies:** Since dispatching occupies a large portion of non-I/O related Web server overhead, choosing the right strategies 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).

We then performed additional experiments to characterize the relative impacts of different I/O models coupled with different concurrency strategies under various loads on Windows NT 4.0 over a 155 Mbps ATM network [8]. These experiments revealed the following results:

• **Throughput is highly sensitive to I/O strategy and file size:** Synchronous I/O mechanisms achieved higher throughput than asynchronous I/O for smaller files. For larger files, the Windows NT system call `TransmitFile` outperformed the other I/O models by a factor of two. However, synchronous I/O coupled with the thread pool strategy consistently outperformed asynchronous I/O coupled with the thread pool strategy. For larger files, the performance between synchronous I/O with thread-per-request and synchronous I/O with thread pool converged.

• **Latency is highly sensitive to hit rate:** For smaller files, the synchronous thread pool gives consistently better performance. For larger files, `TransmitFile` provides better latency under light loads. `TransmitFile` has significantly higher latency under heavier loads.

These results demonstrate the performance variance that occurs as a Web server experiences changing load conditions. Thus, performance can be improved by dynamically adapting the server behavior to these changing conditions. This section explores techniques that can achieve this.

Our technical overview of our proposal begins with an outline of the object-oriented (OO) design of JAWS. JAWS is our prototype adaptive Web server that was implemented to study the performance impact of different server design strategies. We then describe various strategies that we plan to implement, test, and evaluate to improve Web server performance under different situations.

2.1 The Object-Oriented Architecture of JAWS

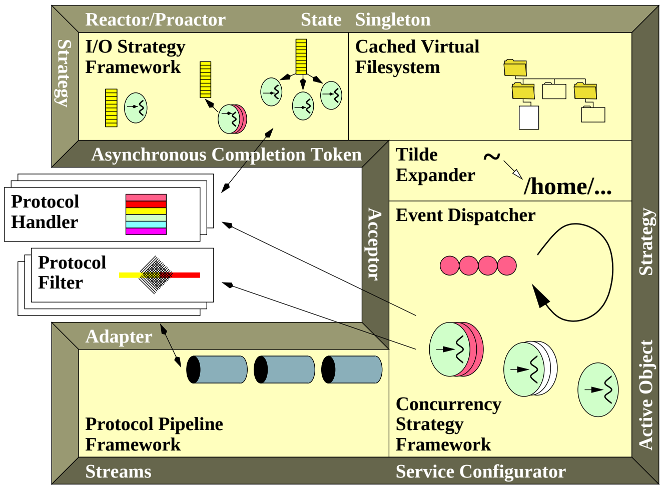

Figure 1: The Object-Oriented Software Architecture of JAWS

Figure 1 illustrates the OO software architecture of the JAWS Web server. As discussed above, concurrency strategies, event dispatching, and caching are key determinants of Web server performance. Therefore, JAWS is designed to allow Web server concurrency and event dispatching strategies to be customized in accordance with key environmental factors. These factors include traffic patterns, workload characteristics, support for kernel-level threading and/or asynchronous I/O in the OS, and the number of available CPUs.

JAWS is structured as a framework [18] that contains the following components: an *Event Dispatcher*, *Concurrency Strategy*, *I/O Strategy*, *Protocol Pipeline*, *Protocol Handlers*, and *Cached Virtual Filesystem*. Each component is structured as a set of collaborating objects implemented with the ACE

C++ communication framework [16]. Each component plays the following role in JAWS:

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

• **Concurrency Strategy:** This implements concurrency mechanisms (such as single-threaded, thread-per-request, or thread pool) that can be selected adaptively at run-time or pre-determined at initialization-time. These strategies are discussed in Section 2.2.1.

• **I/O Strategy:** This implements the I/O mechanisms (such as asynchronous, synchronous and reactive). Multiple I/O mechanisms can be used simultaneously. These strategies are discussed in Section 2.2.2.

• **Protocol Handler:** This object allows system developers to apply the JAWS framework to a variety of Web system applications. A *Protocol Handler* object is parameterized by a concurrency strategy and an I/O strategy, but these remain opaque to the protocol handler. In JAWS, this object implements the parsing and handling of HTTP request methods. The abstraction allows for other protocols (*e.g.*, HTTP/1.1 and DICOM) to be incorporated easily into JAWS. To add a new protocol, developers simply write a new *Protocol Handler* implementation, which is then configured into the JAWS framework.

• **Protocol Pipeline:** This component provides a framework to allow a set of filter operations to be incorporated easily into the data being processed by the *Protocol Handler*.

• **Cached Virtual Filesystem:** The component improves Web server performance by reducing the overhead of filesystem accesses. The caching policy is strategized (*e.g.*, LRU, LFU, Hinted, and Structured). This allows different caching policies to be profiled for effectiveness and enables optimal strategies to be configured statically or dynamically. These strategies are discussed in Section 2.2.3.

• **Tilde Expander:** This mechanism is another cache component that uses a perfect hash table [15] that maps abbreviated user login names (*e.g.* ∼schmidt to user home directories (*e.g.*, /home/cs/faculty/schmidt). When personal Web pages are stored in user home directories, and user directories do not reside in one common root, this component substantially reduces the disk I/O overhead required to access a system user information file, such as `/etc/passwd`.

In general, the OO design of JAWS decouples the strategies of a component from its functionality. For instance, the *Concurrency Strategy* can be decoupled from the *Protocol Handler*. Thus, a wide range of strategies can be supported, tested, and evaluated. As a result, JAWS can adapt to environments that may require different concurrency, I/O, and caching mechanisms.

2.2 Adaptive Web Server Strategies

The discussion below presents the concurrency, I/O and, caching strategies that JAWS will support and summarizes the various alternatives. The JAWS framework allows the different mechanisms to be changed easily, which enables different combinations and configurations. This flexibility supports the following research goals: (1) to empirically determine the appropriate profiles for different configurations by testing each configuration under a variety of system load conditions; and (2) to provide a re-usable framework that achieves optimal performance by allowing both *static* adaptations to Web server architectures and *dynamic* adaptations to changing Web server load conditions.

2.2.1 Adaptive Concurrency Strategies

Our experiments in [8] demonstrate how the choice of concurrency and event dispatching strategies significantly affect the performance of Web servers that are subject to changing load conditions. JAWS addresses this issue by allowing its concurrency strategy to adapt dynamically to current server conditions. Concurrency strategies vary in scope from single threaded, multiple processes, multiple threads, to distributed processes. For this research proposal, the focus is on threading strategies. This section describes various threading concurrency strategies and discusses their relative merits.

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

If the OS platform supports threads (and most modern operating systems do), thread-per-request is a more efficient concurrency strategy than process level concurrency. In our testbed, the cost of creating a thread on Solaris (with `thr_create`) is less than 260 μs (approximately 20 times faster than `fork`), and on Windows NT (with `CreateThread`) it is approximately 400 μs (approximately 10 times faster than `CreateProcess`). PHTTPD uses a thread-per-request strategy and exhibits excellent performance [7].

• **Thread pool:** In the *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 to create a new thread before a request is served. The drawback to this approach is that the size of the thread pool is typically fixed to a particular number of threads. Thus, if the Web server is lightly loaded, system resources may be occupied needlessly, degrading the performance of other applications on the endsystem. If the thread pool is not large enough, latency to serving requests grow. Further adaptation can be applied, however, by allowing the thread pool size to grow and shrink as the server load conditions change.

• **Single-threaded concurrent:** Multi-threaded Web servers are popular since they scale well on many multi-processor platforms. However, on some uni-processors, or on operating systems like Windows NT that support asynchronous I/O, the use of threads can yield excessive context switching and synchronization overhead. Hence, on these platforms it is often more desirable to implement *single-threaded concurrent* servers. This type of server is different from an iterative server (which also only uses one thread) since it handles multiple requests concurrently using *Reactive I/O* or *Asynchronous I/O* (described in Section 2.2.2).

The drawback with a single-threaded concurrency strategy is that it can be complicated to implement since it requires I/O mechanisms that possess "inversion of control" [6]. The potential payoff, however, is a faster server on certain uni-processors and operating systems.

Each concurrent strategy outlined above has positive and negative aspects, which are summarized in Table 1.[2] Thus, to optimize performance, Web servers should be *adaptive* to utilize the most beneficial concurrency strategy for particular traffic characteristics, workload, and hardware/OS platforms.

2.2.2 Adaptive I/O Strategies

The results from [8] (summarized at the top of Section 2) show that the relative performance of different I/O strategies can change under varying server load conditions. JAWS is designed to address this issue by allowing the I/O strategy to adapt dynamically to current server conditions. There are three general types of I/O strategies: *synchronous*, *asynchronous* and *reactive* [6]. This section summarizes each of these models and describes their relative merits in the context of adaptive Web servers.

• **Synchronous I/O** describes the model of I/O interaction between a application process and the kernel where the OS does not return the thread of control to the application until the requested I/O operation either completes, completes partially, or fails. This model is well known to UNIX server programmers and is arguably the easiest to use. In Web servers, [8] shows that synchronous I/O is the best choice for small file transfers. However, the disadvantages of this model are:

1. With a single thread of control, it is not possible to engage in concurrent synchronous I/O operations; and
2. Even using multiple threads (or processes), it is possible for an I/O request to block indefinitely. This makes it difficult for the application to shut down gracefully.

[2]The disadvantage of the pool approaches are related to the `accept` call. In Solaris 2.5, `accept` is not a system call and is not atomic. In BSD 4.4, `accept` *is* an atomic system call. More information is available in [19].

Earlier versions of UNIX provided synchronous I/O exclusively. System V provided nonblocking I/O to avoid the blocking problem. However, this solution requires the application to poll to discover if any I/O is available [10].

• **Reactive I/O** alleviates the blocking problems of synchronous I/O without resorting to polling. In this model, an application uses OS event demultiplexing system calls (*e.g.* `select` in UNIX or `WaitForMultipleObjects` in Win32) to determine which handles can perform I/O. Upon return from the call, the application can perform I/O on the returned handles, *i.e.*, the application *reacts* to multiple events occurring on separate handles.

This style of I/O is widely used and has been codified as the *Reactor* design pattern [17]. However, unless Reactive I/O is carefully encapsulated the technique is error-prone due to the complexity of managing multiple I/O handles. Moreover, Reactive I/O does not make effective use of multiple CPUs. Still, reactive I/O is a reasonable alternative for low-end Web server architectures.

• **Asynchronous I/O** simplifies the de-multiplexing of multiple events in one or more threads of control without blocking the application. When an application initiates an I/O operation, the kernel runs the operation asynchronously while the application proceeds with other processing (such as issuing other I/O operations asynchronously). When the application is done initiating I/O requests, it waits for the the system to complete the requests. The advantages of this approach are:

- The application need not block on I/O requests at any time since these requests complete concurrently as the application is doing other processing.
- Asynchronous I/O scales efficiently to multiple CPUs.

These make asynchronous I/O a good candidate for Web systems which involve large file transfers.

Experiments conducted on Windows NT 4.0 [8] show that synchronous I/O can achieve higher throughput and lower latency when the requested file size is small, but that native OS asynchronous I/O mechanisms can achieve better overall performance for large file transfers. These results stem from the overhead required to submit asynchronous I/O requests to the OS. Reactive I/O (coupled with non-blocking handles) is advantageous to systems that need concurrency without using multiple threads or processes.

This proposal aims to further benchmark the impacts of each I/O strategy and design the JAWS framework to adapt to the most optimal mechanism for a particular Web server profile.

Strategy	Advantages	Disadvantages
Single-threaded	No context switching overhead. Highly portable.	Does not scale for multi-processor 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 1: Summary of Concurrency Strategies

2.2.3 Adaptive Caching Strategies

The results reported in [7] showed accessing the filesystem is a significant performance inhibitor. This concurs with the current Web server performance research that has emphasized on caching to achieve better performance [12, 21]. However, [9] reports that fixed caching strategies do not always provide optimal performance. Therefore, we propose to further characterize the impacts of alternative caching strategies to determine which strategies are most effective during various Web server load conditions.

The caching strategies we plan to study are described below:

• **Least Recently Used** is a cache replacement strategy that assumes most requests for cached objects have *temporal* locality. Thus, when the act of inserting a newly object into the cached requires the removal of another object, the object that was *least recently used* is removed. This strategy is relevant to Web systems that serve content with temporal properties (such as daily news reports and stock quotes).

• **Least Frequently Used** a cache replacement strategy that assumes objects that have been requested frequently are more likely to be requested again. Thus, cache objects that have been *least frequently used* are the first to be replaced in the cache. This strategy is relevant to Web systems with relatively static content, such as Lexus Nexus and other databases of historical fact.

• **Hinted** caching is proposed in [12]. This strategy stems from analysis of Web page retrieval patterns that seem to indicate that Web pages have *spatial* locality. That is, a user browsing a Web page is likely to browse the links within the page. Hinted caching is related to *pre-fetching*, though [12] suggests that the HTTP protocol be altered to allow statistical information about the links (or *(*hints) to be sent back to the requester. This modification allows the client to decide which pages to pre-fetch. The same statistical information can be used to allow the server to determine which pages to *pre-cache*.

• **Structured** caching refers to caches that have knowledge of the data being cached. For HTML pages, structured caching refers to storing the cached objects to support hierarchical browsing of a single Web page. Thus, the cache takes advantage of the structure that is present in a Web page to determine the most relevant portions to be transmitted to the client (*e.g.*, a top level view of the page). This can potentially speed up Web access for clients with limited bandwidth and main memory, such as PDAs. Structured caching is related to the use of B-tree structures in databases, which minimize the number of disk accesses required to retrieve data for a query.

We propose to study each caching strategy empirically to determine the best *static* adaptation for a particular Web server configuration. *A priori* knowledge of the content provided by a Web system may determine which strategy is best to apply. However, there are opportunities for additional adaptivity *within* each type of caching strategy, as follows:

- Cache replacement strategies like LRU and LFU must determine when it is suitable to remove files from the cache. Using *dynamic* adaptation to determine this threshold may provide better performance than using a fixed one [9].
- Hinted caching requires the gathering of statistics by the server, which may alter decisions about which pages to pre-cache.
- Structured caching involves detecting resource limitations on the client that may benefit from a page outline rather than the entire document.

We indend to show that by studying the impacts of the strategies under different conditions, and incorporating the knowledge into the JAWS framework, it is possible for a Web server to apply the best possible caching strategy for a given situation.

2.2.4 Summary of Adaptive Strategies for Web Servers

Many of the adaptive strategies described above have been studied extensively in isolation. However, the research literature contains relatively little empirical analysis on the relative performance of complete Web systems implemented using highly configurable adaptive strategies. Moreover, we believe there is a need for comprehensive understanding of *when* to apply various combinations of adaptive strategies.

The JAWS framework will facilitate this understanding by holding the server framework constant and systemtically applying and testing different strategies under varying load conditions on high-speed networks. Furthermore, we will develop profiles that can predict the best combinations of strategies to use in different Web server conditions. By incorporating these profiles into the framework, JAWS will support the automatically configuration of optimal combinations of concurrency, I/O and, caching strategies.

3 Research Objectives

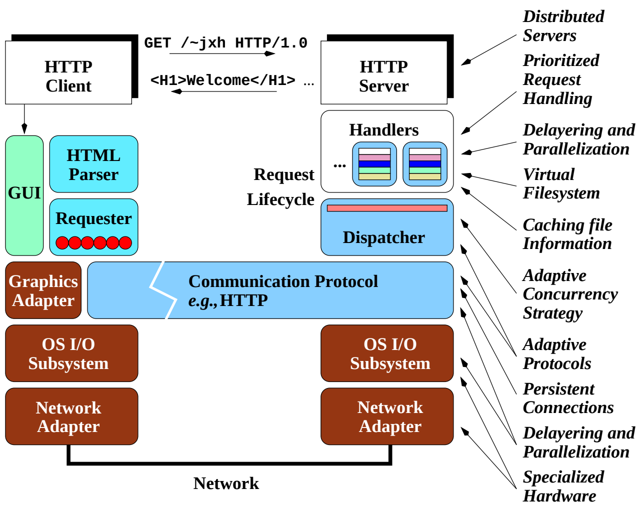

Figure 2: Web System Overview and Server Optimizations

There are many levels at which research on Web server performance can be conducted. Figure 2 illustrate an architectural overview of a Web system and lists potential server-side optimizations. Low-level solutions, such as transport protocol optimizations [11] and specialized hardware [5], are beyond the scope of our work. While these solutions can improve the end-to-end performance of a Web system, they do not directly solve the problem of Web server efficiency. Therefore, our research will leverage off existing work in this field [2, 4, 5] and will focus on the following tasks:

1. Implementing a high-performance Web server. This task is partially completed, but further work is needed. The research aspects to this work involve applying further adaptive optimization strategies and empirically verifying the impact of each strategy. The strategies to be studied include:

- *I/O Strategies* – Asynchronous, Synchronous, and Reactive
- *Caching Strategies* – LRU, LFU, Hinted, and Structured
- *Concurrency Strategies* – Single threaded, Thread-per-request, Thread-per-session (persistent connections), and Thread pool
- *Request Handling Strategies* – Prioritized requests, Parallelized protocol processing, and Content negotiation
- *Adaptive Protocols* – Protocol negotiation (PEP), and Dynamic protocol pipelines

2. Developing a Web server framework. The purpose of the framework is to allow other Web system developers to leverage the results of our work easily. Development of the Web server framework will involve the following research and engineering tasks:

- Identify re-usable architectural components, aided by the use of design patterns as guides and document newly discovered patterns.
- Isolate application-specific portions of the code behind general interfaces to allow new applications to be created more easily.
- Provide a GUI to interactively determine optimal combination of optimization strategies, and strategy parameters.
- Develop a suite of profiles for adaptive optimization strategies.
- For a particular set of hardware characteristics (*e.g.*, network bandwidth, available memory, disk space, processor speed), and OS features (*e.g.*, Memory Mapping, Asynchronous I/O), provide optimization strategies and parameter settings, based on empirical measurements.
- Provide a facility for monitoring server conditions so that run-time changes may be made to the server's behavior in order to optimize performance.

The intended delivery dates for these tasks have been summarized in Figure 3.

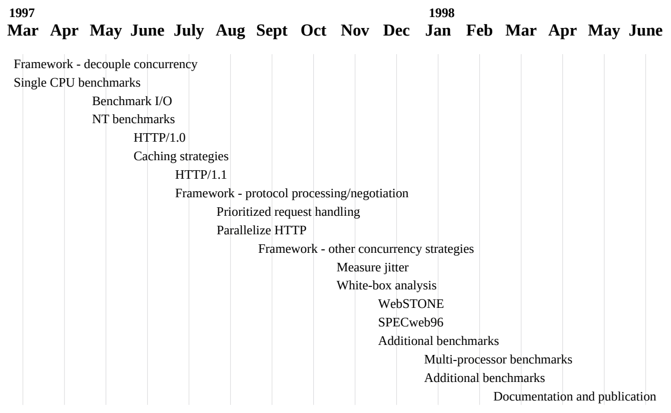

Figure 3: Objectives timeline for progress toward dissertation

4 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, our research objective is to produce the fastest server possible *for a given hardware/OS platform configuration*.

Our research to date has focused on prototyping JAWS and empirically measuring its performance relative to other Web servers over high-speed ATM networks. Servers that performed poorly were studied with whitebox techniques (such as `Quantify`) to discover sources of bottlenecks. Servers that performed well were also analyzed thoroughly to determine what they did right. We found that file system operations incur significant overhead, which JAWS alleviates by applying perfect hashing and other server-side caching techniques. These results corroborate results from other research on Web servers [12, 21, 9].

When network and file I/O are held constant, however, the largest portion of the HTTP request life-cycle 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 and event dispatching strategy. Therefore, we subjected JAWS to varying load conditions and compared the relative performance between combinations of thread pool and thread-per-request concurrency strategies with asynchronous and synchronous I/O strategies on Windows NT 4.0 [8] and Solaris 2.5 [7].

Our preliminary results demonstrate that thread pool combined synchronous I/O performed the best for smaller files. However, on Windows NT 4.0, thread pools combined with the asynchronous `TransmitFile` call performed substantially better for large files. However, `TransmitFile` appeared more sensitive to changes in server hit rate with respect to latency than the other methods.

In addition, our results demonstrate how dynamically adaptive Web servers can provide substantial performance improvements. Thus, we propose an in-depth study of the impacts from applying different combinations of optimization strategies for concurrency, I/O dispatching, and caching techniques. In addition, we propose to test and analyze the conditions for when particular combinations are most effective. Once we create these performance profiles, we propose to refine the JAWS framework to develop high-performance Web systems that can adapt dynamically to changing server load conditions to provide optimal performance.

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

[4] Ramon Caceres. Efficiency of ATM Networks in Transporting Wide-Area Data Traffic, December 1991. Submitted to Computer Networks and ISDN Systems.

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

[6] Tim Harrison, Irfan Pyrarli, Douglas C. Schmidt, and Thomas Jordan. Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers. In *The* 4^{th} *Pattern Languages of Programming Conference*, September 1997.

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

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

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

[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. In *Proceedings of ACM SIGCOMM '95 Conference in Computer Communication Review*, pages 299–314, Boston, MA, USA, August 1995. ACM Press.

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

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

[14] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. In *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996. USENIX.

[15] 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.

[16] 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.

[17] 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.

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

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