Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks

James C. Hu,* Irfan Pyarali,† Douglas C. Schmidt

{jxh,irfan,schmidt}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, Missouri

An abridged version of this paper appeared in the Proceedings of the 2^{nd} Global Internet mini-conference held in conjunction with GLOBECOM '97, Phoenix, AZ, November, 1997.

Abstract

This paper provides two contributions to the study of high-performance Web servers. First, it outlines the optimizations necessary to build efficient and scalable Web servers and illustrates how we've applied some of these optimizations to create JAWS. JAWS is a high-performance Web server that is explicitly designed to alleviate overheads incurred by existing Web servers on high-speed networks. It consistently outperforms existing Web servers (such as Apache, Java Server, PHTTPD, Zeus, and Netscape Enterprise) over 155 Mbps ATM networks on UNIX platforms.

Second, this paper describes how we have customized JAWS to leverage advanced features of Windows NT for multi-processor hardware over ATM. The Windows NT features used in JAWS include asynchronous mechanisms for connection establishment and data transfer. Our previous benchmarking studies demonstrate that once the overhead of disk I/O is reduced to a negligible constant factor (e.g., via memory caches), the primary determinants of Web server performance are the concurrency and event dispatching strategies.

Our performance results over a 155 Mbps ATM link indicate that certain Windows NT asynchronous I/O mechanisms (i.e., `TransmitFile`*) provide superior performance for large file transfers compared with conventional synchronous multi-threaded servers. On the other hand, synchronous event dispatching performed better for files less than 50 Kbytes. Thus, to provide optimal performance, Web servers should be* adaptive*, choosing to use different mechanisms (e.g.,* `TransmitFile`*) to handle requests for large files, while using alternative I/O mechanisms (e.g., synchronous event dispatching) for requests for small files.*

*This work was supported in part by grants from Object Technologies International and Eastman Kodak Company.

†Funding provided by Siemens Medical Engineering.

1 Introduction

The emergence of the World Wide Web (Web) as a mainstream development platform has yielded many hard problems for application developers, who must provide high quality of service to application users. Strategies for improving client performance include client-side caching and caching proxy servers [19]. However, performance bottlenecks persist on the server-side. These bottlenecks arise from factors such as inappropriate choice of concurrency and dispatching strategies, excessive filesystem access, and unnecessary data copying.

As high-speed networks (such as ATM) and high-performance I/O subsystems (such as RAID) become ubiquitous, the bottlenecks of existing Web servers are becoming increasingly problematic. To alleviate these bottlenecks, Web servers must utilize an integrated approach that combines optimizations at multiple levels. Figure 1 illustrates the general architecture of a Web system.

This diagram provides a layered view of the architectural components required for an *HTTP client* to retrieve an HTML file from an *HTTP server*. Through *GUI* interactions, the client application user instructs the HTTP client to retrieve a file. The *requester* is the active component of the client that communicates over the *network*. It issues a request for the file to the server with the appropriate syntax of the *transfer protocol*, in this case HTTP. Incoming requests to the *HTTP server* are received by the *dispatcher*, which is the request demultiplexing engine of the server. It is responsible for creating new threads or processes (for concurrent Web servers) or managing descriptor sets (for single-threaded concurrent servers). Each request is processed by a *handler*, which goes through a *lifecycle* of parsing the request, logging the request, fetching file

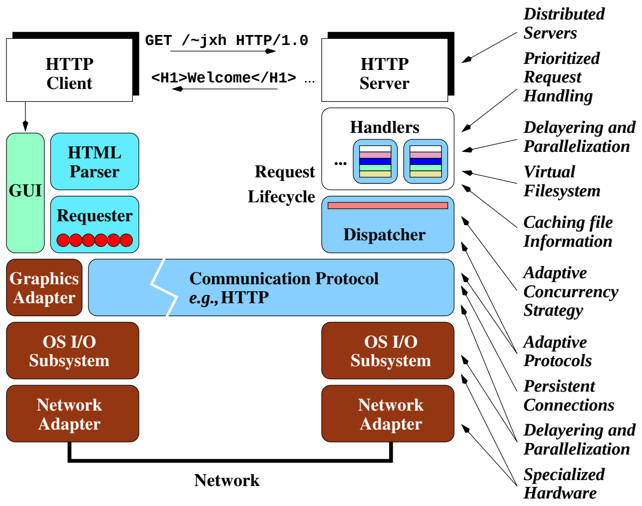

Figure 1: Overview of a Typical Web System and Optimizations

status information, updating the cache, sending the file, and cleaning up after the request is done. When the response returns to the client with the requested file, it is parsed by an *HTML parser* so that the file may be rendered. At this stage, the *requester* may issue other requests on behalf of the client, *e.g.*, in order to fill a client-side cache.

Our experience building Web servers on multiple OS platforms indicates that the effort required to improve performance can be simplified significantly by explicitly leveraging operating system features. For example, an optimized file I/O system that automatically caches open files in main memory helps to reduce latency. Likewise, support for asynchronous event dispatching and the Proactor pattern [5] can increase server throughput by reducing context switching and synchronization overhead incurred from multi-threading.

This paper presents two complementary strategies for developing optimized Web servers. First, we present empirical results demonstrating that to achieve optimal performance, Web servers must support *dynamic* adaptivity (*i.e.*, the ability to update behavior "online" to account for changes in run-time conditions). Second, we describe our recent efforts at adapting a high-performance Web server (originally developed on UNIX) to leverage the asynchronous event dispatching mechanisms on Windows NT. This work illustrates the importance of *static* adaptivity (*i.e.*, changing the behavior of the Web server "offline" to account for OS platform characteristics).

Our research vehicle for demonstrating the effectiveness of dynamic and static adaptation is *JAWS*. JAWS is both an adaptive Web server and a development framework for Web servers that run on multiple OS platforms including Win32 (*i.e.*, Windows NT and Windows '95), most versions of UNIX (*e.g.*, SunOS 4.x and 5.x, SGI IRIX, HP-UX, OSF/1, AIX, Linux, and SCO), and MVS OpenEdition.

We motivate the need for dynamic and static adaptivity in JAWS as follows:

- **The need for dynamic adaptivity:** On many OS platforms, under different workloads, a single, statically configured content transfer mechanism cannot provide optimal performance. As shown in Section 4, the performance of different OS level I/O mechanisms varies considerably according to file size. For instance, on Windows NT 4.0, synchronous I/O provides the best performance for transferring small files, whereas the `TransmitFile` operation provides the best performance for transferring large files under heavy loads.

- **The need for static adaptivity:** To achieve high performance, Web servers must be adapted statically to use native high-performance mechanisms provided by the OS platform. For example, different OS platforms may provide specialized I/O mechanisms (such as asynchronous I/O or bulk data transfer) or specialized devices (such as high-speed ATM network interfaces [2]). Therefore, simply porting a Web server to use common OS mechanisms and APIs (such as BSD sockets, `select`, and POSIX threads) is not sufficient to achieve maximal performance on different OS platforms.

The results in this paper are based on extensions to JAWS' original synchronous event dispatching model (based on the POSIX threading model and BSD sockets). These extensions support the asynchronous event dispatching and communication mechanisms available on Windows NT (JAWS-NT). The Windows NT mechanisms incorporated into JAWS-NT include overlapped I/O, I/O completion ports, `TransmitFile`, `GetQueueCompletionStatus`, and `AcceptEx`.[1]

As shown in Section 4, performance measurements of JAWS-NT over a ~155 Mbps ATM link indicate significant throughput and latency variance between the synchronous and asynchronous event dispatching and concurrency models on Windows NT. In addition, our experience with the Windows NT asynchronous event dispatching mechanisms has revealed other benefits besides improved throughput and latency. For instance, asynchronous event dispatching allows Web servers to significantly reduce the number of threading resources required to handle client requests concurrently.

The remainder of this paper is organized as follows: Section 2 provides an overview of the JAWS server framework design and explains the optimizations we have applied thus far; Section 3 outlines the concurrency strategies supported by JAWS-NT and describes the key differences between the synchronous and asynchronous event dispatching models; Section 4 analyzes our performance measurements of JAWS-NT

[1] These Windows NT mechanisms are all described in Section 3.3.2.

over an ATM network; Section 5 compares a highly optimized JAWS implementation against Netscape Enterprise and Microsoft Internet Information Server (IIS); and Section 7 presents concluding remarks.

2 Design of the JAWS Adaptive Web Server

The performance results presented in this paper were conducted using a version of JAWS that was customized for Windows NT (JAWS-NT). JAWS is both a Web server and framework for building flexible and adaptive high-performance Web systems. The optimizations and design principles in JAWS are guided by results from our empirical analysis [6] of Web server performance bottlenecks over high-speed ATM networks. Assuming sufficiently high network bandwidth and large file system caching, our experiments have identified the following determinants of Web server performance:

• **Concurrency strategy and event dispatching:** Request dispatching occupies a large portion (*i.e.*, ~50%) of non-I/O related Web server overhead. Therefore, the choice of concurrency strategy (*e.g.*, Thread/Process Pool vs. Thread/Process-per-request, etc.) and dispatching strategy (*e.g.*, asynchronous vs. synchronous) has a major impact on performance.

• **Avoiding the filesystem:** Web servers that implement sophisticated file data and file stats caching strategies (such as PHTTPD and JAWS) perform much better than those that do not (such as Apache).

The UNIX version of JAWS (described in [6]) consistently outperforms other servers in our test suite of Web servers over 155 Mbps ATM networks. This section briefly outlines the design principles and optimizations used by JAWS to achieve such high performance.

Figure 2 illustrates the object-oriented software architecture of the JAWS Web server, which contains the following three components:

• **Protocol Handlers:** JAWS' `Protocol Handlers` are components that can be specialized to process various protocol requests. For instance, `HTTP Handlers` parse HTTP 1.0 requests from WWW browsers and perform the work specified by requests (*e.g.*, retrieving web pages). The JAWS framework supports seamless integration of `Handlers` that can be specialized for other protocols (*e.g.*, HTTP/1.1 and DICOM).

• **Event Dispatcher:** This component encapsulates the concurrency and event dispatching strategies utilized by JAWS. Our experimental results indicate that the concurrency and event dispatching strategies significantly affects Web server performance. Therefore, JAWS can customize these strategies to account for environmental factors such as user-level

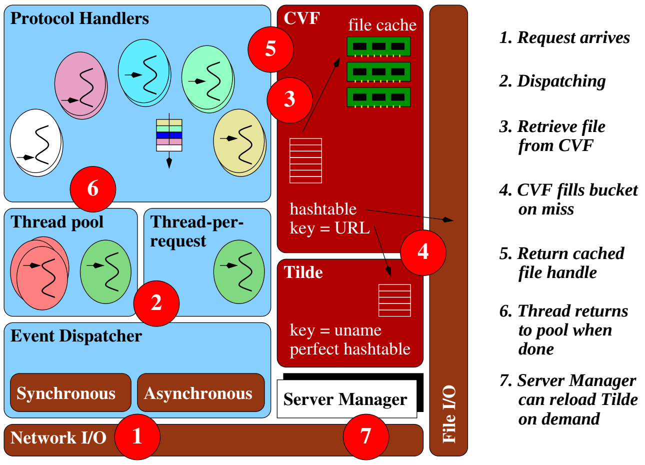

Figure 2: JAWS Design Overview

vs. kernel-level threading in the OS and the number of CPUs in the endsystem.

As discussed in Section 3.2, JAWS currently supports concurrent strategies (such as Thread-per-Request and Thread Pool) that perform well on many OS platforms. Likewise, as discussed in Section 3.1, JAWS' `Event Dispatcher` determines whether its threads use a *synchronous* or an *asynchronous* dispatching model.

• **Server Manager:** This component is responsible for configuring `Protocol Handlers` according to the protocols supported by JAWS. In addition, the `Server Manager` configures the event dispatching models used by `Protocol Handlers`. To support adaptation, the `Server Manager` uses the Service Configurator [7] pattern to dynamically relink (*i.e.*, at run-time) portions JAWS when it is updated (*e.g.*, to support new versions of the HTTP protocol or to configure a more efficient algorithm or implementation).

In addition to these three components, JAWS contains the following two optimized components that alleviate unnecessary filesystem overhead:

• **Tilde:** which is a dynamically loaded, perfect hash table [16] of home directories, which is used to expand the ~ that is often prepended to the username in a URL. `Tilde` is created off-line and can be rebuilt whenever a new user is added to the system and dynamically reloaded into JAWS via the `Server Manager`.

• **Cached Virtual Filesystem (CVF):** `CVF` caches information (such as access permissions and content type) about the files so that future requests for the same file need not access the file system. `CVF` is implemented using a novel synchronization strategy that minimizes locking overhead.

`Tilde` and `CVF` are decoupled from the three core JAWS components described above to allow transparent configurations that depend on platform support for high-performance cached filesystems. Additional information on JAWS I/O optimizations is available in [6].

3 Event Dispatching and Concurrency Strategies for Web Servers

The JAWS `Event Dispatcher` is a flexible component that can be configured to use multiple concurrency strategies (such as Thread Pool and Thread-per-Request). The initial design of JAWS used a *synchronous* event dispatching model since it was developed on Solaris 2.5, which does not provide efficient asynchronous I/O support. This section describes how the JAWS framework was enhanced to support the *asynchronous* event dispatching model provided by Windows NT 4.0.

3.1 Event Dispatching Strategies

3.1.1 Synchronous Event Dispatching

A common Web server architecture is the synchronous event dispatching. The architecture consists of two layers: the *I/O Subsystem*, and the *Protocol Handlers*, as shown in Figure 3. The I/O Subsystem typically resides in the kernel and is implemented with sockets. One socket plays the role of the acceptor, which is a factory that creates new data sockets. Protocol Handlers, having its own thread of control, reads and processes the data coming from the socket that was created from a newly accepted connection. Synchronous event dispatching dedicates the selected thread to the new client for the duration of the file transfer.

3.1.2 Asynchronous Event Dispatching

The asynchronous event dispatching architecture also consists of both I/O subsystem and protocol handler layers, as shown in Figure 4. However, in asynchronous I/O, each I/O operation is "handed off" to the kernel, where it runs to completion. Thus, the initiating thread does not block. When the kernel has completed the operation, the kernel notifies the process through an *I/O completion port*. An I/O completion port is a kernel-level thread-safe queue of I/O completion notifications. The pros and cons of using asynchronous event dispatching are described in [5].

There are several benefits of using I/O completion ports.

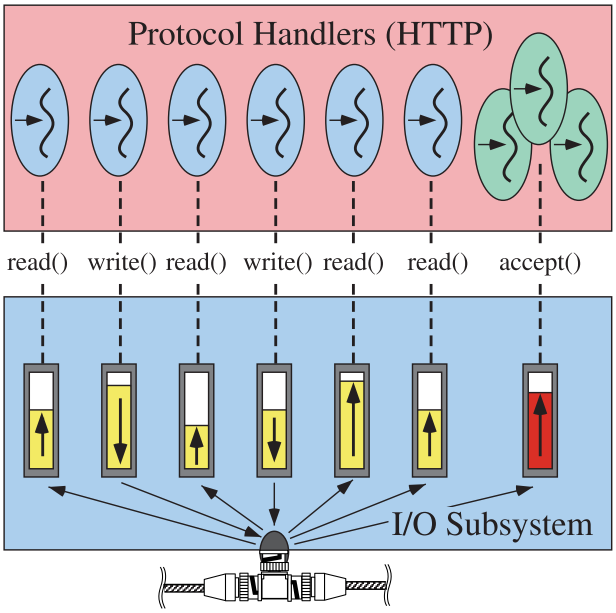

Figure 3: Synchronous Event Dispatching

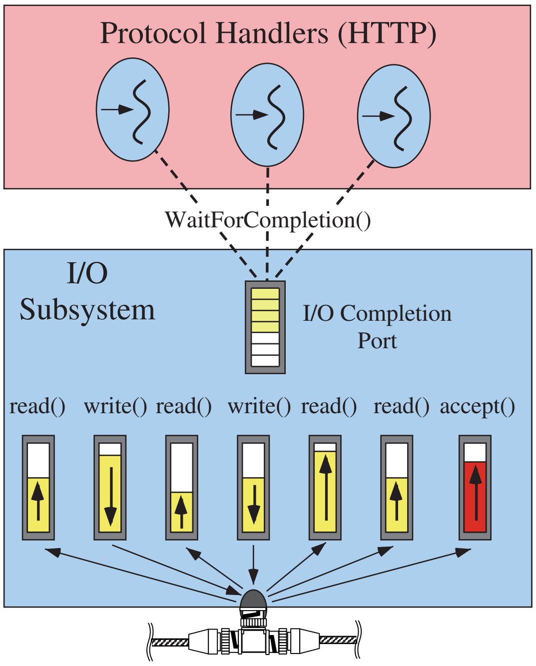

Figure 4: Asynchronous Event Dispatching

- *Increased flexibility and scalability* – The thread initiating the asynchronous I/O operation and the thread dequeueing the completion status from the port can be different. This makes it possible to tune the level of concurrency in an application by simply increasing the number of completion handler threads.

- *Fewer threads and less overhead* – Since threads no longer block on I/O operations, the asynchronous Thread Pool requires significantly fewer threads than the synchronous Thread Pool. The reduction of threads in the system reduces context switching and synchronization overhead.

3.2 Concurrency Strategies

Existing Web servers use a wide range of concurrency strategies. These strategies include single-threaded concurrency (*e.g.*, Roxen), process concurrency (*e.g.*, Apache), and thread concurrency (*e.g.*, PHTTPD and Zeus). Single-threaded servers cannot take advantage of multi-CPU hardware concurrency. Likewise, servers that use multiple processes incur higher process creation overhead. Since process creation incurs significantly greater overhead than thread creation, our discussion focuses on threading strategies.

Our measurements revealed that the time required to create a thread on a 180 MHz, dual-CPU Pentium PRO2 running Windows NT 4.0 is ∼0.4ms. However, the time required to create a process is ∼4.5ms, which is an order of magnitude higher. As a result, JAWS implements concurrency via the threading strategies described below.

3.2.1 Thread-per-Request

In the Thread-per-Request model, a new thread is spawned to handle each incoming request. As shown in Figure 5, one thread (the acceptor) blocks on the acceptor socket. The acceptor thread serves as a factory that creates a new handler thread to interact with each client.

After creating a new handler thread, the acceptor thread continues to wait for new connections on the acceptor socket. In contrast, the handler thread reads the HTTP request, services it, and transmits the result to the client. The lifecycle of a handler thread completes after the data transfer operation is finished.

The Thread-per-Request model is useful for long-duration requests from multiple clients. It is less useful for short-duration requests due to the overhead of creating a new thread for each request. In addition, it can consume a large number of OS resources if many clients simultaneously perform requests during periods of peak load.

3.2.2 Thread Pool

In the Thread Pool model, a group of threads are pre-spawned during Web server initialization, as shown in Figure 6. Pre-spawning eliminates the overhead of creating a new thread for

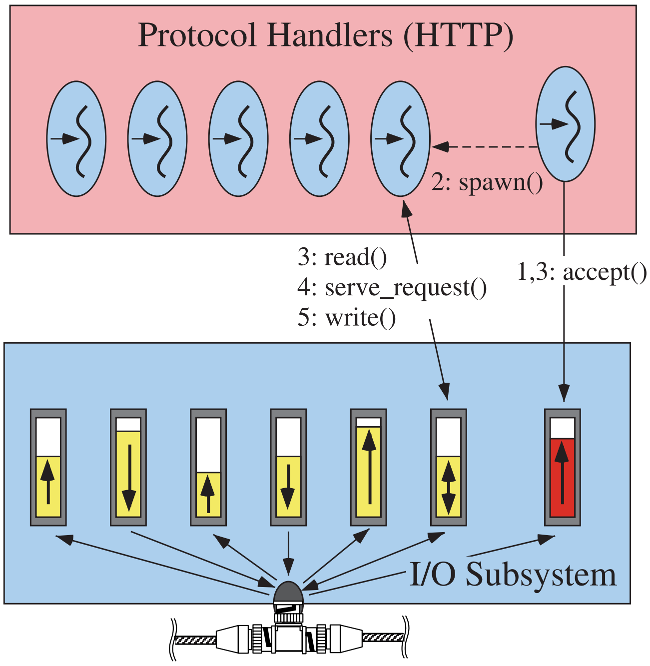

Figure 5: Thread-per-Request

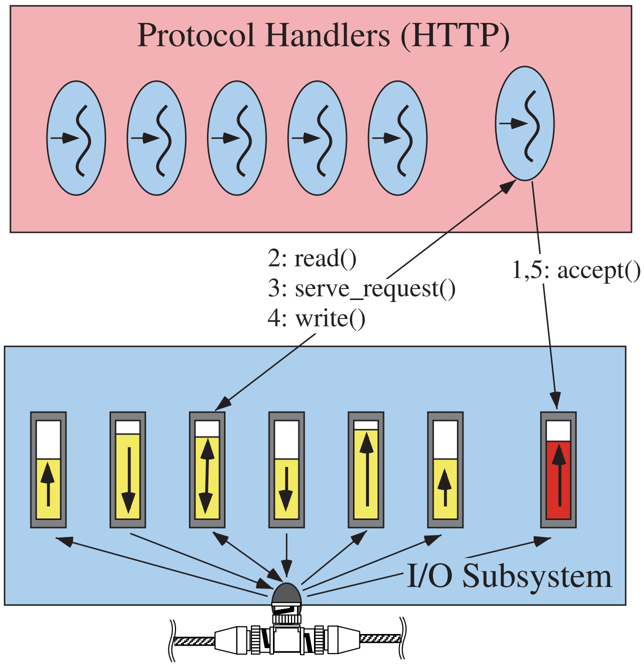

Figure 6: Thread Pool

each request. Each thread blocks in `accept` waiting for connection requests to arrive from clients. When a new connection arrives, the OS selects a thread from the pool to `accept` it.

The thread then performs a synchronous `read` from the new connection. Once the entire HTTP request has been read, the thread performs the necessary computation and filesystem operations to service the request. The requested data is then transmitted synchronously to the client. The thread returns to the Thread Pool and reinvokes `accept` after the data transmission completes.

Synchronous Thread Pool is useful for bounding the number of OS resources consumed by a Web server. Client requests can execute concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests must be queued (typically in the kernel's TCP layer) until a thread becomes available.

To reduce latency, the Thread Pool can be configured to always have threads available to service new requests. The number of threads needed to support this policy can be very high during peak loads as threads block in long-duration synchronous I/O operations.

The asynchronous Thread Pool approach (described in Section 3.2.3) improves this model by considerably reducing the number of threads in the system.

Incidentally, in Solaris 2.5 it is not possible to have multiple threads in a process all blocking simultaneously in `accept` on the same acceptor socket. Therefore, it is necessary to protect `accept` with a mutex lock in a Thread Pool Web server. In contrast, Windows NT 4.0 supports simultaneous `accept` calls, so additional synchronization is not required in JAWS-NT.

3.2.3 Asynchronous Thread Pool

Figure 4 shows the asynchronous Thread Pool model. Like the synchronous Thread Pool model, the Thread Pool is created during Web server initialization. Unlike the synchronous model, however, the threads wait on a *completion port* rather than waiting on `accept`. The OS queues up results from all asynchronous operations (*e.g.*, asynchronous accepts, reads, and writes) on the completion port.

The result of each asynchronous operation is handled by a thread selected by the OS from the pool of threads waiting on the completion port. The asynchronous model is useful since the same programming model works for a single thread, as well as multiple threads. However, the thread that initiated the asynchronous operation need not be the one selected to handle its completion. Therefore, it is hard to implement concurrency strategies other than Thread Pool with an asynchronous event dispatching model.

3.2.4 Thread-per-Connection

In the Thread-per-Connection model the newly created handler thread is responsible for the lifetime of the client connection, rather than just a single request from the client. Therefore, the new thread may serve multiple requests before terminating. Since HTTP 1.0 establishes a new connection for each request, Thread-per-Connection is not suitable. This model is applicable, however, in HTTP 1.1, which supports persistent connections [11, 14].

Thread-per-Connection provides good support for prioritization of client requests. For instance, higher priority clients can be associated with higher priority threads. Thus, request from high priority clients will be served ahead of other requests since the OS can preempt lower priority threads.

One drawback to Thread-per-Connection is that if certain connections receive considerably more requests than others, they can become a performance bottleneck. In contrast, Thread Pool and Thread-per-Request provide better support for load balancing.

3.3 Optimizing JAWS for Windows NT

3.3.1 Overview of Windows NT Asynchronous I/O

The Windows NT asynchronous I/O model supports *proactive* semantics, which allow applications to actively initiate I/O-related operations (*e.g.*, `ReadFile`, `WriteFile`, and `TransmitFile`). The following steps are required to program asynchronous I/O on Windows NT:

1. I/O and event `HANDLE` creation: First, an application creates a `HANDLE` that corresponds to an I/O channel for the type of networking mechanism being used (such as a socket or named pipe).

Next, an application creates a `HANDLE` to a Win32 event object and uses this event object's `HANDLE` to initialize an overlapped I/O structure. The event object will be signaled when asynchronous operations on the `HANDLE` complete.

2. Asynchronous operation invocation: The `HANDLE` to the I/O channel and the overlapped I/O structure are then passed to the asynchronous I/O operation (such as `WriteFile`, `ReadFile`, or `AcceptEx`). The initiated operation proceeds asynchronously and does not block the caller.

3. Asynchronous operation completion: When an asynchronous operation completes, the event object specified inside the overlapped I/O structure is set to the "signaled" state. Subsequently, Win32 demultiplexing functions (such as `WaitForSingleObject` or `WaitForMultipleObjects` can be used to detect the signaled state of the Win32 event object. These functions indicate when an outstanding asynchronous operation has completed.

3.3.2 Overview of Windows NT Functions Relevant to Web Servers

The following Win32 functions are particularly relevant for developers of asynchronous Web Servers on Windows NT:

• **GetQueueCompletionStatus:** The function `GetQueue-CompletionStatus` attempts to dequeue an I/O completion result from a specified completion port. If there are no completion results queued on the port, the function blocks the calling thread waiting for asynchronous operations associated with the completion port to finish. The blocking thread returns from the `GetQueuedCompletionStatus` function when (1) it can dequeue a completion packet or (2) when the function times out.

Windows NT selects the thread that has executed most recently from among the waiter threads on the completion port to handle the new connection. This reduces context switching overhead since it increases the likelyhood that thread context information is still cached in the CPU and OS.

• **AcceptEx:** The `AcceptEx` function combines several socket functions into a single API/kernel transition. When it complete successfully, `AcceptEx` performs the following three tasks: (1) a new connection is accepted, (2) both the local and remote addresses for the connection are returned, and (3) the first block of data sent by the remote client is received. Microsoft [10] claims that programs establishing connections with `AcceptEx` will perform better than those using the `accept` function. However, our results in Section 4 show that over high-speed networks there is not much difference in performance.

• **TransmitFile:** `TransmitFile` is a custom Win32 function that sends file data over a network connection, either synchronously or asynchronously. The function uses the Windows NT virtual memory cache manager to retrieve the file data. As shown in Section 4, the asynchronous form of `TransmitFile` is the most efficient mechanism for transferring large amounts of data over sockets on Windows NT.

In addition to transmitting files, `TransmitFile` allows data to be prepended and appended before and after the file data, respectively. This is particularly well-suited for Web servers since they typically send `HTTP` header and trailer data with the requested file. Hence, all the data to the client can be sent in a single system call, which minimizes mode switching overhead.

The Windows NT *Server* optimizes `TransmitFile` for high performance; all our benchmarks were run on NT Server. The Windows NT *Workstation* optimizes the function for minimum memory and resource utilization. Our measurements confirm that that `TransmitFile` on Windows NT *Server* substantially outperforms `TransmitFile` on Windows NT *Workstation*.

3.3.3 Customizing JAWS for Windows NT Asynchronous I/O

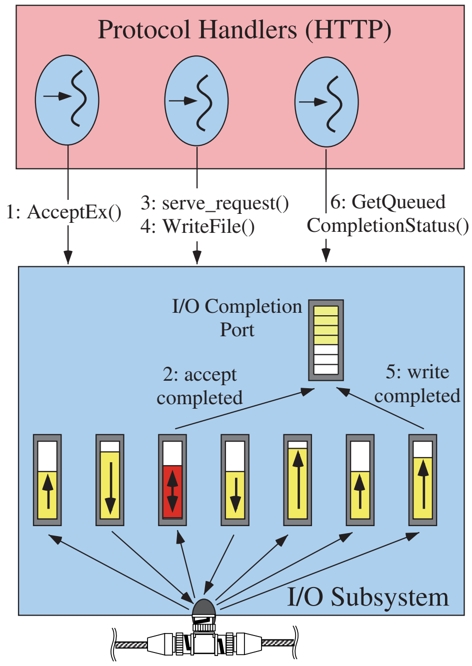

Figure 7: Asynchronous Event Dispatching (Windows NT)

We modified the original Solaris implementation of JAWS to use the Windows NT asynchronous event dispatching model described above. Figure 7 shows the interactions between components in the resulting JAWS-NT model.

When a JAWS Web server process begins execution, the main thread initiates multiple `AcceptEx` calls asynchronously. All threads in the process block on the completion port by calling `GetQueuedCompletionStatus`. When a new connection arrives, the kernel places the result from `AcceptEx` onto the completion port. If the initial data block received by `AcceptEx` does not contain the entire HTTP request, the selected handler thread must initiate an asynchronous read (using `ReadFile`).

After the entire HTTP request is received from the client, the handler thread services the request by locating the file to transmit to the client. If the server is using `WriteFile` to transmit the file data to the client, it memory maps the requested file first.[2] After initiating the asynchronous transfer (*e.g.*, using `WriteFile` or `TransmitFile`) the handler thread blocks on the completion port. When the asynchronous transfer completes, the result is queued up at the completion

[2] If the server is using `TransmitFile`, there is no need to memory map the requested file since `TransmitFile` uses the operating system's virtual memory cache manager to retrieve the file data.

port. The OS then selects a thread from those waiting on the completion port. This thread dequeues the result and performs the necessary cleanup (*e.g.*, closes the data socket) to finalize the HTTP transaction.

It is possible that the thread initiating `AcceptEx`, the thread initiating `WriteFile` or `TransmitFile`, and the thread dequeueing the completion status from the port might be different. This increases the adaptivity of JAWS since it is possible to tune the level of concurrency simply by increasing the number of completion handler threads. Moreover, due to the design of the JAWS `Event Dispatcher` framework, these changes do not require any modifications to its concurrency architecture.

4 Benchmarking Testbed and Results

4.1 Hardware Testbed

Our hardware testbed is shown in Figure 8.

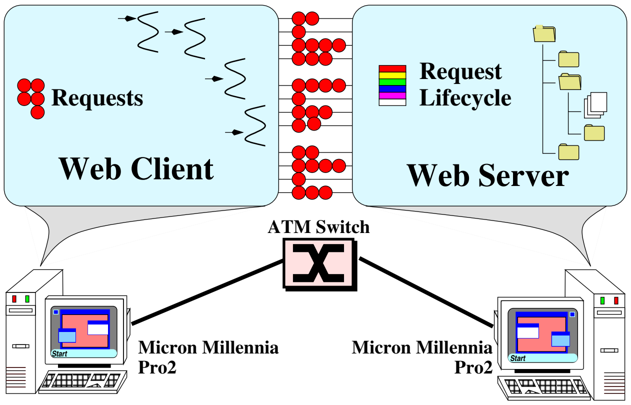

Figure 8: Benchmarking Testbed Overview

The testbed consists of two Micron Millennia PRO2 plus workstations. Each PRO2 has 128 MB of RAM and is equipped with 2 PentiumPro processors. The client machine has a clock speed of 200 MHz, while the server machine runs 180 MHz. In addition, each PRO2 has an ENI-155P-MF-S ATM card made by Efficient Networks, Inc. and is driven by Orca 3.01 driver software. The two workstations were connected via an ATM network running through a FORE Systems ASX-200BX, with a maximum bandwidth of 622 Mbps. However, due to limitations of LAN emulation mode, the peak bandwidth of our testbed is approximately 120 Mbps.

4.2 Software Request Generator

We used the WebSTONE [4] v2.0 benchmarking software to collect client- and server-side metrics. These metrics included *average server throughput*, and *average client latency*. WebSTONE is a standard benchmarking utility, capable of generating load requests that simulate typical Web server file access patterns. Our experiments used WebSTONE to generate loads and gather statistics for particular file sizes in order to determine the impacts of different concurrency and event dispatching strategies.

The file access pattern used in the tests is shown in Table 1.

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

Table 1: File Access Patterns

This table represents actual load conditions on popular servers, based on a study of file access patterns conducted by SPEC [1].

4.3 Experimental Results

The results presented below compare the performance of several different adaptations of the JAWS Web server. We discuss the effect of different event dispatching and I/O models on throughput and latency. 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 delay in milliseconds seen by the client from the time it sends the request to the time it completely receives the file. 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.

The five graphs shown for each of throughput and latency represent different file sizes used in each experiment, 500 bytes through 5 Mbytes by factors of 10. These files sizes represent the spectrum of files sizes benchmarked in our experiments, in order to discover what impact file size has on performance.

4.3.1 Throughput Comparisons

Figures 9-13 demonstrate the variance of throughput as the size of the requested file and the server hit rate are systematically increased. As expected, the throughput for each connection generally degrades as the connections per second increases. This stems from the growing number of simultaneous connections being maintained, which decreases the throughput per connection.

As shown in Figure 11, the throughput of Thread-per-Request can degrade rapidly for smaller files as the connection load increases. In contrast, the throughput of the synchronous Thread Pool implementation degrade more gracefully. The reason for this difference is that Thread-per-Request incurs higher thread creation overhead since a new thread is spawned for each `GET` request. In contrast, thread creation overhead in the Thread Pool strategy is amortized by pre-spawning threads when the server begins execution.

The results in figures 9-13 illustrate that `TransmitFile` performs extremely poorly for small files (*i.e.*, < 50 Kbytes). Our experiments indicate that the performance of `TransmitFile` depends directly upon the number of simultaneous requests. We believe that during heavy server loads (*i.e.*, high hit rates), `TransmitFile` is forced to wait while the kernel services incoming requests. This creates a high number of simultaneous connections, degrading server performance.

As the size of the file grows, however, `TransmitFile` rapidly outperforms the synchronous dispatching models. For instance, at heavy loads with the 5 Mbyte file (shown in Figure 13), it outperforms the next closest model by nearly 40%. `TransmitFile` is optimized to take advantage of Windows NT kernel features, thereby reducing the number of data copies and context switches.

4.3.2 Latency Comparisons

Figures 9-13 demonstrate the variance of latency performance as the size of the requested file and the server hit rate increase. As expected, as the connections per second increases, the latency generally increases, as well. This reflects the additional load placed on the server, which reduces its ability to service new client requests.

As before, `TransmitFile` performs extremely poorly for small files. However, as the file size grows, its latency rapidly improves relative to synchronous dispatching during light loads.

4.3.3 Summary of Benchmark Results

As illustrated in the results presented above, there is significant variance in throughput and latency depending on the concurrency and event dispatching mechanisms. For small files, the synchronous Thread Pool strategy provides better overall performance. Under moderate loads, the synchronous event dispatching model provides slightly better latency than the asynchronous model. Under heavy loads and with large file transfers, however, the asynchronous model using `TransmitFile` provides better quality of service. Thus, under Windows NT, an optimal Web server should adapt itself to either event dispatching and file I/O model, depending on the server's workload and distribution of file requests.

5 Comparing JAWS With Other Windows NT Web Servers

This section compares a highly optimized JAWS implementation against Netscape Enterprise and Microsoft Internet Information Server (IIS). These results are shown in Figures 14-18. They demonstrate that JAWS is competitive with state-of-the-art commercial Web server implementations.

The figures reveal that JAWS does not perform as well as Enterprise or IIS for small files. However, as the file size grows, JAWS overtakes the other servers in performance. Two conclusions can be drawn from these results. First, there are still performance issues beyond the scope of this paper that require research to determine how to improve JAWS performance for transferring small files. Second, it affirms our hypothesis that a Web server can only achieve optimal performance by employing adaptive techniques.

6 Related Work

As shown in Figure 1, the JAWS framework focuses on optimizing a range of layers in a Web system. Therefore, it is influenced by a variety of related efforts. This section describes existing work that is most relevant to JAWS.

6.1 Performance Evaluation

The need for high-capacity servers has spurred commercial sector activity and many server implementations are available on the market [17]. The growing number of Web servers has prompted the need for assessing their relative performance. The current standard benchmarks available are WebStone [4] (by SGI) and SPECweb96 [1] (by SPEC), both heavily influenced by the design of LADDIS [21]. WebStone and SPECweb96 attempt to measure overall performance. They rate the performance of a server with a single number (a higher number indicates better performance). These benchmarks are based on a process-based concurrency model and utilize multiple machines to simulate heavy loads. We have found that

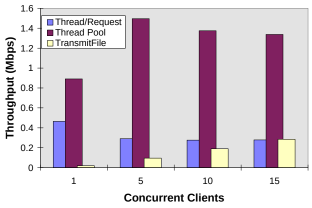

Figure 9: Experiment Results from 500 Byte File

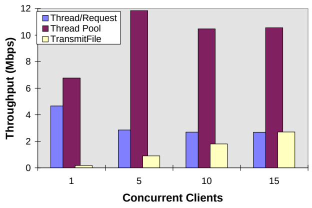

Figure 10: Experiment Results from 5K File

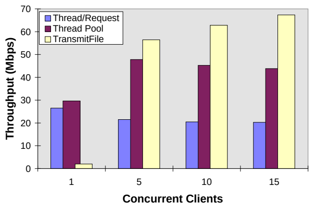

Figure 11: Experiment Results from 50K File

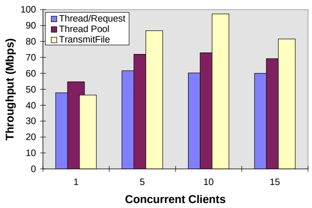

Figure 12: Experiment Results from 500K File

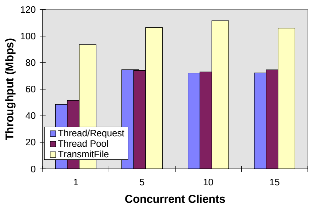

Figure 13: Experiment Results from 5M File

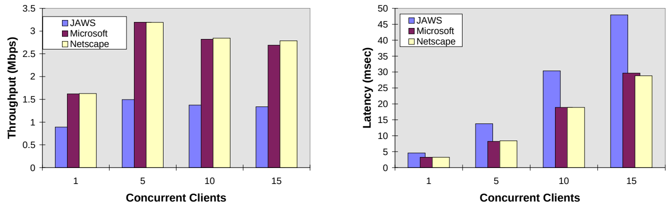

Figure 14: Performance Results from 500 Byte File

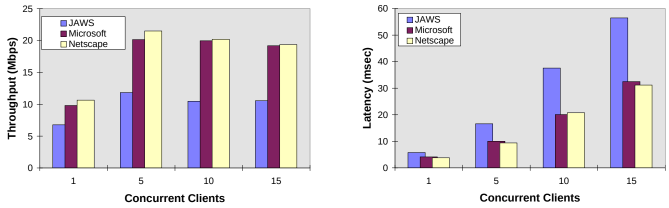

Figure 15: Performance Results from 5K File

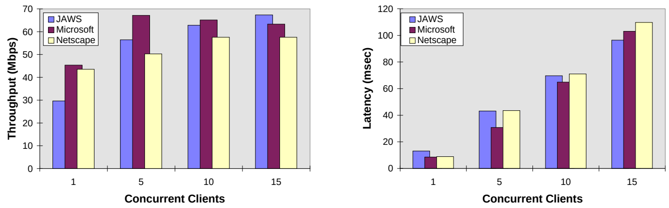

Figure 16: Performance Results from 50K File

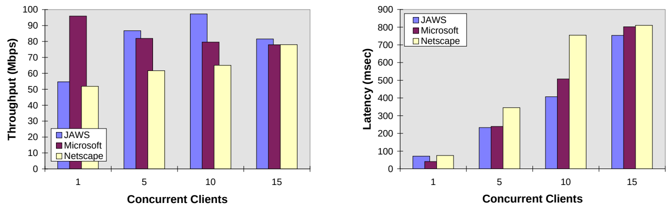

Figure 17: Performance Results from 500K File

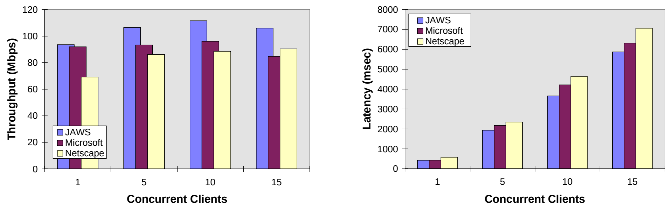

Figure 18: Performance Results from 5M File

a thread-based concurrency model allows a single client machine to generate higher loads, thus requiring fewer machines in the benchmarking testbed.

6.2 Optimization Strategies

One goal of benchmarking Web servers is to discover bottlenecks that require optimization to improve performance. One way to improve performance is by removing overhead in the protocol itself. The W^{3}C 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 [18, 14]. The need for persistent connections to improve latency has been noted in [11]. In addition, latency can be improved by using caching proxies and caching clients, although the removal policy must be considered carefully [20]. The efforts presented in this paper are orthogonal to work on client-side caching. Our attempts to optimize performance have focused on (1) server-side caching (*e.g.,* via our `CVF` file system) and (2) utilizing concurrency and event dispatching processing routines that are customized to the OS platform in order to reduce server load and improve end-to-end quality of service.

6.3 Web Server Design and Implementation

Design patterns are re-usable software abstractions that have been observed to occur many times in actual solutions [3]. 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. Many of these ideas are applicable to Web server design.

Katz presents an NCSA prototype of a scalable Web server design [8]. 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 multi-threading 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. Yeager and McGrath of NCSA discuss the tradeoffs of some Web server designs and their performance in [13].

Our work attempts to apply design patterns and concurrency strategies provide a design which is both scalable and adaptive. We achieve static adaptation through start-up time configuration options, and dynamic adaptation through use of dynamically loadable modules which alter the behavior of the server.

6.4 Concurrency Strategies

A concurrency strategy specifies the policies and mechanisms for allowing multiple tasks to execute simultaneously. For Web servers, a task is an object that encapsulates server request handling. There are four general classes of Web server concurrency strategies:

- *Iterative* – An iterative server is a server that has no application-level concurrency: each task that begins runs to completion.

- *Single-threaded concurrent* – 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.

- *Thread-per-Request* – The Thread-per-Request strategy allows each task to execute in its own thread of control.

- *Thread Pool* – A Thread Pool approach fixes the number of executing threads and assigns tasks to available threads.

Surprisingly, Web server literature contains relatively little information on the performance of alternative concurrency strategies. This may result from the fact that most Web server implementations tightly couple their concurrency strategy with the other components in the server. In contrast, concurrency strategies have been studied extensively in the context of parallel protocol stacks that run on shared memory platforms.

[9] measured the impact of synchronization on Thread-per-Request implementations of TCP and UDP transport protocols built within a multi-processor version of the *x*-kernel; [12] examined performance issues in parallelizing TCP-based and UDP-based protocol stacks using a Thread-per-Request strategy in a different multi-processor version of the *x*-kernel; and [15] measured the performance of the TCP/IP protocol stack using a thread-per-connection strategy in a multi-processor version of System V STREAMS.

The results presented in this paper extend existing research on protocol stack parallelism in several ways. First, we measure the performance of a variety of representative concurrency strategies. Second, our experiments report the impact of synchronous and asynchronous event dispatching strategies on Web server performance, whereas existing work on parallel protocol stacks has focused on synchronous event dispatching. Our results illustrate how operating systems like Windows NT that support asynchronous event dispatching strategies and customized file/network transfer operations can perform significantly better than purely synchronous strategies under various workloads (*e.g.*, transfers involving large files).

7 Concluding Remarks

This paper describes *static* and *dynamic* adaptations that can be applied to develop high-perform Web servers. Common static adaptations include configuring an event dispatching model that is customized for OS platform-specific features (such as the I/O completion ports and `TransmitFile` on Windows NT). Common dynamic adaptations include prioritized request handling, caching strategies, and threading strategies. We illustrate the results of adapting the JAWS Web server framework to leverage the native asynchronous dispatching mechanisms provided by Windows NT.

Our results demonstrate that to alleviate bottlenecks, Web servers must utilize an integrated approach that combines optimizations at multiple levels of a Web endsystem. For example, a Web server should take advantage of special I/O system calls, specialized hardware and knowledge of the file access patterns.

As illustrated in Section 4, there is significant variance in throughput and latency under different server load conditions. For small files, the synchronous Thread Pool strategy provides better overall performance. However, under heavy loads and with large file transfers, the asynchronous model using `TransmitFile` provides more consistent quality of service. Thus, under Windows NT, an optimal Web server should adapt itself dynamically to either event dispatching and file I/O model, depending on the server's workload.

Acknowledgements

We would like to thank Chris Dawson and Dan Swanger of FORE Systems, Inc., for providing the ATM cards and ATM switch used for our benchmarking testbed. In addition, we would like to thank Sumedh Mungee, Darrell Brunsch, Everett Anderson, Tim Harrison, and John DeHart for helping to develop our ATM Web server testbed.

References

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

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

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

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

[5] Tim Harrison, Irfan Pyarali, 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 (Washington University technical report #WUCS-97-34)*, September 1997.

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

[7] Prashant Jain and Douglas C. Schmidt. Service Configurator: A Pattern for Dynamic Configuration of Services. In *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*. USENIX, June 1997.

[8] 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.

[9] Mats Bjorkman and Per Gunningberg. Locking Strategies in Multiprocessor Implementations of Protocols. *Transactions on Networking*, 3(6), 1996.

[10] *Microsoft Developers Studio, Version 4.2 - Software Development Kit*, 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] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation*. USENIX Association, November 1994.

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

[14] 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/.

[15] Sunil Saxena, J. Kent Peacock, Fred Yang, Vijaya Verma, and Mohan Krishnan. Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes. In *Proceedings of the Winter USENIX Conference*, pages 85–106, San Diego, CA, January 1993.

[16] 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.

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

[18] 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/.

[19] 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.

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

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