High-Performance Memory-Based Web Servers: Kernel and User-Space Performance

Philippe Joubert*, Robert B. King†, Rich Neves*, Mark Russinovich‡, John M. Tracey§

IBM. T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

Web server performance has steadily improved since the inception of the World Wide Web. We observe performance gains of two orders of magnitude between the original process-based Web servers and today's threaded Web servers. Commercial and academic Web servers achieved much of these gains using new or improved event-notification mechanisms and techniques to eliminate reading and copying data, both of which required new operating system primitives. More recently, experimental and production Web servers began integrating HTTP processing in the TCP/IP stack and providing zero copy access to a kernel-managed cache. These kernel-mode Web servers improved upon newer user-mode Web servers by a factor of two to six.

This paper analyzes the significant performance gap between the newer user-mode and kernel-mode Web servers on Linux and Windows 2000. Several user-mode and kernel-mode Web servers are compared in three areas: data movement, event notification, and communication code path. To establish a user-mode baseline, the paper measures the performance of highly optimized Web servers. The paper positions these user-mode implementations with those from related research projects. In particular, the "Adaptive Fast Path Architecture" (AFPA) is described and then used to implement kernel-mode Web servers on Linux and Windows 2000. AFPA is a platform for implementing kernel-mode network servers on production operating systems without kernel modifications. AFPA runs on Linux, Windows 2000, AIX, and S/390. The results show that kernel-mode performance greatly exceeds the performance of user-mode servers implementing a variety of performance optimizations. The paper concludes that significant opportunities remain to bridge the gap between user-mode and kernel-mode Web server performance.

1 Introduction

Increasing demand for Web content and services has motivated techniques to grow Web server capacity. As a result, Web server performance has steadily improved and Web-hosting infrastructure has become more complex. Today's Web "farms" are multi-tiered and employ several types of specialized server systems dedicated to caching static content, applications, and databases. For example, network service providers use proxy caches to geographically distribute content on behalf of specific customers. This reduces bandwidth costs and improves end-user response times. Akamai [1], for instance, has built a commercial service for caching portions of their customer's static content using geographically distributed edge caches. In addition to caching static content, Web servers are able to cache dynamic content such as price lists, stock quotes, or sports scores. Dynamic content often changes at a coarse enough frequency or requires publishing at intervals sufficient for caching. Commercial efforts [2] and research projects [3] have successfully exploited the ability to cache most forms of dynamic content at the front tier of a Web delivery architecture. While some forms of dynamic content have real-time publishing requirements and remain difficult to cache, it has been shown by [4] that the ratio between all forms of dynamic and static content has remained constant, defying the commonly-held belief that dynamic workloads will dominate over time.

Caching Web servers are well-suited to analyzing network server performance tradeoffs. These servers are

*Philippe Joubert and Rich Neves' current affiliation is ReefEdge Inc. email: `philippe,rich@reefedge.com`

†email: `rbking2@us.ibm.com`

‡Mark Russinovich's current affiliation is Winternals Software, 3101 Bee Caves Rd, Austin TX 78746. email: `mark@sysinternals.com`

§email: `traceyj@us.ibm.com`

simple to implement and measure with existing, unmodified static Web benchmarks. Caching Web servers reduce network server logic to parsing an HTTP GET request. It's possible to parse such a request with a few lines of C code. What remains is experimentation with thread models, scheduling mechanisms, and new operating system primitives to reduce memory copies. The simplicity of parsing HTTP GET requests reduces the complexity of the code required for kernel-mode caching Web servers. Furthermore, it's usually possible to implement kernel-mode caching Web servers without modifying the operating system kernel, providing useful control cases for assessing user-mode optimizations.

This paper analyzes the performance gap between the fastest currently available user-mode caching Web servers and their kernel-mode counterparts while holding the operating system and hardware fixed. The goal of this analysis is to identify the potential performance gains possible for future user-mode primitives. User-mode servers employing current "best practices" are used to establish a baseline for fastest possible user-mode performance. The tested user-mode servers employ several techniques to minimize data copies and reduce overhead of network event notification. The paper measures several kernel-mode Web servers, including servers based on a platform called "Adaptive Fast Path Architecture" (AFPA). The experiments are repeated for two different operating systems: Linux 2.3.51 and Windows 2000. In all cases, the CPU hardware, TCP/IP stack, network hardware and operating system are held constant.

The paper compares the performance of several user-mode and kernel-mode caching Web servers using different workloads. The results show a wide performance margin between the better performing user-mode and kernel-mode servers. The user-mode servers are shown effective in reducing memory copies and reads while also reducing scheduling overhead with efficient event notification mechanisms and single thread, asynchronous I/O implementations. The best of these efforts are still two to six times slower than the fastest achieved kernel-mode performance on the unmodified Linux and Windows 2000 operating systems tested using the same hardware. The results reveal significant potential to improve user-mode server performance.

The paper is organized as follows: Section 2 classifies Web server performance issues, describes current user-mode and kernel-mode approaches, and describes related work. Section 3 describes the Adaptive Fast Path Architecture, a platform for building kernel-mode network servers. Section 4 describes the methodology used to measure and analyze user-mode and kernel-mode Web server implementations. Section 5 reports and analyzes the performance results for representative user-mode and kernel-mode Web servers. Finally, Section 6 draws conclusions from the performance analysis, and Section 7 makes recommendations regarding Web server design, and describes future work.

2 Web Server Performance Issues

Techniques to improve caching Web server performance address one or more of the following objectives: elimination of data copies and reads, reduction of scheduling and context switching overhead due to event notification, and reduction of overall communication overhead in the socket layer, TCP/IP stack, link layer, and network interface hardware.

2.1 Data Copies and Reads

Eliminating copies and reads for a Web transaction offers significant performance improvements for large responses. Data copies can be difficult to avoid in user-mode Web servers where the data to be sent resides in the file system cache. In cases where data is already mapped into the user-mode address space, BSD-style socket implementations will perform one or more copies before delivering the data to the network adapter. Even where data copies are eliminated, the additional overhead in reading the data to compute a checksum remains. The data copy problem is solved by providing a mechanism to send response data directly from the file system cache to the network interface. The checksum problem is solved either by precomputing and embedding the checksum in a Web cache object or by relying on network interface hardware to offload the checksum computation.

2.2 Event Notification

A second performance issue is minimizing the cost of event notification. We define event notification as the queuing of a client request by a server for response by a server task. In the case of a HTTP 1.0 request, the client request is formed by the arrival of a TCP SYN packet and subsequent data packets containing the request. The server must handle these two events with minimal overhead. First, it must complete the three-way TCP handshake started with the arrival of the first SYN packet. Second, it must receive the data forming the request, and read this data into a user-mode memory area. To handle multiple clients, the server supports concurrency either by assigning a single task from a pool of tasks to each client request or by using asynchronous system calls to manage many requests with a few tasks.

Achieving efficient event notification requires a thread model with minimal scheduling overhead. The mapping between threads and requests is taxonomized by [5] as multiple process/thread (MP) or single process event driven (SPED). In the MP model, a server creates a new task for each new request. Because creating a new task can be time consuming, most MP servers reduce the overhead by pre-allocating a pool of tasks. However, pre-allocating a pool of tasks to avoid task creation still incurs unwanted scheduling overhead. Every request requires a reschedule to the task for that request. Apache [6] is the canonical example of the MP architecture. Ideally, the unnecessary scheduling inherent to the MP model is avoided in a design where a single task services requests on behalf of multiple clients.

In the SPED model, a few processes handle requests from multiple clients concurrently. The SPED model relies on asynchronous notification mechanism for notifying a server task of incoming network requests. For example, `select()` is the event notification mechanism commonly used on user-mode UNIX Web servers and I/O completion ports are commonly used on Windows 2000. Web servers such as Zeus [7], IIS [8] use a SPED model. Flash [5] also uses a SPED model for cached content, using only one thread for serving cache hits. The Windows 2000 APIs implementing zero copy data transfer and efficient event notification are described in [9].

2.3 Communication Code Path

A third performance issue is the overall communication code path through the socket layer, TCP/IP stack, link layer, and network interface. The socket layer is not necessarily tailored to the needs of Web servers. Researchers have modified existing socket APIs or implemented new APIs with Web servers in mind [10].

Some commercial operating systems provide interfaces specifically for Web servers. In particular, Windows NT provides interfaces eliminating redundant system calls: `AcceptEx()` and `TransmitFile()`. In addition to other benefits, these interfaces aggregate several system calls, reducing the code path between the Web server and TCP/IP stack. For example, `AcceptEx()` combines accepting a new connection, reading the request data, and obtaining peer address information, eliminating system calls and redundant socket layer code. Likewise, `TransmitFile()` combines reading data from the file system and writing header and data to the socket, also eliminating system calls.

In addition to socket layer optimizations, the TCP/IP stack has also been improved for Web server workloads. For example, TCP/IP implementations have been redesigned to efficiently manage short-lived connections [11]. Commercial operating systems have also been optimized for short-lived connections, improving management of TCP control blocks and sockets in the TIME_WAIT TCP/IP state.

The network interface hardware and corresponding driver are other key places for optimizations in the communication code path. For the purposes of this paper, we hold the network interface and driver implementations fixed when comparing servers on the same operating system. For completeness, it should be noted that network interfaces and their drivers can have a significant impact on performance. "Smart" network interface cards with on board processors are capable of coalescing interrupts, offloading fragmentation, checksum computation, and higher level TCP/IP processing such as connection establishment [12].

2.4 Current Approaches

User-mode Approaches

This paper analyzes several user-mode Web servers taking advantage of the performance enhancements described above. These user-mode Web servers rely heavily on the operating system to provide the primitives necessary to reduce data movement, limit event notification overhead, and minimize the communication code path. One approach is to optimize existing interfaces and their implementations. For example, implementations of `select()` and `poll()` have been improved by [13, 14] to reduce event notification overhead.

Another approach is to define completely new interfaces, building Web servers around these new interfaces. For example, new user-mode interfaces to eliminate memory copies and mitigate checksum computation include IO-Lite [15] and Windows NT's `TransmitFile()` API. IO-Lite provides a generic interface and mechanism to unify data management among operating system subsystems and user-mode servers. `TransmitFile()` provides the same performance effect in avoiding data copies, but is limited to sending files with prefix or suffix data from the file system cache. AIX implements a similar zero copy API called `sendfile()`. Linux provides a `sendfile()` API, but the implementation requires a data copy to move the data from the file system to the network stack. This paper analyzes the performance of IIS and Zeus, two production Web servers leveraging examples of these new APIs. Lastly, the paper describes an additional SPED user-mode Web server for Windows 2000 and Linux called Howl. Howl is described in more detail in Section 5.3.

Kernel-mode Approaches

Kernel-mode Web servers have been implemented in the context of both production and experimental operating systems. Migration of services considered integral to a server's operation into the kernel is not a new idea. For example, most commercial operating systems include kernel-mode file servers. Delivery of static Web responses amounts to sending files on a network interface and does not require extensive request parsing. A kernel-mode Web server can fetch response data from a file system or kernel-managed Web cache. If the kernel-mode caching Web server determines that it cannot serve the request from its cache, it forwards it to a full-featured user-mode Web server.

Kernel-mode Web servers can be characterized according to the degree of their integration with the TCP/IP stack and whether responses are derived in a thread or interrupt context. Microsoft's Scalable Web Cache (SWC) [16] is tightly integrated with the Windows 2000 TCP/IP stack. By contrast, Linux's kHTTPd [17] uses socket interfaces in kernel-mode. Both SWC and kHTTPd handle response processing from kernel-mode threads. TUX [18] is another in-kernel Web server introduced by RedHat on Linux. Like kHTTPd, TUX uses a threaded model, but it offers greater features and performance. First, TUX caches objects in a pinned memory cache rather than using the file system. Second, TUX implements zero copy TCP send from this pinned memory cache and a checksum cache for network adapters without hardware support for offloading checksum computation.

Other Approaches

In addition to extending production operating systems, researchers have implemented specialized or new operating systems to experiment with Web server performance. The Lava hit-server [19] achieves cache performance limited only by memory bus bandwidth. While this paper holds the TCP/IP stack, network driver, and network hardware fixed, the hit-server focuses on network driver optimizations and a non-TCP transport protocol to minimize memory conflicts between the CPU and network controller for performance. The Cheetah Web server [20] is another example of a Web server designed with performance in mind on a new operating system. Rather than extending production systems as described in this paper, the Cheetah Web server uses a specialized TCP stack on a research operating system called Exokernel [21]. The Exokernel approach demonstrates the performance possible when subsystems such as the network stack and file system are tightly integrated. The AFPA results in this paper appear consistent with the factor of three to six performance gain reported for Cheetah on the Exokernel operating system. However, the AFPA results use unmodified, production operating systems allowing direct comparisons with state of the art user-mode optimizations on Linux and Windows 2000.

3 AFPA

We now provide a brief overview of the Adaptive Fast Path Architecture (AFPA) [22], a software architecture for high-performance network servers. AFPA features:

- Support for a variety of application protocols.
- Direct integration with the TCP/IP protocol stack.
- A kernel-managed, zero copy cache.

3.1 Overview

AFPA is a flexible kernel-mode platform for high-performance network servers. The architecture is flexible in several ways. First, it can be applied to a variety of application protocols such as HTTP, FTP, LDAP, and DNS. Such protocols are implemented as AFPA modules. Second, it has been implemented on four platforms: Linux, Windows 2000, AIX, and OS/390. The latter three implementations have been incorporated into current IBM products, the first of which was released as the Netfinity Web Server Accelerator in 1998. The architecture was implemented on Linux and Windows 2000 solely as a kernel module. Third, it can be used as a caching server or an efficient layer 7 router. Fourth, it can be tightly integrated and co-located with a conventional user-level network server or implemented as a stand-alone front-end accelerator that offloads processing from a set of "back-end" servers without requiring any modification to the conventional servers. Fifth, AFPA can be used to enforce quality of service [23]. This section focuses on AFPA application to Web servers.

Several factors contribute to AFPA's efficiency. These factors are now described in terms of data movement, event notification, and communication code path for the

Linux and Windows 2000 AFPA implementations. First, data copies and reads are eliminated, improving performance when sending larger responses. The data copies are avoided by passing references to pinned cache objects directly to the protocol stack. Reads are eliminated by avoiding checksum computation when sending cache objects as responses. On Windows 2000, checksum computation is off-loaded to the network interface hardware. On Linux, cache objects include pre-computed checksums.

Second, scheduling and context switching overhead in responding to TCP/IP events is significantly reduced or eliminated using AFPA. AFPA parses requests on the same software interrupt on which TCP/IP processing occurs. AFPA then sends corresponding responses from the same interrupt context or queues the response for sending in a thread context. In implementations where responses are derived from software interrupt context, no scheduling or context switching overhead is incurred. As shown in Section 5, responding from software interrupt provides better performance, but responses must reside in pinned memory. The AFPA module can also use a thread-based configuration where responses are sent from a thread context. This approach mitigates livelock problems inherent to the software interrupt approach [24]. A hybrid approach has also been implemented. Requests for content not currently pinned are processed on software interrupt, but unpinned responses are sent from a kernel-mode thread context where page faults are tolerated.

Third, the overall communication overhead incurred in AFPA implementations is less than a user-mode server relying on a socket API. AFPA interfaces directly with TCP/IP by overloading TCP/IP events with HTTP specific processing. These events include connection establishment (SYN), data arrival, and disconnection (FIN). In overloading these events, AFPA drives a state machine associated with the protocol modules such as HTTP. Direct integration with TCP avoids the queueing and descriptor management incurred using a socket API.

3.2 State Machine

Using a state machine interface, it is possible to implement a variety of protocols in the AFPA framework. Protocol specific code for a given application such as Web serving is encapsulated in an AFPA module.

Each AFPA module is partitioned into three components: state, program actions, and control elements. AFPA manages three types of internal state data for each module instance: global data, connection data, and request data. Global data is data independent of each connection and request. Connection data is information that changes on a per connection basis. Request data is information that changes per request within the same connection. State functions are either written by the module developer, exist in another AFPA module's library of state functions, or are intrinsic to AFPA as a general service. State functions indirectly manage state transitions through a separately managed table. The layer of indirection permits state functions to be reusable and reconfigurable.

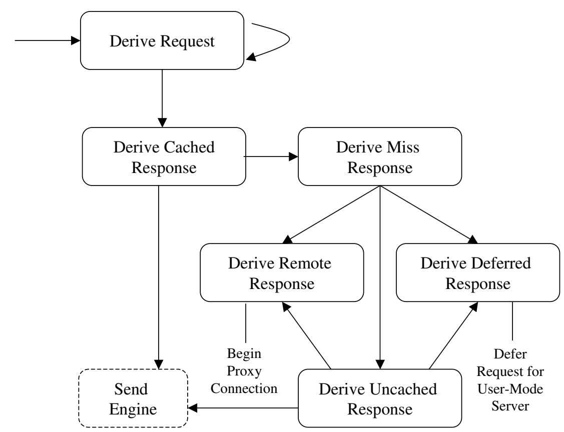

Figure 1: AFPA HTTP State Machine

A simplified version of the state machine used by the AFPA HTTP module is shown in Figure 1. The state machine illustrates how the HTTP module can be used as a standalone Web cache, work in conjunction with a user-mode Web server, or act as a front end Web cache for one or more back end Web servers. The module implements a function corresponding to each of the following client connection states:

- `Derive Request`

  This function is invoked when new data is received on a connection. When sufficient data has arrived to complete a well-formed request, the function computes the parsed request, storing it in the connection data structure.

- `Derive Cached Response`

  This function uses the parsed request in the connection data structure to perform a key-based lookup in the AFPA cache. If successful, the cache object is sent. If unsuccessful, the connection transitions to `Derive Miss Response` state.

- `Derive Miss Response`

This function is invoked when a response is not found in the AFPA cache. The HTTP module either performs necessary file I/O to create a new cache object (`Derive Uncached Response`), passes the request to a local user-mode Web server (`Derive Deferred Response`), or sends the request to a remote Web server (`Derive Remote Response`).

- `Derive Uncached Response`

  This function creates a cache object and response header. The file is either read into the newly created cache object or a reference to the file system cache is created depending on the AFPA implementation. Once created, the cache object is sent to the client. Any errors creating the cache object may result in sending request to another Web server or generating an error response.

- `Derive Remote Response`

  This function routes the parsed request to a remote server. In versions of AFPA modules acting as content-based routers, this function expands to an alternate state machine for managing connections to other Web servers.

- `Derive Deferred Response`

  This function routes the parsed request to a user-mode server which takes control of the connection.

- `Send Engine`

  The `Send Engine` is a function invoked by the `Derive Cached Response` or `Derive Uncached Response` states. The send engine sends the cache object. Persistence, serialization, and fragmentation of large requests are managed by AFPA. This function is exported by the AFPA runtime system.

3.3 TCP/IP Integration

A key aspect of AFPA is its close integration with the TCP/IP protocol stack. Layer 7 protocols are processed on the same software interrupt on which TCP/IP input processing occurs. To achieve this, AFPA relies on the ability to extend the TCP/IP stack through callback mechanisms. AFPA cache objects use native TCP/IP data structures such as BSD mbufs, Linux skbuffs, or Windows 2000 MDLs (lists of chained page frames used to describe buffers) [25].

3.3.1 Linux TCP/IP Integration

AFPA on Linux intercepts events in the TCP/IP stack without kernel modification. The Linux kernel socket structure contains function pointers which are invoked whenever the state of the connection changes, inbound data is queued on the socket, or outbound data is removed from the socket queue. AFPA on Linux replaces the data arrival hook with its own. When the first data packet arrives on the socket, the main AFPA hook gets called from within the network bottom half (i.e. software interrupt handler). AFPA then parses the request packet, looks up the cache object, queues the response in the socket output queue, and sends the response. If the request does not fit entirely into the first packet, AFPA creates a connection context which it retrieves when the next packet arrives and parses the request.

To manage sending data, AFPA on Linux rewrites the skbuff free hook, which is called whenever a network buffer is freed. This hook is used to send responses in 64 kB chunks. The last packet of a chunk is flagged. When the AFPA hook detects that last packet of a chunk, the AFPA hook sends the next chunk. When a request is not found in the cache, the request is queued and picked up by a service kernel thread. Tight synchronization has to be provided between the thread and the software interrupt which are competing for the socket's accept queue.

AFPA on Linux allocates a number of 128 KB blocks of pinned memory which are managed by AFPA's own memory allocator. When APFA creates a cache object, it opens the corresponding file and reads it into Ethernetframe-sized buffers. AFPA allocates space in each buffer for TCP, IP, and Ethernet headers. It reserves additional space in the first buffer for HTTP headers. When APFA receives a request, it fills in the HTTP header and queues each frame associated with the object for transmission. For large responses, AFPA queues frames in 64 KB chunks. If an additional request is received for a cache object that is in the process of being sent, AFPA makes an additional copy of the buffer (i.e. Ethernet frame).

3.3.2 Windows 2000 TCP/IP Integration

On Windows, AFPA interacts with the TCP stack using the Transport Driver Interface (TDI) [25]. TDI is an interface, defined by Microsoft, by which kernel-mode "clients" interact with protocol drivers such as TCP/IP. TDI defines a set of client requests, including accept, connect, disconnect, send, and receive. It also defines a set of callback routines, each associated with a network event such as connection establishment, disconnection, and reception. TDI allows but does not require a client to register a callback routine for any event per connection end point. TDI uses an asynchronous model. Each request has an associated client-specified completion routine that is invoked when the request completes (whether synchronously or asynchronously). Client-registered callback routines are invoked asynchronously as well.

When a TCP SYN packet arrives on a port to which AFPA is bound, TDI invokes AFPA's connect event handler. This routine allocates an AFPA connection structure, in which application-specific information associated with the connection is stored, then builds and returns an accept request. The completion routine for the accept request simply cleans up in case of error. Arrival of request data from the client causes AFPA's receive event handler to be invoked.

On Windows 2000, AFPA reads cache objects from the file system, pins them in memory, and passes them to the TCP stack in 64 KB chunks. Each chunk is represented by an entry in the cache object's pin array. When AFPA creates a cache object, it opens the corresponding file. AFPA initializes each entry in the pin array to indicate that the corresponding chunk has not yet been read or pinned. When it sends a cache object, AFPA sequentially traverses the object's pin array. If a chunk is currently pinned, its pin count is incremented, and it is sent immediately in the context of a software interrupt. If the chunk is not pinned, the request must be queued to a pool of worker threads because the Windows 2000 memory architecture does not permit access to unpinned memory from software interrupts. The send completion routine decrements the pin count (which acts as a reference count) for the current chunk and repeats the process for the next chunk.

3.4 Cache Architecture

A complete description of the AFPA cache is beyond the scope of this paper. We do, however describe several aspects particularly relevant to response processing. First, AFPA cache objects contain mutable header data, immutable header data, and data payload. On Windows 2000, the cache object is represented as an MDL. Cache objects do not include precomputed checksums because Windows 2000 supports a number of network adapters that offload checksum computation as well as payload fragmentation. On Linux, the cache object is presented as a list of skbuff structures with precomputed checksums. This architecture allows zero copy send operations, not supported by the Linux 2.2 kernel. Second, AFPA cache objects are opaque data types which can be backed by any number of memory systems. For example, it is possible to implement AFPA cache objects using Windows 2000 support for x86 PAE (Physical Address Extensions) mode and manage a cache of up to 64 GB in size. It is also possible to back AFPA cache objects directly with the file system cache, thereby leveraging the system cache for selected cached files. Third, the cache architecture supports a hybrid approach to handling responses at software interrupt or kernel thread. This allows a two level cache where frequently accessed small files remained pinned, allowing them to be delivered in a software interrupt context, while larger and less frequently accessed files are served using threads. Finally, cache objects are divided into fragments. Each fragment can be pinned and passed to the TCP/IP stack independently.

4 Experimental Methodology

In this section, we present experimental methodology used to compare user-mode Web server performance with kernel-mode implementations based on the AFPA framework described in the previous section. The goal of the methodology is to establish a baseline for user-mode performance and compare the best performing user-mode approaches with Web servers based on AFPA. We compare several user-mode and kernel-mode HTTP servers: Apache 1.3.9 [6], Zeus 3.3.5 [7], IIS 5.0 [8], and two experimental Web servers referred to as Howl. We also consider other kernel-mode Web servers: kHTTPd [17] and TUX [18] on Linux, and Microsoft's SWC 2.0 [16].

4.1 Workload

We use two different synthetic workloads for our experiments: SPECWeb96 [26] and WebStone 2.5 [27]. SPECWeb96 was the first standard HTTP benchmark. The SPECWeb96 working set comprises files that range in size from 100 bytes to 900 kB, where small files are referenced more often than large files (50% of the total number of requests reference files smaller than 10 kB). In addition, the SPECWeb96 working set scales with the expected server throughput. In all of our experiments, the entire working set fit into the server's RAM, thus avoiding any performance distortion due to disk accesses.

SPECWeb96 has been superseded by SPECWeb99 as the industry-accepted Web serving benchmark. SPECWeb99 exercises HTTP/1.1 features, such as persistent connections, and includes requests for dynamically generated pages.

Although SPECWeb96 does not take into account some aspects of current HTTP workloads (e.g. no persistent connections, no dynamic content), it is well suited for measuring static file serving performance, which is the main purpose of our performance evaluation. Furthermore, large HTTP sites often use several servers that are partitioned into groups serving different types of content such as static files, user logins, and databases. The static content servers are likely to experience workloads similar to the SPECWeb96 workload. Finally, the SPECWeb96 execution guidelines are sufficiently strict as to allow meaningful comparison of independently reported results.

The results presented here do not meet SPECWeb96 reporting guidelines and are not certified SPECWeb96 results. The SPECWeb96 benchmark was executed for the largest workload corresponding to the reported result rather than ten evenly spaced lower throughput workloads as required by SPECWeb96 for reporting purposes. This does not affect the results reported in this paper.

WebStone is another HTTP server benchmark. Unlike SPECWeb96, it allows a user to change the workload characteristics, making it easier to identify performance bottlenecks for given file sizes. For WebStone, our workload consists of fixed-size files, ranging from 64 bytes to 1 MB. The file size is varied in each test.

4.2 Test Environment

Experiments were performed on two operating systems: Windows 2000 Advanced Server (build 2195) and RedHat Linux 6.1 with a Linux 2.3.51 kernel. One server, TUX, which does not run on a Linux 2.3.51 kernel, was run on a Linux 2.4.0 kernel instead. AFPA on Windows 2000, IIS, and SWC were run on Windows 2000. AFPA on Linux, kHTTPd, TUX, Zeus, and Apache were run on Linux. To quantify the benefit of serving responses in a software interrupt context, a version of AFPA that does not include this optimization and instead serves all responses using kernel threads was implemented.

All experiments were performed on the same server hardware: an IBM Netfinity 7000 M10 with four 450 Mhz Pentium II Xeon processors, 4 GB of RAM and four Alteon ACEnic gigabit Ethernet adapters. The server hardware has two 33 Mhz PCI buses (one 32 bit and one 64 bit). Each PCI bus had two gigabit Ethernet adapters. Distributing these adapters over the two PCI buses was necessary to maximize the bandwidth of the memory bus. For all experiments, only one of the server's four CPUs were used. The presence of three empty CPU sockets does not interfere with the uniprocessor experiments. Ten client machines were used to generate load. The clients were IBM Intellistation Z-Pro systems with two 450 Mhz Pentium II Xeon processors, 256 MB RAM), and a single Alteon ACEnic gigabit Ethernet adapter. The clients ran RedHat Linux 6.1 and were connected to the server via a pair of Alteon ACEswitch 180 gigabit Ethernet switches.

The Netfinity 7000 M10 supports up to 280 MB/s memory to memory bandwidth based on timing `memcpy()`. In practice, the tested Netfinity hardware is at most capable of 200 MB/s bandwidth from main memory to the PCI buses. Including TCP/IP headers, HTTP request, and HTTP response, the maximum possible SPECWeb96 result is 11,400 requests per second.

All experiments were run using 9000 byte (jumbo) Ethernet frames. We chose jumbo Ethernet frames rather than standard 1500 byte Ethernet frames since it allowed our SPECWeb96 results to be compared with officially published results [26]. Limited experiments using standard Ethernet frames did not reveal in any significant difference in the performance trends seen with 9000 byte frames.

We note the following limitations of our test methodology. All experiments were performed with the same limited number of client machines. Our results focus almost entirely on uniprocessor rather than multiprocessor servers. Experiments were performed solely with non-persistent connections. Our analysis is constrained to static content only. Finally, results are reported only for the Linux and Windows 2000 operating systems running on the same Intel processor.

4.3 Performance Tuning

On the server side, Linux and Windows 2000, as well as each individual Web server, were tuned to achieve maximum performance. To this end, we used tuning parameters provided with submitted SPECWeb96 results. For servers that support time-to-live values for cached objects, we set the timeout to eliminate cache invalidations. This ensures we achieve a 100% hit rate.

5 Performance Analysis

Two benchmarks are used to compare the user-mode and kernel-mode Web servers. SPECWeb96 is used for comparing workloads with mixed file sizes. Webstone is used to compare performance for fixed file sizes.

5.1 SPECWeb96 workload

The results for the SPECWeb96 workload are presented in Figure 2. Results are presented for the following user-mode Web servers: Apache (Linux), Zeus (Linux), IIS (Windows 2000). Results are also presented for the following kernel-mode Web servers: kHTTPd (Linux), TUX (Linux), SWC (Windows 2000), AFPA on Linux, and AFPA on Windows 2000. Table 1 enumerates and describes the Web servers tested on Linux and Windows 2000.

	architecture	cache	0 copy	direct TCP
Apache	MP/user	filesystem	no	no
Zeus	SPED/user	filesystem	no	no
IIS	SPED/user	filesystem	yes	no
kHTTPd	SPED/kernel	filesystem	no	no
TUX	SPED/kernel	memory	yes	no
SWC	SPED/kernel	fs or mem	yes	yes
AFPA	softint/kernel	fs or mem	yes	yes

Table 1: Web Server Characteristics

The "architecture" column describes servers as MP, SPED, or softint (software interrupt) as defined in Section 2.2 and kernel-mode or user-mode. The "cache" attribute defines whether the Web server's cache is backed by the file system, memory, or both. The "0 copy" column indicates whether or not the Web server performs a copy to send a cache object. The "direct TCP" column indicates whether or not the Web server is directly integrated with the TCP/IP stack or uses the socket layer.

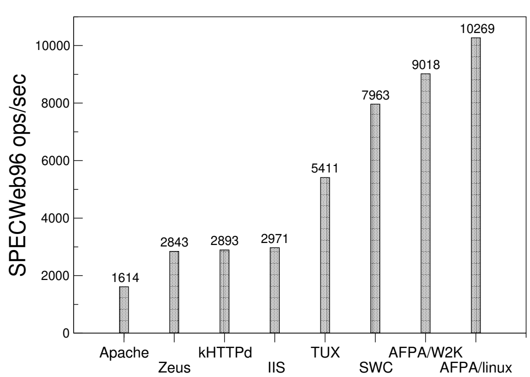

Figure 2: SPECWeb96 results

The results show that AFPA on Linux achieves the fastest performance of the tested servers. The SPECWeb96 result of 10,269 represents over 1.2 Gb per second server throughput. This amounts to 90% of the hardware capacity as described in section 4.2. Therefore, AFPA on Linux is a reasonable performance target for evaluating user-mode optimizations.

Kernel-mode servers appear to have a factor of three performance advantage over production user-mode servers. On Linux, the fastest user-mode server measured (Zeus) is 3.6 times slower than the fastest kernel-mode server (AFPA). On Windows 2000, the fastest user-mode server measured (IIS) is 3 times slower than the best kernel-mode server (AFPA). Overall, the fastest kernel-mode implementation (AFPA on Linux) is 3.5 times faster than the best performing user-mode implementation measured (IIS on Windows 2000).

The slowest user-mode server in the SPECWeb96 results was Apache. This is consistent with other published results [5, 28]. Apache seems to be penalized by a significant process scheduling overhead. Note, however, that Apache 1.3.9 does not feature a memory-based static content cache; it uses the file system cache. Among other optimizations, adding a memory based cache to Apache reportedly increases its performance by 70% on Linux [29] which would bring Apache in line with IIS and Zeus.

As mentioned before, both IIS and Zeus employ SPED architectures. Although Linux does not feature zero copy send, Zeus was on par with IIS. This somewhat contradicts previous attempts at comparing user-mode Linux and Windows Web servers [28]. These earlier results were, however, obtained with Apache which exhibits lower performance than Zeus.

In comparing kernel-mode servers, we found kHTTPd to be relatively slow on the SPECWeb96 workload compared to other kernel Web servers. In fact, kHTTPd achieves nearly the same SPECWeb96 result (2893) as Zeus (2843). kHTTPd has limited performance for two reasons. First, kHTTPd requires one copy to send the response. This is unavoidable due to kHTTPd's reliance on the file system as a cache. The Linux file system interfaces lack a zero copy mechanism to send file system data. Second, kHTTPd uses the kernel-mode version of the Linux socket interface rather than interfacing directly with the TCP/IP stack. Therefore, kHTTPd's performance is not significantly different than Linux user-mode Web servers, which are also forced by Linux to use a one-copy send and a socket interface.

TUX offers nearly twice the performance of kHTTPd, due primarily to its pinned memory cache and zero copy send implementation. TUX and kHTTPd have otherwise similar architectures for serving static content out of main memory. They both use the socket API from kernel threads for sending objects.

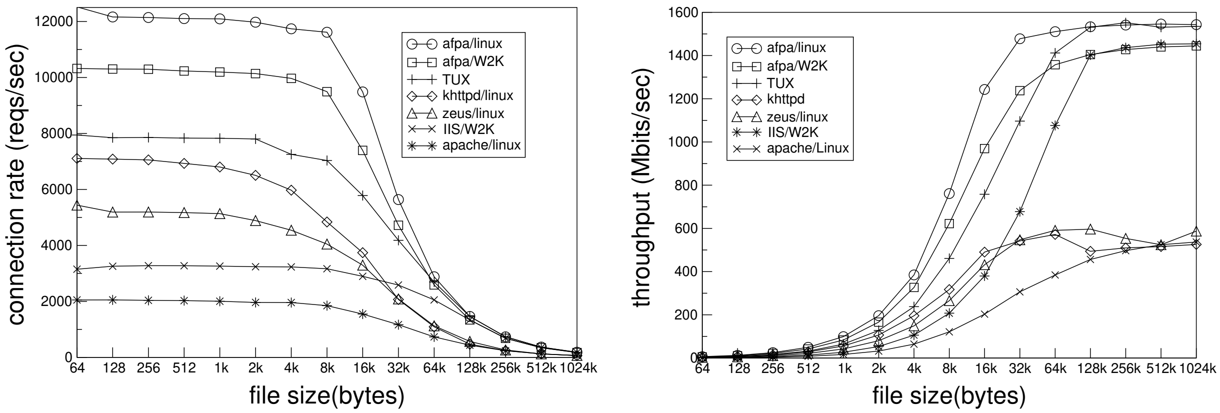

Figure 3: Fixed file size performance results

5.2 Analyzing a Fixed File Size Workload

We next studied how performance varied as a function of response size using a set of file sizes ranging from 64 bytes to 1 MB. The experiments were performed using the Webstone benchmarking tool. The connection rates and delivered bandwidths are reported in Figure 3. For small files, request latency was the dominant performance factor.

Linux vs. Windows 2000

For 64 bytes files, AFPA on Linux was 21% faster (12,522 requests per second) than AFPA on Windows 2000 (10,321 requests per second). In addition to obtaining the requests per second, we also used the Intel processor performance counters to measure several metrics under the same workload. We found two metrics significantly different between AFPA on Linux and Windows 2000: the number of instructions executed and instruction TLB misses. Because the average cycles per instruction were nearly identical for both cases, we conclude that the instruction count is a useful metric for comparing the two implementations. In addition, both AFPA implementations used the exact same source code to implement the HTTP and caching logic. This implies that any differences between the two would have to be limited to the interfaces used to integrate AFPA in the TCP/IP stack, the TCP/IP stack itself, and network driver. AFPA on Linux executed 19% fewer instructions than AFPA on Windows 2000 (26,000 versus 31,000 instructions per request). We also find that the number of instruction TLB misses was ten per request on Windows 2000 versus zero per request on Linux. The Linux kernel, TCP/IP stack, and kernel modules are stored entirely in non-pageable 4 MB pages, so it does not experience any instruction TLB misses. Only the Windows 2000 kernel is mapped using 4 MB pages; the TCP/IP stack is not.

Thread vs. Software Interrupt

Using the Pentium performance counters, we also compared the software interrupt-version of AFPA with the threaded version of AFPA on Windows 2000. For 64 byte files the software interrupt version was 12% faster than the threaded version (10,321 versus 9,209 requests per second). This closely matches the difference in the number of instructions executed. This difference corresponds to the overhead of queueing/dequeuing work items and scheduling the thread.

Effect of File Size

For large files, performance is determined primarily by the speed at which the server can move data to the network. As file size increases, the operating system overhead for user-mode servers accounts for less and less in the overall cost of processing requests. This is because processing latency is completely amortized for large files. For example, Apache lags behind the other Web servers for performance on small files, but is just as good as other user-mode Linux servers for large files. On Windows 2000, IIS performs as well as AFPA for files 128 kB and larger, while it is 3.27 times slower than AFPA for 64 byte files.

For user-mode Web servers, IIS was slower than Zeus for files smaller than 32 kB, but for larger files gained an

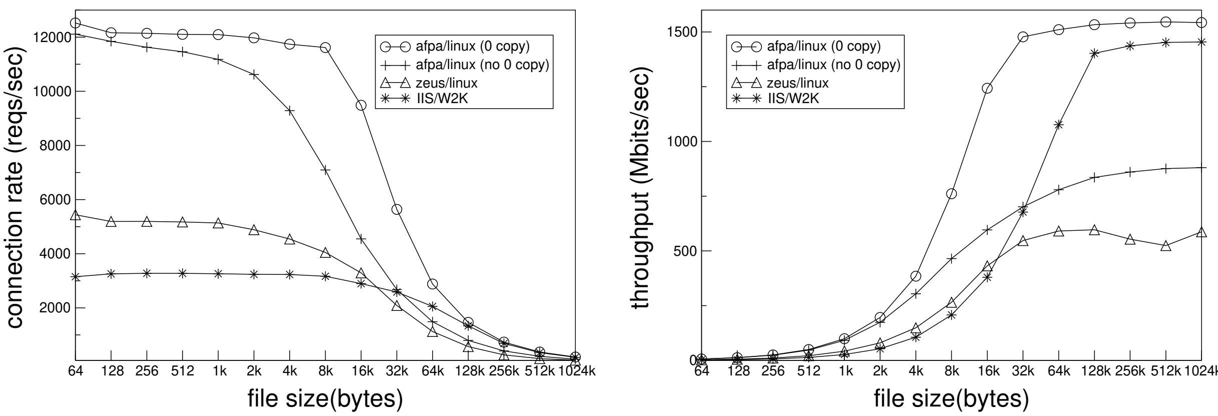

Figure 4: Efficiency of zero copy TCP send

advantage from having a zero copy send interface. The importance of a zero copy TCP send is further emphasized on the throughput graph. There is almost a three-fold performance difference between Web servers using zero copy send interfaces and those using a one-copy send interface. TUX achieved three times the throughput of kHTTPd for large files. We also ran a modified version of AFPA on Linux that does not use the AFPA zero copy architecture. Its throughput on large files was half that of the zero copy version.

Jumbo Frames

Another interesting result is the nearly flat connection rate for transfers less than 8 kB. Even with jumbo frames enabled one would expect a more significant decrease in the connection rate. It appears that the Alteon firmware is optimized for bulk data transfers rather than fast connection set-up. We ran some tests using a single client thread requesting 1 kB transfers on Alteon, Intel gigabit, and Intel 100 Mb adapters. This configuration measures connection latency. We found the Alteon adapter to be between two and three times slower than the Intel gigabit and 100 Mb adapters, respectively. The high connection set-up cost on the Alteon adapter most probably accounts for the flat connection rate.

Zero copy TCP send

In order to evaluate the performance gain of a zero copy send interface in the TCP/IP stack, we ran a modified version of AFPA on Linux that does not use the AFPA zero copy cache architecture. In this version, network buffers are allocated through the standard Linux `sock_wmalloc()` primitive; file data is copied from the AFPA cache into network buffers and checksummed before being sent. Figure 4 summarizes the performance of these two implementations plus Zeus on Linux (which does not use zero copy sends) and IIS on Windows 2000 (which does zero copy sends through the `TransmitFile()` API).

As expected, the performance advantage of a zero copy send interface increased with the file size. It is important to note, however, that the benefits of a zero copy interface can be seen for relatively small files. For 4 kB files the performance difference is 25% and then grows to 111% for 32 kB files.

For full efficiency, a zero copy send interface also requires a network adapter with outbound packet checksumming capability (such as the Alteon adapter used in our test bed) in order to avoid reading the data to checksum it.

5.3 Howl

In defining the best possible user-mode performance it's important to not rely solely on commercial user-mode examples for performance analysis. To that end a user-mode Web server was implemented using the best practices for performance on Linux and Windows 2000. This user-mode Web server is referred to as Howl. Howl is an attempt at estimating the maximum performance that can be achieved by a user-mode server on current versions of Linux and Windows 2000 using standard APIs. On Linux, Howl is a simple loop executing four system calls to process an HTTP request: `accept()`, `read()`, `write()`, and `close()`. It uses a user-mode version of the AFPA cache for storing responses (with pregenerated HTTP response headers). Howl offers very limited functionality and performance (no logging, only one request is processed at a time). Howl is a test case. It is not intended for general use. But under the assumption that (i) all requests fit into the first data packet, (ii) all requests hit in the cache, (iii) all responses can be sent with a non blocking write (by configuring a sufficiently large socket buffer), and (iv) consecutive client requests do not block the server task receiving the request, Howl has performance close to the best achievable using the standard Linux APIs. On Windows 2000, Howl uses `AcceptEx()`, `TransmitFile()`, and I/O Completion ports to achieve best possible user-mode performance.

Compared to the user-mode results shown in Figure 3, Howl on Linux is 32% faster than Zeus and Howl on Windows is 21% faster than IIS for 64 byte files. Likewise, for 1 kB files, Howl on Linux is 29% faster than Zeus and Howl on Windows is 16% faster than IIS. Given that Zeus and IIS are production-level web servers, it is not surprising to marginally improve upon their performance with minimal prototypes such as Howl. However, the 16% to 29% performance improvement using such prototypes for 1 kB files is significantly smaller the 235% to 312% gap between these servers and their kernel-mode counter parts. Given the size of this gap, further significant performance improvements appear unlikely without new user-mode APIs and operating system modifications.

6 Conclusion

The paper showed the performance of several highly optimized user-mode Web servers, comparing these servers to multiple kernel-mode Web servers while holding TCP/IP and network driver implementations fixed. The paper concludes that the best performing user-mode Web servers are at least two times slower than the faster kernel-mode server on the same hardware and unmodified operating system. The best kernel-mode results were achieved using a software interrupt approach where responses are sent on software interrupt (Linux bottom half handler or Windows 2000 deferred procedure call) rather than a separately scheduled thread. The results showed that software interrupt based kernel-mode servers perform 10% to 20% better than Overall, the best performing Web servers share three attributes. First, they use a zero copy interface between cache and network without TCP checksum computation to efficiently serve responses greater than 4 kB in size. Second, these servers use an efficient event notification mechanism to serve responses less than 4 kB in size with minimal scheduling overhead. Third, these servers minimize communication code path using new socket APIs or eliminating the socket layer altogether.

7 Future Work

The results motivate future work to close the gap with the kernel-mode approaches described in this paper. First, the software interrupt kernel-mode approach suffers from the problems described by [24]. Second, even the thread-based kernel-mode approach has limited applicability beyond simple caching. While it's possible to embed application-specific code in the kernel, the approach is awkward and leads to a paradigm where the benefits of address spaces are lost.

Future work will focus on support for user-mode servers to achieve kernel-mode performance without implementing application-specific code in the kernel. In particular, the performance advantages of a kernel-mode approach might be amortized by batching multiple layer 7 requests before indicating them to a user-mode server. Likewise, response processing might also be aggregated over multiple responses before indicating those responses back to the kernel. Future work will explore batching techniques to amortize user-mode overhead using a kernel-mode request/response engine and cache.

References

[1] Akamai Technologies, Inc. Akamai freeflow service. `http://www.akamai.com/service/network.html`.

[2] TimesTen Performance Software. Timesten front-tier. `http://www.timesten.com`.

[3] Jim Challenger, Arun Iyengar, and Paul Danzig. A scalable system for consistently caching dynamic Web data. In *Proceedings of IEEE INFOCOM'99*, March 1999.

[4] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 16–31, December 1999.

[5] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, June 1999.

[6] The Apache Group. Apache http server project. `http://www.apache.org`.

[7] Zeus Technology Ltd. Zeus web server. `http://www.zeus.com`.

[8] Microsoft Corporation. Internet information services features. `http://www.microsoft.com/windows2000/guide/server/features/web.asp`.

[9] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. High performance Web servers on Windows NT: Design and performance. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 149–149, Berkeley, CA, USA, August 1997. USENIX.

[10] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS '98)*, Madison, WI, June 1998.

[11] Jeffrey C. Mogul. Operating systems support for busy internet servers. Technical Report Technical Note TN-49, Digital Western Research Laboratory, Palo Alto, CA., May 1995.

[12] Alacritech. `http://www.alacritech.com`.

[13] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Usenix Annual Technical Conference*, pages 253–265, 1999.

[14] Niels Provos and Chuck Lever. Scalable network I/O in Linux. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference (FREENIX-00)*, pages 109–120, Berkeley, CA, June 18–23 2000. USENIX Association.

[15] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-lite: A unified I/O buffering and caching system. In *Operating Systems Design and Implementation (OSDI '99)*, pages 15–28, 1999.

[16] Microsoft Corporation. Installation and performance tuning of microsoft scalable web cache (swc 2.0). `http://www.microsoft.com/technet/iis/swc2.asp`.

[17] Arjan van de Ven. kHTTPd Linux http accelerator. `http://www.fenrus.demon.nl`.

[18] Ingo Molnar. Answers from planet TUX: Ingo Molnar responds. `http://slash-dot.org/articles/00/07/20/1440204.shtml`.

[19] Jochen Liedtke, Vsevolod Panteleenko, Trent Jaeger, and Nayeem Islam. High-performance caching with the lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 131–142, Berkeley, USA, June 15–19 1998. USENIX Association.

[20] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Héctor M. Briceño, Russel Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. Technical Report CMU-CS-00-117, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA., mar 2000.

[21] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 52–65, New York, October 5–8 1997. ACM Press.

[22] Elbert C Hu, Philippe A Joubert, Robert B King, Jason LaVoie, and John M Tracey. Adaptive Fast Path Architecture. *to appear in IBM Journal of Research and Development*, April 2001.

[23] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Differentiation in overloaded web servers. In *Proceedings of the USENIX 2001 Annual Technical Conference*, June 2001.

[24] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[25] David Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2001.

[26] The Standard Performance Evaluation Corporation. Specweb96 benchmark. `http://www.spec.org/osg/web96`.

[27] Mindcraft Inc. Webstone - the benchmark for web servers. `http://www.mindcraft.com/webstone`.

[28] Mindcraft Inc. Open benchmark: Windows NT Server 4.0 and Linux. `http://www.mindcraft.com/whitepapers/openbench1.html`.

[29] SGI. Accelerating apache. `http://www.oss.sgi.com/apache`.