Performance Issues in WWW Servers

Erich Nahum, Tsipora Barzilai, and Dilip Kandlur
IBM T.J. Watson Research Center
Hawthorne, NY 10532
{nahum,tsipora,kandlur}@watson.ibm.com

April 29, 1998

Abstract

This paper evaluates performance issues in WWW servers on UNIX-style platforms. While other work has focused on reducing the use of kernel primitives, we explore ways in which the operating system and the network protocol stack can improve support for high-performance WWW servers. We examine two proposed socket functions, `acceptex()` and `transmitfile()`, comparing `transmitfile()`'s effectiveness with an `mmap()`/`writev()` combination. We show how `transmitfile()` provides the necessary semantic support to eliminate copies and checksums in the kernel, and quantify the utility of the function's header and close options. We also present mechanisms to reduce the number of packets exchanged in an HTTP transaction, both increasing server performance and reducing network utilization. We explore these issues with a high-performance WWW server, using IBM AIX machines connected by both Ethernet and ATM, and driven by the WebStone and SpecWeb WWW server benchmarks. Our combination of mechanisms improve server throughput by up to 50 percent, reduce server CPU utilization up to 67 percent, and eliminate 33 percent of the packets in an HTTP exchange, without compromising interoperability.

1 Introduction

The phenomenal growth of the World-Wide Web, in both the volume of information on it and the numbers of users desiring access to it, is placing dramatic increases in the performance requirements for large scale information servers. WWW server performance is thus a central issue in providing ubiquitous, reliable, and efficient information access. This paper evaluates issues in WWW server performance on UNIX-style platforms. While other work has focused on reducing the use of kernel primitives, we explore ways in which the operating system and the network protocol stack can improve support for high-performance WWW servers. Issues we consider include:

- *new socket functions.* Microsoft has added two new socket functions to NT [17], `acceptex()` and `transmitfile()`, ostensibly for performance reasons. These functions streamline the operations occuring in a typical HTTP transaction, and with suitable options provide the necessary semantics to enable further protocol stack optimizations. How effective are they, and which options are useful? Does `transmitfile()` provide any benefit over the already-available `mmap()` and `writev()` system calls?
- *eliminating copies.* BSD-derived Unix operating systems [26] use different buffering mechanisms in the file system and the networking code, forcing data to be copied when it is moved from one subsystem to another. How well can we approximate a zero-copy integrated I/O architecture [33], while continuing to exploit the benefits of existing file systems?
- *eliminating checksums.* The cost of calculating checksums can be expensive in network protocol processing [4, 10, 14, 24]. What sort of performance impact will eliminating checksums have for WWW servers?
- *reducing packet exchanges.* It is well-known that TCP was not designed for client-server traffic, exchanging more packets that is semantically necessary. How can the packet count be reduced, without violating the TCP protocol specification?

We evaluate these issues using an experimental testbed with several IBM RS/6000 workstations running AIX 4.2.1, connected over both 100 Mbit Ethernet and 155 Mbit ATM. We use the industry-standard WWW workloads SpecWeb96 and WebStone to drive the system with HTTP 1.0 requests, and the AIX kernel profiling tool UTLD [12] to identify bottlenecks. We use Rice University's Flash WWW server, which exploits all currently-known user-level optimizations. We evaluate the utility of the new socket functions (`acceptex()` and `transmitfile()`), implemented as AIX 4.2.1 kernel extensions, in order to quantify their benefit for WWW servers.

Our experience confirms previous work showing that WWW servers spend most of their time in the kernel [1, 18, 37]. described further in the Appendix. The choice of execution model, threads or processes, significantly affects what performance improvements are available to WWW servers [16]. Many optimizations are easier to apply with servers that use a single-process model than those that use a multi-process model.

We build upon previous work by exploring ways in which the operating system and protocol stack can improve support for high-performance WWW servers. We examine the benefits of two proposed socket functions, `acceptex()` and `transmitfile()`, comparing the effectiveness of `transmitfile()` with an `mmap()`/`writev()` combination. We show how `transmitfile()` provides the necessary semantic support for further optimizations, such as eliminating copies and reducing packet exchanges, and quantify the utility of the function's header and close options. We present mechanisms to reduce the number of packets exchanged in an HTTP transaction, both increasing server performance and reducing network utilization.

We find that `acceptex()` provides little benefit for WWW servers, and that using a single-copy `transmitfile()` offers no advantage over `mmap()`/`writev()`. However, a zero copy implementation can improve performance by up to 50 percent for large requests, and eliminating the checksum increases throughput by an additional 10 percent. Our techniques for reducing packet exchanges, without compromising interoperability, eliminate 33 percent of the packets and improve server performance by up to 16 percent for small transacations. Hence, the combination of these techniques effectively benefits the entire range of WWW server workloads.

The rest of this paper is organized as follows: Section 2 provides more background on WWW servers and reviews previous work. Section 3 describes our experimental setup, and Section 4 presents our results in detail. Section 5 presents our conclusions and briefly discusses our plans for future work. We also include an Appendix which summarizes the benefits of our techniques and illustrates where WWW servers spend time in the kernel.

2 Background and Related Work

In this section we provide an overview of a typical HTTP transaction and discuss related work. To gain a better understanding of performance in a WWW server, we outline the steps required to process a typical request (i.e., a static GET). For each request:

1. `accept()` is called to get a new connection.
2. `getsockname()` is called to determine the remote host.
3. `read()` is called on the socket to get the HTTP request.
4. `setsockopt()` is called to disable the Nagle algorithm.
5. `gettimeofday()` is called to determine the time of the request for logging purposes.
6. The request is parsed, identifying the appropriate file to send.
7. `stat()` is called to obtain the file status and size.
8. `open()` is called on the requested file.
9. `read()` is called on the file descriptor to read the file into the server.
10. `write()` is called on the socket to send the HTTP header to the client.
11. `write()` is called on the socket to send the file to the client.

12. `close()` is called to close the file.

13. `close()` is called to shutdown the connection.

14. `write()` is called on the log file descriptor to log the request.

The order of these actions may change slightly, but will have little impact on performance. For example, the socket could be closed before the file is, reducing the latency as seen by an individual client, but the pathlength of instructions to service each request would not change, thus the server throughput would be unaffected.

In general, many WWW server performance optimizations are focused around reducing the frequency or cost of the above operations:

- Microsoft has implemented the `acceptex()` and `transmitfile()` socket functions in Windows NT. `acceptex()` combines the `accept()`, `getsockname()`, and `recv()` system calls (steps 1, 2, and 3 above). `transmitfile()` directs the kernel to send a file identified by a file descriptor using the specified socket, replacing the `read()` and `write()` system calls (steps 9 and 11). `transmitfile()` also includes 2 optional arguments: the header option and the close option. The header argument is used to pass a buffer which will be sent before the file, and is typically used for prepending the HTTP header, eliminating another `write()` system call (step 10). The close option instructs the operating system to shut down the connection after the send is completed, removing the need for the `close()` system call (step 13). `transmitfile()` reduces both the number of system calls and the data movement into and out of user space from the kernel. James Hu *et. al.* evaluate the use of these functions on NT in [17].

- Zeus [19] manages a cache of `mmap()`'ed files, as exposed by James Hu *et. al.* [16]. In the case of a file hit, the `stat()`, `open()`, `read()`, and `close()` system calls (steps 7, 8, 9, and 12) are eliminated. Zeus also uses `writev()`, to combine the two `write()` calls (steps 10 and 11), and further reduce the use of system calls.

- Yiming Hu *et. al.* [18] use kernel profiling to identify where time is spent in the operating system running WWW servers. They use this information to propose and evaluate several caching techniques to improve performance of the Apache WWW server. They use a URI cache, reducing the cost of URI parsing (step 6), and cache file state, eliminating a `stat()` call (step 7). File contents are also cached for files under 100 KB, eliminating the `open()`, `read()`, and `close()` system calls (steps 8, 9, and 12). Files larger than 100 KB are read in via `mmap()` rather than `read()` (step 9). In this case, while the number of system calls are the same, the data is not copied, reducing the number of times the data is touched.

A central issue is that the process model used by the server can affect what sort of optimizations are possible. For example, since `mmap()` maps a file into a single process' address space, `mmap()`'ed files can not be dynamically shared across multiple processes. Apache, which uses a process model, cannot take full advantage of a cache of `mmap()`'ed files the way a multithreaded server such as Zeus does. Hu *et. al.* avoid this problem by keeping a copy of recently-accessed files in the server's user-space memory, but this means those files are effectively double-buffered, increasing memory requirements.

In contrast to the research above, which attempts to reduce the use of operating system services, other work has centered around improving the OS performance directly:

- Kaashoek *et. al.* [22, 23] advocate a customized operating system tailored specifically for servers. They demonstrate a prototype HTTP server OS that they claim performs an order of magnitude better than a conventional OS. Mechanisms they utilize include a unified disk buffer cache and network retransmission buffer, an event-driven (rather than process-driven) execution model, compiler assisted integrated layer processing, and storing precomputed checksums of WWW documents to eliminate the need for fast-path checksum calculation.

- Druschel, Pai, and Zwaenopol [13] take issue with the idea that extensible micro-kernel operating systems are necessary for good performance, claiming that many of the performance techniques used in micro-kernels are equally suitable for monolithic kernel structures. They present an implementation to support this viewpoint, an integrated I/O system for UNIX called I/O Lite [33]. Their system provides a new I/O interface, and they demonstrate performance improvements from 10 to 100 percent on a number of benchmarks.

Several other analyses of WWW server performance have also been performed [1, 27, 28, 37]. Several of these identify performance issues that have been addressed in the operating system that we employ, AIX [11], such as separately managing TCP connections in the TIME_WAIT state or using hash tables for PCB lookup.

We build upon previous work by exploring ways in which the operating system and the network protocol stack can improve support for high-performance WWW servers. Like Druschel, Pai, and Zwaenopol, we believe a general-purpose operating system is necessary in order to support a wide variety of services over HTTP, such as dynamic content or CGI. However, we take a more conservative stance than they do as to how much the API can change. Given the difficulty in getting modifications to the API adopted, we believe the fewer changes the better, and show how a single API change can achieve much of the benefits of I/O Lite. We provide an in-depth analysis of the `acceptex()` and `transmitfile()` functions on a UNIX platform, and show how these functions enable further optimizations. We quantify the performance benefit of several optimizations in the context of a WWW workload, such as eliminating data copies and checksumming, and show how to reduce the number of packets in an HTTP exchange, further improving performance as well as reducing network utilization.

3 Experimental Setup and Testbed

In this Section we describe our experimental testbed, including the hardware we use, the operating system and extensions, the WWW server software, and the WWW client workload.

3.1 Hardware

Our testbed consists of 3 IBM 43P AIX workstations, connected by both 100 Mbit Ethernet and 155 Mbit ATM. Each machine has a PowerPC 604 processor running at 133 MHz, and 128 MB of RAM. The 604 has 16 KB on-chip 4-way associative instruction and data caches. The 43P's used also have a unified 512 KB direct-mapped secondary cache. Two machines act as clients, the third is the server.

3.2 WWW Client Workload

We use both the WebStone [36] and SpecWeb [8] benchmarks to evaluate WWW server performance. Considerable debate [3, 5] has occurred over how accurate these benchmarks are as predictors of 'real-world' performance. WebStone defaults to submitting a distribution of requests which are spread across only 5 files (albeit of different sizes), and thus is not considered very realistic. The distribution of requests offered by SpecWeb is much larger, across dozens of files of varying sizes, but SpecWeb also has its detractors.

Figure 1 shows the cumulative distribution of file sizes requested by the SpecWeb and WebStone benchmarks. For comparison purposes, a distribution made from the logs of the Kasparov-Deep Blue Chess site is included. Looking at the distributions, we agree that SpecWeb is not wholly representative, but believe that it is more realistic than WebStone. We thus use the two benchmarks in different ways. WebStone allows easier configuration, so we use it to load the server with many concurrent requests for the *same file*. We then vary the size of that file, comparing how different servers behave with a request for a file of a particular size. We can then state 'server A has better performance than server B for files of size K when K is in the VM cache,' without saying that scenario is representative of real workloads. We use SpecWeb as a *system benchmark*, to load the server with concurrent requests for a *range* of files, offering an indication of how useful some performance technique will be in real WWW server environments.

3.3 Operating System and Extensions

We use AIX version 4.2.1 on our machines, with the addition of 3 kernel extensions that we developed:

- *Acceptex()*. This implements the functionality of `acceptex()` system call, as described earlier in section 2. It combines the `accept()`, `getsockname()`, and `recv()` system calls.

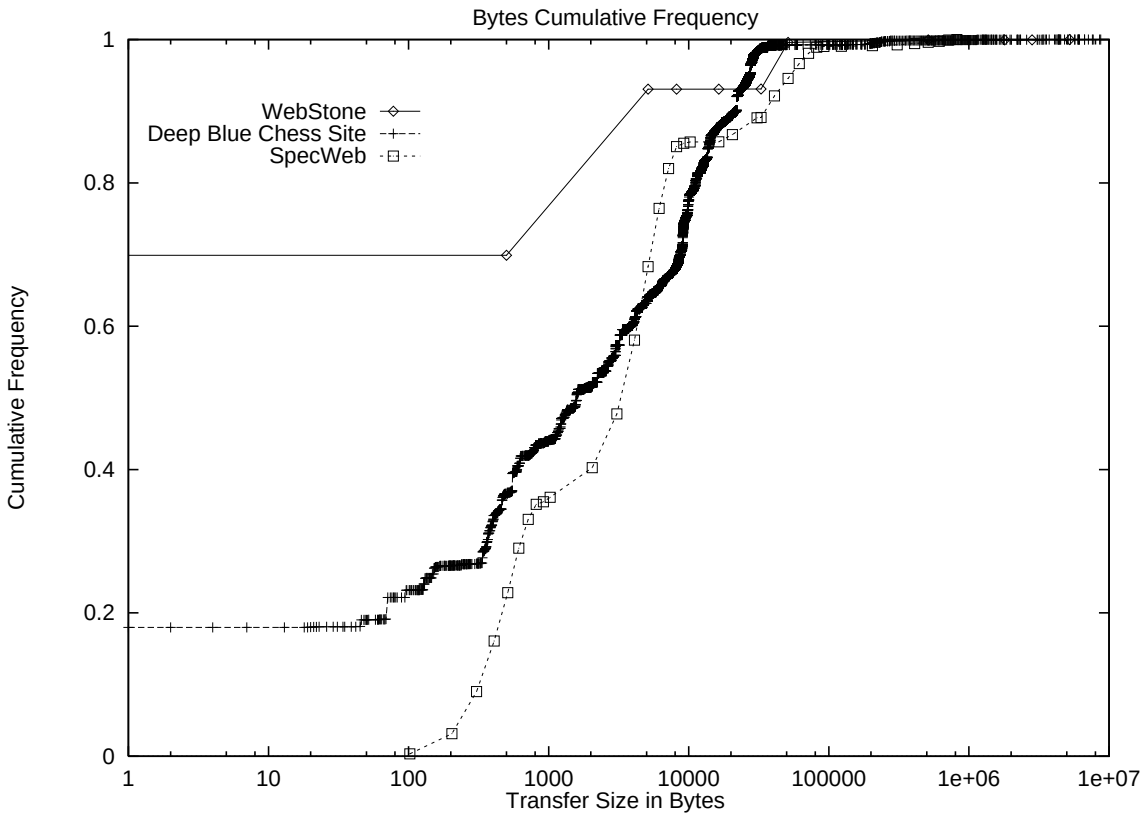

Figure 1: File Size Cumulative Distributions

- *Transmitfile().* This implements the functionality of the `transmitfile()` system call, also described in section 2. This implementation of `transmitfile()` allocates an mbuf in the kernel, reads the file into it using the `fp_read()` internal kernel function, and calls the socket's `pru_usrreq()` function with the SEND option set. Thus, a single copy of the file data is incurred for each HTTP request, even if the file is in the VM cache. Our `transmitfile()` implementation also supports the header buffer and socket close options.

- *Transmitfile() with Mbuf Caching.* This is the same as the `transmitfile()` kernel extension above, except that we have added a caching mechanism within the kernel that is separate from the VM system. On each `transmitfile()` the kernel checks to see if the file is present in the mbuf cache, and if so, re-sends the mbufs rather than calling `fp_read()`. If the file is not present, it is added to the mbuf cache, which is managed with a least-recently used (LRU) policy. Since AIX does not posess an integrated I/O system, `transmitfile()` requires a copy of the data when moving a file from the file system to the network protocol stack. We attempt to estimate the performance benefit of an integrated I/O system by using this mbuf cache. If the cache has a reasonably good hit rate, most files will be served from the mbuf cache, thus providing a close approximation of a zero-copy implementation.

We also made several changes to the TCP/IP protocol stack to reduce the number of packets used in an HTTP exchange. We discuss those implementation details in Section 4.

3.4 WWW Server Software

For our experiments, we use the Flash WWW server developed by Pai *et. al.*[33] at Rice University, as part of their work on I/O Lite. Originally derived from the tiny/turbo/throttling HTTP daemon (thttpd) [25], Flash is a single-threaded event-driven server that uses the `select()` system call and asynchronous I/O. To our knowledge, Flash exploits *all* optimizations that are available to a user-space web server without modifying the operating system. It caches files in user space with `mmap()`, caches `stat()` information, caches URI lookups, and exploits `writev()`. It is reported to be competitive with Zeus [32], which is well-known for its performance [16]. Section 4.1 compares Flash's performance to several other WWW servers.

In order to illustrate and understand the performance benefits of various features, such as the proposed functions, eliminating checksum computation, and copying, we modify the server incrementally and measure the difference in performance. By observing how throughput changes as features are added, we can quantify what the utility of that feature is. The incremental steps we take are:

1. *Baseline.* This is the baseline WWW server, without any optimizations.

2. *Baseline with AcceptExtended (AE).* This adds support for `acceptex()` system call to the server.

3. *Baseline with AE and Transmitfile (TF).* This takes the server in step 2 and adds support for the single-copy `transmitfile()` system call, but does not use any optional arguments.

4. *Baseline with AE, TF, and header option.* This takes the server in step 3 but uses the header argument to `transmitfile()` to pass the HTTP headers, rather than a separate `write()` call.

5. *Baseline with AE, MBuf Caching TF, and header option.* This uses the same server from step 4 but uses the `transmitfile()` implementation that caches mbufs in the kernel, as described above in Section 3.3.

6. *Baseline with AE, MBuf Caching TF, header and close options.* This uses the same environment from step 5, but extends the server to use the close option to `transmitfile()` to close the socket, rather than use a separate `close()` call.

7. *Baseline with AE, MBuf Caching TF, header and close options, and reduced packet exchanges.* The same setup as from step 6, but here the TCP/IP stack has been modified to reduce the packet count, as described in more detail in Section 4.6.

4 Results

In this Section we present our results, showing how various optimizations benefit requests for different file sizes.

File Size (in bytes)	APACHE BASE	APACHE CACHE	ICS BASE	ICS CACHE	FLASH
1024	212.73	282.80	258.65	421.78	769.22
2048	205.05	271.48	241.27	383.02	713.48
4096	182.22	241.08	216.67	342.68	597.53
16384	128.87	174.22	135.97	216.67	334.87
65536	58.15	83.30	53.58	94.83	120.77
262144	17.75	26.63	15.53	30.58	34.52
1048576	4.38	7.32	3.85	8.03	8.57
4194304	*.**	1.82	0.93	1.88	1.90

Table 1: Throughput in Operations/sec

4.1 Baseline Analysis

Here we compare Flash's performance to that of several other WWW servers, in order to illustrate that Flash provides an appropriate platform for our experiments. Table 1 shows the HTTP throughput for several WWW servers: Apache, Apache-Cache, ICS, ICS-Cache, and Flash.

Apache is the freely available WWW server, version 1.2.4, found at `www.apache.org`. Apache is reported to have the largest market share of all WWW servers, with estimates that roughly 50 percent of web sites on the Internet use it [30]. Apache is a process-based server, forking several processes which serially accept new connections, and has very few optimizations.

Apache-Cache is a version of Apache 1.2.4 adapted by Yiming Hu *et. al.* [18] at the University of Rhode Island. As discussed in Section 2, it includes several performance enhancements, including caching URI lookups, caching file state, caching certain string manipulations, caching of files less than 100 KB in user space, and using `mmap()` for files larger than 100 KB. As can be seen from Table 1, this server is up to 67 percent faster than the base Apache. However, it maintains Apache's process architecture.

ICS is IBM's Internet Connection Server (ICS), also known as Lotus Domino Go Webserver (LGD), version 4.6. Derived from the CERN httpd, it uses a single process with multiple threads. ICS allows static caching of files, but they must be explicitly configured by the server administrator.

ICS-Cache is a ICSconfiguration where files used by the WebStone benchmark are cached. It provides an upper-bound on the performance ICScan provide.

Flash was described in Section 3.4. As can be seen in Table 1, Flash provides the highest throughput across a range of file sizes, particularly the small files that are most frequently requested from WWW servers. (We were unable to get Apache to complete the 4 MB WebStone test, as indicated by '*' in the appropriate row and column in Table 1.) Since Flash provides the best performance, we use it to present our results for the rest of this Section, to determine whether the proposed mechanisms can benefit a highly optimized WWW server.

4.2 Using the AcceptEx and Transmitfile Socket Functions

We modified Flash in steps to use the `acceptex()` and `transmitfile()` functions, to see how they affected performance. `acceptex()` replaced calls to `accept()` and `read()`, as described in Section 2. Figure 2 (a) shows the throughput for the Flash server with and without the `acceptex()` system call. As can be seen, the system call makes little or no difference.

Adding `transmitfile()` support to Flash involved slightly more work. We removed Flash's `mmap()` cache, and instead called `transmitfile()` on each request, rather than using `writev()` with `mmap()`'ed files. We did, however, retain Flash's mechanism to cache open file descriptors, so that `open()` and `close()` would not necessarily be invoked for each request. Note that caching open file descriptors would not be convenient with a process-based server such as Apache, since file descriptors are not easily shared across processes. To test `transmitfile()` without the header option, we also had to add a separate `write()` call to send the HTTP header.

Figure 2 (b) shows the change in throughput for the Flash server after adding support for `transmitfile()`. Here the

File Size (in bytes)	FLASH	FLASH AE	Diff (%)
1024	769.22	762.57	*-0.86*
2048	713.48	705.63	*-1.10*
4096	597.53	593.22	*-0.72*
8192	451.93	453.70	*0.39*
16384	334.87	328.03	*-2.04*
65536	120.77	120.03	*-0.61*
262144	34.52	34.18	*-0.98*
1048576	8.57	8.42	*-1.75*
4194304	1.90	1.88	*-1.05*

(a) Using the `acceptex()` API

File Size (in bytes)	FLASH AE	FLASH AETF	Diff (%)
1024	762.57	680.17	*-10.81*
2048	705.63	620.38	*-12.08*
4096	593.22	529.12	*-10.81*
8192	453.70	426.53	*-5.99*
16384	328.03	300.73	*-8.32*
65536	120.03	108.28	*-9.79*
262144	34.18	32.03	*-6.29*
1048576	8.42	8.12	*-3.56*
4194304	1.88	1.92	*2.13*

(b) Using the `transmitfile()` API, no header option

Figure 2: Change in HTTP ops/sec using proposed API's

File Size (in bytes)	FLASH AETF NO HEADER	FLASH AETF WITH HEADER	Diff (%)
1024	680.17	713.35	*4.88*
2048	620.38	660.57	*6.48*
4096	529.12	536.65	*1.42*
8192	426.53	437.45	*2.56*
16384	300.73	311.53	*3.59*
65536	108.28	108.32	*0.04*
262144	32.03	32.10	*0.22*
1048576	8.12	8.02	*-1.23*
4194304	1.92	1.97	*2.60*

(a) Using the `transmitfile()` API, with header option

File Size (in bytes)	FLASH AE	FLASH AETF WITH HEADER	Diff (%)
1024	762.57	713.35	*-6.45*
2048	705.63	660.57	*-6.39*
4096	593.22	536.65	*-9.54*
8192	453.70	437.45	*-3.58*
16384	328.03	311.53	*-5.03*
65536	120.03	108.32	*-9.76*
262144	34.18	32.10	*-6.09*
1048576	8.42	8.02	*-4.75*
4194304	1.88	1.97	*4.79*

(b) Using the `transmitfile()` API

Figure 3: Change in HTTP ops/sec using proposed API's

`transmitfile()` implementation is the single-copy version, and the header option is not used. As we can see, performance degrades by up to 12 percent, we believe both due to the extra `write()` call and because of interactions with the Nagle algorithm, which we discuss further in Section 4.7.

Figure 3 (a) shows the difference in performance using `transmitfile()` with and without the header option. When the header option is used, there is no extra `write()` call, and throughput improves somewhat. Figure 3 (b) shows the final change with and without `transmitfile()` with the header option. In this case, `transmitfile()` does not improve performance, and even seems to reduce throughput slightly, for reasons we are still investigating. However, the conclusion can be drawn that a single-copy `transmitfile()` seems to offer no performance benefit over a `mmap()/writev()` combination.

It is clear that when using `transmitfile()`, the header option should be used for the same reasons high-performance servers use `writev()` rather than multiple calls to `write()`. For simplicity, readers should assume all further results reported with `transmitfile()` use the header option.

4.3 Eliminating the Copy

Figure 4 (a) shows the throughput for the Flash server using the single-copy and mbuf caching versions of the `transmitfile()` system call. Here we see substantial performance improvements of up to 50 percent, with caching the mbufs making progressively larger differences with files up to 64 KB. For files larger than 64 KB, the network appears to be saturated, even using the single-copy `transmitfile()`. However, caching the mbufs still reduces the amount of work required by the server for large files as well. Figure 4 (b) shows the reduction in CPU utilization, as measured by `iostat`. In addition to the increase in performance, we see that there is a significant reduction in utilization as well. Multiple network interfaces, or faster

File Size (in bytes)	FLASH AETF	FLASH AETF MBUF	Diff (%)
1024	713.35	727.25	*1.95*
2048	660.57	684.63	*3.64*
4096	536.65	610.90	*13.84*
8192	437.45	515.38	*17.81*
16384	311.53	399.47	*28.23*
65536	108.32	166.63	*53.83*
262144	32.10	42.48	*32.34*
1048576	8.02	10.37	*29.30*
4194304	1.97	2.38	*20.81*

(a) Throughput in HTTP operations/sec

File Size (in bytes)	FLASH AETF	FLASH AETF MBUF	Diff (%)
1024	100.00	100.00	*0.00*
2048	100.00	100.00	*0.00*
4096	100.00	100.00	*0.00*
8192	100.00	100.00	*0.00*
16384	100.00	100.00	*0.00*
65536	100.00	45.00	*55.00*
262144	100.00	10.00	*90.00*
1048576	100.00	5.00	*95.00*
4194304	100.00	2.00	*98.00*

(b) CPU Utilization in percent

Figure 4: Impact of using MBuf Caching `transmitfile()` API

File Size (in bytes)	FLASH ATM AETF MBUF CLOSE	FLASH ATM AETF MBUF CLOSE CKSUM	Diff (%)
1024	711.53	726.02	*2.04*
2048	713.12	717.82	*0.66*
4096	682.70	696.73	*2.06*
8192	609.58	637.80	*4.63*
16384	502.35	547.93	*9.07*
65536	234.27	253.77	*8.32*
262144	64.00	63.75	*-0.39*
1048576	15.73	15.82	*0.57*
4194304	3.58	3.62	*1.12*

(a) Throughput in HTTP operations/sec

File Size (in bytes)	FLASH ATM AETF MBUF CLOSE	FLASH ATM AETF MBUF CLOSE CKSUM	Diff (%)
1024	100.00	100.00	*0.00*
2048	100.00	100.00	*0.00*
4096	100.00	100.00	*0.00*
8192	100.00	100.00	*0.00*
16384	100.00	100.00	*0.00*
65536	80.00	70.00	*12.50*
262144	25.00	15.00	*40.00*
1048576	10.00	4.00	*60.00*
4194304	3.00	1.00	*66.67*

(b) CPU Utilization in percent

Figure 5: Impact of Eliminating Checksums

interfaces such as gigabit Ethernet, would provide more bandwidth, which in turn would allow greater throughput for larger files.

4.4 Offloading the Checksum to the Adaptor

Using the ATM adaptors in our environment, AIX allows disabling the Internet checksum, which provides us a close approximation of how performance would change if the checksum was offloaded to the adaptor. Figure 5 (a) shows the change in throughput for the Flash server when the host CPU does not perform the checksum. We observe increases in performance of up to 9 percent, but again are limited by network bandwidth for files larger than 64 KB. Figure 5 (b) shows the reduction in CPU utilization, as measured by `iostat`. We see that utilization is reduced by up to 67 percent. Again, increases in bandwidth would allow offloading the checksum to improve throughput even further.

4.5 The Close Option to TransmitFile

Recall from Section 2 that the close option to `transmitfile()` shuts down the connection after sending the file. In our initial implementation, a close call was added in the `transmitfile()` implementation in the socket layer, but only a small performance win was observed.

1. Client: SYN 0:0(0)
2. Server: SYN 0:0(0) ACK 1
3. Client: ACK 1
4. Client: 1:61(60) ACK 1
5. **Server: 1:1159(1158) ACK 61**
6. **Server: FIN 1159:1159(0) ACK 61**
7. Client: ACK 1160
8. Client: FIN 61:61(0) ACK 1160
9. Server: ACK 62

(a) Original

1. Client: SYN 0:0(0)
2. Server: SYN 0:0(0) ACK 1
3. Client: ACK 1
4. Client: 1:61(60) ACK 1
5. **Server: FIN 1:1159(1158) ACK 61**
6. Client: ACK 1160
7. Client: FIN 61:61(0) ACK 1160
8. Server: ACK 62

(b) Piggybacking the FIN

Figure 6: TCP Packet Exchanges in HTTP

File Size (in bytes)	FLASH AETF MBUF	FLASH AETF MBUF CLOSE	Diff (%)
1024	727.25	777.87	*6.96*
2048	684.63	729.78	*6.59*
4096	610.90	645.17	*5.61*
8192	515.38	527.45	*2.34*
16384	399.47	402.92	*0.86*
65536	166.63	166.48	*-0.09*
262144	42.48	42.67	*0.45*
1048576	10.37	10.38	*0.10*
4194304	2.38	2.28	*-4.20*

(a) Using 100 Mbit Ethernet

File Size (in bytes)	FLASH ATM AETF MBUF	FLASH ATM AETF MBUF CLOSE	Diff (%)
1024	669.18	711.53	*6.33*
2048	656.77	713.12	*8.58*
4096	634.40	682.70	*7.61*
8192	584.43	609.58	*4.30*
16384	481.68	502.35	*4.29*
65536	233.57	234.27	*0.30*
262144	63.95	64.00	*0.08*
1048576	15.70	15.73	*0.19*
4194304	3.65	3.58	*-1.92*

(b) Using 155 Mbit ATM

Figure 7: Impact of Close Option

Figure 6 (a) shows the sequence of TCP packets exchanged in a typical HTTP transaction requesting a 1 KB file, taken from `tcpdump` [21]. One can see that the sixth packet in the exchange carries only a FIN bit, signaling that the server is done sending data. This information, a single bit, can easily be carried by the fifth packet if the server's TCP knows that the connection is finished *before* it sends the last packet. However, BSD-derived TCP implementations have historically not included a semantic operation for both queuing data and shutting down the connection. The close option to `transmitfile()` provides half of the required mechanism. The other half must be added to the TCP layer, and be invoked through the BSD in-kernel socket interface. The socket layer calls lower-layer protocols through the protocol-independent `pr_usrreq` function pointer, which in turn invokes `tcp_usrreq`. `tcp_usrreq` supports the `PRU_SEND` and `PRU_DISCONNECT` operations, but not the 'queue-and-close' functionality described above. However, it was a simple matter to add an additional option, `PRU_SEND_DISCONNECT`, which appends data to the socket buffer, sets the TCP connection state to TCP_FIN_WAIT_1, and then calls `tcp_output`. Our `transmitfile()` implementation thus calls `pr_usrreq` with `PRU_SEND_DISCONNECT` when sufficient send buffer space is available. Figure 6 (b) shows the sequence of TCP packets after this change is made. It can be seen that the server piggybacks the FIN bit on the last data segment.

At first glance, this mechanism might seem to violate the optimize-the-common-case rule of header prediction [20], since the data is processed along the slow path along with the FIN. However, the costs in taking the slow path are more than made up by the savings incurred from not processing an additional packet, including interrupt overhead, copying the packet from the network device, and calculating TCP and IP header checksums.

1. Client: SYN 0:0(0)
2. Server: SYN 0:0(0) ACK 1
3. **Client: ACK 1**
4. **Client: 1:61(60) ACK 1**
5. Server: FIN 1:1159(1158) ACK 61
6. Client: FIN 61:61(0) ACK 1160
7. Server: ACK 62

(a) Delaying ACK of FIN

1. Client: SYN 0:0(0)
2. Server: SYN 0:0(0) ACK 1
3. **Client: 1:61(60) ACK 1**
4. Server: FIN 1:1159(1158) ACK 61
5. Client: FIN 61:61(0) ACK 1160
6. Server: ACK 62

(b) Delaying ACK of SYN-ACK

Figure 8: Reducing HTTP TCP Packet Exchanges Further

Figure 7 (a) shows the subsequent change in throughput between experiments with and without the close option on our 100 Mbit Ethernet testbed. Figure 7 (b) presents the same information for the 155 Mbit ATM environment. We observe that in requests for small files, up to a 7 percent increase in HTTP throughput is achieved on Ethernet, and up to 8 percent on ATM.

Note from Figure 7 that there is no change for large files, because in these cases the FIN is already piggybacked. For larger transfers, data queues in the send buffer, waiting for acknowledgments to return and open the flow control and congestion control windows. While this data is waiting to be sent, the connection is closed by the WWW server, allowing `tcp_output` to recognize the final segment and send it with the FIN bit set.

Thus, the close option only helps in situations where the sender is not limited by the congestion window, which is a function of the segment size and the number of ACKs received. Using Ethernet, which has a 1500 byte MTU, this is up to 8 KB transfers. On ATM, which in our testbed has a 9 KB MTU, the close option benefits transfers of up to 16 KB, as seen in Figure 7 (b). While this optimization only affects transfers for small files, recall from Figure 1 that the average file transfer is under 4 KB, and that 85 percent of transfers are for files under 16 KB. Thus, many transactions will benefit. Finally, reducing the packet count not only lessens the load on the server, but also improves network utilization, helping reduce congestion on the Internet.

4.6 Reducing Packet Exchanges Further

Examining the packet exchanges in Figure 6, it can be seen that further reductions in packets are possible, given the redundant information being communicated. For example, in Figure 6 (b), it can be seen that packet 7, which sends the client's FIN, contains all the ACK information in packet 6, which ACK's the server's FIN. Similarly, all the information in packet 3, which ACK's the server's SYN-ACK, is available in packet 4, which contains the client's HTTP GET request. This is because acknowledgments in TCP are cumulative, and because TCP requires every packet to contain an ACK (except for the initial SYN packet). Eliminating these redundant packets improves server performance, since fewer packets require processing.

An important question is whether eliminating these packets violates the TCP protocol specification [34, 6, 7, 35]. Our understanding of the protocol is that it does not, and that these packets are artifacts of the BSD implementation. In these cases, acknowledgments are delayed, not eliminated, which is consistent with TCP's delayed ACK strategy, and in practice the clients' packets will be sent immediately. In the case of the SYN-ACK the GET request will quickly follow, and in the case of the FIN the client will shut down its side of the connection in response and send its own FIN.

Figure 8 (a) shows the TCP packet exchange after removing the ACK of server FIN when the state is `TCPS_ESTABLISHED` (i.e., the client). This was enabled by removing a line in `tcp_input` which forces `TF_ACKNOW` to be set when a FIN is received. Instead, the normal 200 ms. timeout is used, so that the ACK will be piggybacked on the next outgoing packet, which in this case is the client's FIN. Figure 9 (a) shows the change in performance, between experiments with and without the delayed ACK of the FIN, with increases in throughput of up to 4.5 percent.

Figure 8 (b) shows the exchange after removing the ACK of the server's SYN-ACK. Again, this was achieved by removing a line in `tcp_input` which sets `TF_ACKNOW` when a SYN-ACK is received, and by preventing `needoutput` from being set in that case. Figure 9 (b) presents the subsequent change in throughput between experiments with and without the delayed ACK of the SYN-ACK. As can be seen, removing the unnecessary ACK results in a 4 percent additional increase in performance.

We have not yet evaluated how delaying these ACKs might affect the performance of other applications that use TCP, such

File Size (in bytes)	FLASH AETF MBUF CLOSE	FLASH AETF MBUF CLOSE DELFIN	Diff (%)
1024	777.87	811.12	*4.27*
2048	729.78	763.13	*4.57*
4096	645.17	673.55	*4.40*
8192	527.45	550.37	*4.35*
16384	402.92	413.12	*2.53*
65536	166.48	166.85	*0.22*
262144	42.67	42.60	*-0.16*
1048576	10.38	10.42	*0.39*
4194304	2.28	2.27	*-0.44*

(a) Delaying ACK of the FIN

File Size (in bytes)	FLASH AETF MBUF CLOSE DELFIN	FLASH AETF MBUF CLOSE DELFIN DELSYN	Diff (%)
1024	811.12	845.40	*4.23*
2048	763.13	793.92	*4.03*
4096	673.55	691.35	*2.64*
8192	550.37	558.37	*1.45*
16384	413.12	418.00	*1.18*
65536	166.85	168.05	*0.72*
262144	42.60	42.58	*-0.05*
1048576	10.42	10.27	*-1.44*
4194304	2.27	2.32	*2.20*

(b) Delaying ACK of the SYN-ACK

Figure 9: Throughput in Operations/sec

as SMTP or FTP. Since these applications wait for the server to respond before sending data, for example, delaying the ACK of the SYN-ACK might increase their delay by up to 200 ms. We do not believe this would be a major problem, especially given the predominance of HTTP traffic. However, if it were a concern, the delayed ACK feature could easily be made runtime configurable, so that it would be specifically enabled only on machines devoted as HTTP servers.

4.7 Packetization Issues and the Nagle Algorithm

Based on our experience, we believe it is important to reiterate an issue concerning WWW servers and the Nagle algorithm [29]. Researchers have presented evidence [15, 31] that the Nagle algorithm should be disabled, in order to reduce the latency as observed by the client and to protect against unforeseen interactions between TCP and HTTP with persistent connections.

Nagle restricts sending of packets when the segment available to send is less than a full MTU size, in order reduce transmission of small packets and thus improve network utilization. The algorithm works as follows: if all outstanding data has been acknowledged, any segment is sent immediately. If there is unacknowledged data, the segment is only transmitted if it is a full MTU size. Otherwise, it is queued in the hope that more data will soon be delivered to the TCP layer and then a full MTU can be sent. However, if the 200 ms. timer expires before any more data is presented to the TCP layer, the segment is then sent. Nagle is not a major issue with HTTP 1.0 traffic, since segments less than an MTU size will be pushed out with the FIN when the connection is closed [15]. However, when using persistent connections, Nagle can unnecessarily delay the last segment of a response, if it is less than a full MTU. That segment will wait for the next 200 ms. timeout, since the connection is not necessarily closed immediately, as would occur in HTTP 1.0.

However, if Nagle is disabled, care should be taken with how data is queued into the socket layer, otherwise a packet will be sent on each `write()` call, needlessly producing extra work for the server and extra packets on the network. WWW servers avoid this problem either by using `writev()` (e.g., Flash, JAWS, and Zeus), or by using their own buffering scheme that aggregates data in user space and then calling `write()` (e.g., Apache).

While the Nagle algorithm does not affect the packet count with HTTP 1.0 traffic, disabling it can lower server performance slightly by adding a `setsockopt()` call on the fast path for servicing an HTTP request. Flash does not disable Nagle, and we found that throughput serving 1 KB files fell about 2 percent after adding a `setsockopt()` call on each new connection to disable Nagle. This cost could be removed in either of two fashions: First, an extra option could be added to `transmitfile()` to disable Nagle, which would avoid an extra system call. Second, the cost could be taken out of the fast path by using `setsockopt()` on the parent listen socket and allowing the option to be inherited by subsequently accepted sockets. While some socket options are inherited from the parent listen socket, current versions of BSD, including AIX, do not inherit the Nagle setting. However, this could easily be changed.

File Size (in bytes)	FLASH	FLASH AETF MBUF CLOSE DELFIN DELSYN	Diff (%)
1024	769.22	845.40	*9.90*
2048	713.48	793.92	*11.27*
4096	597.53	691.35	*15.70*
8192	451.93	558.37	*23.55*
16384	334.87	418.00	*24.82*
65536	120.77	168.05	*39.15*
262144	34.52	42.58	*23.35*
1048576	8.57	10.27	*19.84*
4194304	1.90	2.32	*22.11*

(a) Using 100 Mbit Ethernet

File Size (in bytes)	FLASH ATM	FLASH ATM AETF MBUF CLOSE DELFIN DELSYN	Diff (%)
1024	673.53	795.88	*18.17*
2048	645.73	786.72	*21.83*
4096	587.75	768.23	*30.71*
8192	515.27	708.00	*37.40*
16384	405.72	587.85	*44.89*
65536	168.22	254.23	*51.13*
262144	52.33	63.63	*21.59*
1048576	13.37	15.93	*19.15*
4194304	2.85	3.68	*29.12*

(b) Using 155 Mbit ATM

Figure 10: Total Change in HTTP Performance

5 Conclusions

This work has evaluated several issues in improving the performance of WWW servers, examining ways to reduce both per-byte and per-connection costs. Figure 10 presents the total change in HTTP throughput as a result of our optimizations. We see improvements of up to 40 percent using Ethernet and up to 50 percent over ATM. Recall also from Section 4 that these numbers are conservative for transfering large files, in that our testbed is network-limited for those cases, and that we see a corresponding drop in server CPU utilization. We summarize our conclusions as follows:

- *acceptex.* We find little or no increase in performance using this function, on either process-based or thread-based WWW servers. In addition, UTLD profiling shows that servers spend relatively small amount of time in the `accept()`, `getsockname()`, and `read()` system calls.
- *transmitfile.* Using a `transmitfile()` implementation that incurs a single copy provides no advantage over an `mmap()/writev()` combination, even when the header option is exploited. However, an implementation tied to an integrated I/O system, which does not copy data, provides substantially better performance. In our mbuf caching testbed, we observed an increase in throughput of up to 50 percent, and a reduction in CPU utilization of up to 70 percent. For environments using very high-speed interfaces such as gigabit Ethernet, where the machine is not network-limited, we expect an even greater improvement in performance.
- *offloading checksums.* We find that offloading the checksum to the network device can improve WWW server performance by up to 10 percent, and reduce CPU utilization by up to 67 percent. In very high-speed environments, such as gigabit Ethernet, we expect further increases in throughput. In order to accommodate network interfaces that do not support checksum offoad, our mbuf cache mechanism can be enhanced to allow caching of the checksum values in the mbufs.
- *reducing packet exchanges.* We show how the close option to `transmitfile()` provides the semantic support to enable piggybacking the FIN on the last data segment, eliminating one packet and improving throughput by 4 percent on 100 Mbit Ethernet and 8 percent on 155 Mbit ATM. We also show how delaying acknowledgments for the FIN and SYN-ACK packets can eliminate 2 more packets, increasing performance an additional 10 percent. In total, we reduce the packets in a typical HTTP exchange from 9 to 6, reducing network utilization and raising server throughput by 16 percent.

We should point out that, while we have evaluated these optimizations in the context of WWW serving, they have utility for other applications as well. For example, reducing packet exchanges should help other TCP-based applications. `transmitfile()` is a general function and can be used by other network servers, such as NFS, FTP, or SMB. In addition,

it is easier for a developer to use `transmitfile()` that to implement a custom mechanism, such as a cache of `mmap()`'ed files. Finally, a `transmitfile()` cache in the kernel can be used by all applications running on the machine. Thus, if a file is simultaneously served over several different session protocols (e.g., HTTP and SMB), the kernel can benefit from this sharing.

For future work, we plan to evaluate our mechanisms with HTTP 1.1 workloads. Given the current transition to 1.1, it is important to understand performance under this scenario. Since HTTP 1.1 has persistent connections, it is likely that per-connection optimizations such as the close option and delayed ACK mechanisms will most likely be less significant. However, per-byte optimizations should be even more effective than with HTTP 1.0.

Acknowledgments

Special thanks to Vivek Pai for the Flash source, without which this work would have been much more difficult. Thanks also to Yiming Hu for his Apache code, and Chij-Mehn Chang for the original prototypes of `acceptex()` and `transmitfile()`. This work has benefitted from discussions with Herman Dierks, Yiming Hu, Vivek Pai, Dave Marquardt, Rich Neves, and Satya Sharma. Roch Guerin and Arvind Krishna provided useful feedback on earlier drafts of this paper.

References

[1] Jussara M. Almeida, Virgilio Almeida, and David J.Yates. Measuring the behavior of a world-wide web server. In *Seventh IFIP Conference on High Performance Networking (HPN)*, White Plains, NY, April 1997.

[2] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–646, Oct 1997.

[3] Gurav Banga and Peter Druschel. Measuring the capacity of a web server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec 1997.

[4] David Banks and Michael Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[5] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[6] Robert Braden. Requirements for internet hosts – communication layers. In *Network Information Center RFC 1122*, October 1989.

[7] David D. Clark. Window and acknowledgement strategy in TCP. In *Network Information Center RFC 813*, pages 1–22, July 1982.

[8] The Standard Performance Evaluation Corporation. SpecWeb96. http://www.spec.org/osg/web96.

[9] Mark Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, May 1996.

[10] Chris Dalton, Greg Watson, David Banks, Costas Clamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 11(2):36–43, July 1993.

[11] Herman Dierks. Personal communication. IBM AIX Development, Austin TX.

[12] IBM RISC System/6000 Division. Utld 1.2 user's guide. IBM Confidential.

[13] Peter Druschel, Vivek S. Pai, and Willy Zwaenepoel. Extensible kernels are leading OS research astray. In *Sixth Workshop on Hot Topics in Operating Systems*, Cape Code, MA, May 1997.

[14] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, London, England, August 1994.

[15] J. Heidemann. Performance interactions between P-HTTP and TCP implementations. *ACM Computer Communication Review*, 27(2):65–73, April 1997.

[16] James C. Hu, Sumedh Mungee, and Douglas C. Schmidt. Techniques for developing and measuring high-performance web servers over ATM networks. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, San Francisco, CA, Mar 1998.

[17] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference (held as part of GLOBECOM '97)*, Phoenix, AZ, Nov 1997.

[18] Yiming Hu, Ashwini Nanda, and Qing Yang. Measurement, analysis, and performance improvement of the Apache web server. Technical Report 1097-0001, University of Rhode Island Department of Electrical and Computer Engineering, Oct 1997.

[19] Zeus Inc. The Zeus WWW server. http://www.zeus.co.uk.

[20] Van Jacobson. 4BSD header prediction. *ACM Computer Communication Review*, 20(2):13–15, April 1990.

[21] Van Jacobson, Craig Leres, and Steve McCanne. tcpdump. Available at ftp://ftp.ee.lbnl.gov/tcpdump.tar.Z.

[22] M. Frans Kaashoek, Dawson Engler, Gregory R. Ganger, Hector Briceno, Russell Hunt, David Mazieres, Tom Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[23] M. Frans Kaashoek, Dawson Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. In *1996 SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[24] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.

[25] Acme Laboratories. thttpd: The tiny/turbo/throttling HTTP server. Available at http://www.acme.com/software/thttpd.

[26] S. J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[27] Jeffrey C. Mogul. Network behavior of a busy web server and its clients. Technical Report 95/5, Digital Equipment Corporation Western Research Lab, Palo Alto, CA, October 1995.

[28] Jeffrey C. Mogul. Operating systems support for busy Internet servers. In *Proceedings Fifth Workshop on Hot Topics in Operatings Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[29] John Nagle. Congestion control in IP/TCP internetworks. In *Network Information Center RFC 896*, January 1984.

[30] Netcraft. The Netcraft WWW server survey. Available at http://www.netcraft.co.uk/Survey.

[31] Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hokon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Cannes, France, September 1997.

[32] Vivek Pai. Personal communication. Rice University CS Dept., Houston TX.

[33] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. I/O Lite: A copy-free UNIX I/O system. Department of Computer Science, Rice University.

[34] Jon Postel. Transmission Control Protocol. *Network Information Center RFC 793*, pages 1–85, September 1981.

[35] W. Richard Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. In *Network Information Center RFC 2001*, January 1997.

[36] Gene Trent and Mark Sake. WebStone: The first generation in HTTP server benchmarking. http://www.sgi.com/Products/WebFORCE/WebStone.

[37] David J. Yates, Virgilio Almeida, and Jussara M. Almeida. On the interaction between an operating system and web server. Technical Report CS 97-012, Boston University Computer Science Department, Boston, MA, July 1997.

Appendix

In this appendix we include the summarized performance results for our server, and reports from UTLD, illustrating where the time is spent for each server. Table 2 shows how the performance of Flash over 100 Mbit Ethernet changes as various features and optimizations are added. Table 3 lists the same information for Flash over 155 Mbit ATM. The '*' denote cases where the CPU is network-limited, and is thus not fully utilized.

Figure 11 (a) gives the performance breakdown when servicing requests for the same 1 KB file. The "other" category is the sum of all the other less-significant categories, each of which contributed less than 2 percent of the total time. We observe the following:

- For small files, most time is spent in the application itself (27–42 percent of execution time), followed by the Ethernet device driver (the phxentdd entry above, 10–15 percent), `write()` system call (7–10 percent), and I/O interrupts (3–5 percent).

File Size (in bytes)	FLASH	FLASH AE	FLASH AETF	FLASH AETF MBUF	FLASH AETF MBUF CLOSE	FLASH AETF MBUF CLOSE DELFIN	FLASH AETF MBUF CLOSE DELFIN DELSYN
1024	769.22	762.57	713.35	727.25	777.87	811.12	845.40
2048	713.48	705.63	660.57	684.63	729.78	763.13	793.92
4096	597.53	593.22	536.65	610.90	645.17	673.55	691.35
8192	451.93	453.70	437.45	515.38	527.45	550.37	558.37
16384	334.87	328.03	311.53	399.47	402.92	413.12	418.00
65536	120.77	120.03	108.32	166.63	166.48	166.85	168.05
262144	34.52	34.18	32.10	42.48	42.67	42.60	42.58
1048576	8.57	8.42	8.02	10.37	10.38	10.42	10.27
4194304	1.90	1.88	1.97	2.38	2.28	2.27	2.32

Table 2: Throughput in Operations/sec

File Size (in bytes)	FLASH ATM	FLASH ATM AE	FLASH ATM AETF	FLASH ATM AETF MBUF	FLASH ATM AETF MBUF CLOSE	FLASH ATM AETF MBUF CLOSE CKSUM	FLASH ATM AETF MBUF CLOSE DELFIN	FLASH ATM AETF MBUF CLOSE DELFIN CKSUM	FLASH ATM AETF MBUF CLOSE DELFIN DELSYN	FLASH ATM AETF MBUF CLOSE DELFIN DELSYN CKSUM
1024	673.53	666.50	632.95	669.18	711.53	726.02	743.82	761.62	788.08	795.88
2048	645.73	636.27	611.40	656.77	713.12	717.82	746.10	753.98	772.33	786.72
4096	587.75	580.72	547.87	634.40	682.70	696.73	721.65	732.82	752.45	768.23
8192	515.27	505.92	478.90	584.43	609.58	637.80	643.32	665.73	674.73	708.00
16384	405.72	392.97	382.68	481.68	502.35	547.93	533.58	566.17	544.00	587.85
65536	168.22	169.07	153.37	233.57	234.27	253.77	233.85	253.85	233.95	254.23
262144	52.33	52.38	45.37	63.95	64.00	63.75	63.97	63.83	63.83	63.63
1048576	13.37	13.18	11.53	15.70	15.73	15.82	15.78	15.93	15.73	15.93
4194304	2.85	2.85	2.90	3.65	3.58	3.62	3.77	3.58	3.83	3.68

Table 3: Throughput in Operations/sec

Category	APACHE CACHE	ICS BASE	FLASH BASE
APPLICATION	28.43	42.27	27.49
incinterval	3.95	0.01	0.01
phxentdd	16.34	18.10	33.16
kwritev	6.88	6.77	13.43
kreadv	1.71	2.33	2.15
open	0.00	5.09	0.02
close	0.87	5.19	7.02
DATA	16.60	0.00	0.10
I/O	3.99	3.50	5.08
WAIT	2.05	0.00	0.00
select	3.21	0.01	0.36
naccept	1.41	0.00	2.78
stat	0.00	5.75	0.16
sigaction	2.13	0.01	0.00
Other	11.82	9.19	6.88
TOTAL	99.45	98.88	98.72

(a) 1 KB Requests

Category	APACHE CACHE	ICS BASE	FLASH BASE
APPLICATION	67.42	18.94	11.23
phxentdd	10.75	16.33	42.65
kwritev	0.93	35.52	29.51
kreadv	0.07	11.27	0.00
DATA	3.77	0.04	0.00
I/O	2.78	4.89	2.97
sync	0.00	0.00	3.33
level	1.10	2.26	0.51
Other	7.91	6.08	4.84
TOTAL	94.81	95.40	95.11

(b) 1 MB Requests

Figure 11: UTLD Breakdown for 3 WWW Servers

- `write()` still has a noticeable performance cost, from 7 to 10 percent, despite the fact that the files sent are only 1 KB.
- The process-based Apache server spends between 7 and 17 percent of their time in data-access TLB page faults (the DATA entry above). The single-process servers (Flash and CWS), on the other hand, do not pay any observable cost for TLB faults. This is presumably because a threaded server puts less memory pressure on the VM system than does a process-based server.
- In ICS, which does not cache files in user space, `open()`, `close()`, and `stat()` calls can add up to a significant performance penalty for small files, each consuming up to 5 percent of execution time. Flash and Apache-Cache, which cache both file state and content, do not spend significant time in these routines.

Figure 11 (b) presents the same information for servers when the requests are for the same 1 MB file. Several items are worth noting:

- For large files, most time is spent in the application itself, from 17 to 77 percent of execution time.
- The Ethernet device driver consumes from 10 to 42 percent of CPU cycles.
- In some cases, `write()` still has a substantial performance cost, as much as 35 percent of execution time.
- The costs for TLB faults are still present in the Apache-based server, although they contribute a relatively smaller proportion to execution time (under 4 percent).
- As one might expect, for serving large files, the `open()`, `close()`, and `stat()` calls do not significantly impact performance.