Performance Issues in WWW Servers

Erich Nahum, Tsipora Barzilai, Dilip Kandlur

***Abstract*— This paper evaluates techniques for improving operating system and network protocol software support for high-performance WWW servers. We study approaches in 3 categories: new socket functions, per-byte optimizations, and per-connection optimizations. We examine two proposed socket functions, `acceptex()`and `send_file()`, comparing `send_file()`'s effectiveness with a combination of `mmap()` and `writev()`. We show how `send_file()`provides the necessary semantic support to eliminate copies and checksums in the kernel, and quantify the benefit 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, without compromising interoperability. Results using WebStone show that our combination of mechanisms can improve server throughput by up to 64 percent, and can eliminate up to 33 percent of the packets in an HTTP exchange. Results with SURGE show an aggregate increase in server throughput of 25 percent.**

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 dramatically increasing 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 Windows NT [16], `acceptex()`and `transmitfile()`, and HP has a similar function in HPUX called `send_file()`. These API's streamline the programming interface used by a web server in a typical HTTP transaction, but what performance benefit, if any, do they provide? Does `transmitfile()`or `send_file()`show any improvement over the already available `mmap()`and `writev()`system calls?
- *per-byte optimizations.* It is well-known that data touching operations, such as copying and checksumming, are expensive [5, 11, 13, 23]. BSD-derived Unix operating systems [24] 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 [31], while continuing to exploit the benefits of existing file systems? What sort of performance impact will eliminating data touching operations have for WWW servers?
- *per-connection optimizations.* TCP connection management was not designed for client-server traffic, exchanging more packets that is semantically necessary. While the transition to persistent connections in HTTP 1.1 will improve this, most machines are still using HTTP 1.0. How can the per-connection overhead be reduced, without violating the TCP protocol specification?

The authors are with the IBM T.J. Watson Research Center.

We study these issues using a testbed of several IBM RS/6000 AIX workstations connected over 100 mbps Ethernet. We use Rice University's Flash WWW server, which exploits most currently-known user-level optimizations, and utilize the WebStone and SURGE WWW workload generators to drive the system with HTTP 1.0 requests.

Our experience confirms previous work showing that Web servers spend most of their time in the kernel [1, 17, 35]. The choice of execution model, threads or processes, significantly affects what performance improvements are available to WWW servers [15]. 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 studying 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 `send_file()`, comparing the latter's effectiveness with an combination of `mmap()`and `writev()`. We show how using `send_file()`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 `send_file()`offers no advantage over `mmap()/writev()`. However, a zero copy implementation can improve performance by up to 51 percent for large requests, and eliminating the checksum increases throughput by an additional 8 percent. Without compromising interoperability, our techniques for reducing packet exchanges eliminate up to 33 percent of the packets, and improve server performance by up to 20 percent for small transfers. Hence, the combination of these techniques effectively benefits the entire range of WWW server workloads. Macrobenchmark results using SURGE demonstrate an aggregate performance improvement of 25 percent.

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 plans for future work.

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 (for logging purposes).
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.
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.

Many WWW server performance optimizations reduce the frequency or cost of the above operations:

- Microsoft has added the socket functions `acceptex()` and `transmitfile()`to Windows NT. HP has a similar function `send_file()`in HPUX 11. `acceptex()` combines the `accept()`, `getsockname()`, and `recv()` system calls (steps 1, 2, and 3 above). `transmitfile()` and `send_file()`direct 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). The functions also include 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, eliminating the need for the `close()`system call (step 13). `send_file()`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 Windows NT in [16].
- Zeus [18] manages a cache of `mmap()`'ed files, as exposed by James Hu *et al.* [15]. 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 caches time information, avoiding `gettimeofday()` calls, and uses `writev()`, to combine the two `write()`calls (steps 10 and 11).
- Yiming Hu *et al.* [17] 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.

Table 1 summarizes in tabular form the steps taken for a web transaction, the available user-level optimizations for each step, and the possible operating system primitives that can be used. As can be seen, certain steps have more than one possible technique that can be used. 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 an single-process event-driven server such as Zeus does.

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.* [21, 22] 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 Zwaenepoel [12] 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 microkernels 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 [31]. Their system provides a new I/O interface and offers functionality such as zero-copy data movement and checksum caching. 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, 25, 26, 35]. Some of these identify performance issues that have been addressed in the AIX operating system that we employ, such as using hash tables for PCB lookup or carefully managing TCP connections in the TIME_WAIT state.

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 Zwaenepoel, 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 wish to minimize changes, and show how a single API change can achieve much of the benefits of I/O Lite. We provide an analysis of the `acceptex()`and `send_file()`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 used, the WWW client workload generators, the operating system, and the WWW server software.

3.1 Hardware

Our testbed consists of 4 IBM 43P RS/6000 workstations, with one machine acting as the server, and three others as clients. The server has 3 100 mbps Ethernet network interfaces, and each client is connected point-to-point full duplex with the server. Each machine has 128 MB of RAM and a 200 MHz PowerPC 604e processor. The 604e has separate on-chip instruction and data caches, each of which is 32 KB with 4-way associativity. The 43P's used also have a unified 1 MB direct-mapped secondary cache, and a SPECInt95 rating of 7.79.

3.2 Client Workload Generator Software

We use both the WebStone [34] and SURGE [6] workload generators to evaluate WWW server performance. Considerable debate [3, 6, 27] has occurred over how accurate benchmarks such as WebStone and SpecWeb [9] are as predictors of 'real-world' performance. For example, 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 still does not fully capture document sizes or other server workload characteristics such as document popularity [6]. As an example, Figure A shows the popularity of documents requested by the WebStone, SpecWeb and SURGE workload generators. For comparison purposes, we include document rankings made from the server logs of the Kasparov-Deep Blue Chess site from May 1997, and from the logs taken from IBM's corporate WWW site from June 1998. WebStone and SpecWeb each do a poor job of capturing popularity, but as can be seen, SURGE is quite close to the actual logs.

We thus use both WebStone and SURGE, but in different ways. We utilize WebStone as a *microbenchmark* to load the server with many concurrent requests for the *same file*. Varying the size of the file in different experiments, we can see how a performance optimization benefits requests for a transfer of a particular size. We use SURGE as a *macrobenchmark*, to load the server with concurrent requests for a *range* of files, to see how beneficial a particular technique will be for *aggregate* WWW server performance.

3.3 OS and Web Server Software

We use AIX version 4.3.1 on our machines, with the addition of several modifications that we developed: the functions `acceptex()`and `send_file()`, and a modified TCP protocol implementation. We discuss the implementation details in Section 4.

For our experiments, we wished to use the fastest WWW server possible, to show that our techniques could benefit even the most well-optimized software. To this end, we compared several different WWW servers on our testbed: Apache 1.2.4, Apache-Cache, Apache 1.3.0, Zeus, Flash, and Flash using `poll()`.

Apache is the freely available WWW server, 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 [29]. Apache is a process-based server, forking several processes which serially accept new connections. We evaluate both version 1.2.4, which has very few optimizations, and version 1.3.0, which has several performance improvements, including use of `mmap()`and `writev()`.

Apache-Cache is a version of Apache 1.2.4 adapted by Yiming Hu *et al.* [17] 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 2, this server is up to 48 percent faster than the base Apache. However, it maintains Apache's process architecture.

Zeus [18] is a commercial WWW server that is well-known for its performance [15]. Many vendors, including IBM, use Zeus for providing results at the SpecWeb WWW site [9]. The results here are from version 3.1.2.

Flash is a WWW server developed by Pai *et al.* [31] at Rice University as part of their work on I/O Lite. Flash is a single-threaded event-driven server that uses the `select()` system call and asynchronous I/O. Flash exploits almost 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()`.

Flash-Poll is a version of Flash which we modified to use `poll()`rather than `select()`. Banga and Mogul [4] have shown that event-driven servers that use `select()`, such as Flash, can suffer performance problems when managing large numbers of active connections. To avoid these problems, we adapted Flash to use `poll()`, a similar function derived from SVR4, which is available on some forms of Unix, including AIX.

Table 2 shows the throughput for the servers across a range of requested file sizes, where throughput is in HTTP operations/second. The numbers are the average over a 60 second interval, and are generated using the WebStone benchmark, which is configured to use 30 client processes. All Apache servers were configured with HostNameLookups off, and MaxRequestsPerChild set to 30000, for maximum performance. As can be seen in Table 2, Flash provides the highest throughput across a range of file sizes, particularly the small files that are most frequently requested from WWW servers. Zeus was configured using the default options. While Zeus' performance might be improved through better option tuning, such as seen on the SpecWeb site, since it is not open source, it is not amenable for our purposes. Since Flash-Poll provides the best performance, we use it as the baseline upon which we evaluate our techniques. Readers should assume that all subsequent numbers reported with Flash are obtained with the version that uses `poll()`.

4 Results

In this Section we present our results, showing how various optimizations benefit requests for different file sizes. To illustrate the performance benefits of each technique, we incrementally modify the server and measure the difference in throughput. By observing how performance changes as each feature is added, we can quantify what the utility of that feature is. Using WebStone, we evaluate the optimizations in the following order: the proposed socket functions, per-byte optimizations, and per-connection optimizations. We then show the aggregate performance improvements using SURGE.

File Size (KB)	Flash Poll	Flash Poll AE	Diff (%)
1	1140.11	1151.89	*1.03*
2	1059.26	1072.58	*1.26*
4	904.15	913.93	*1.08*
8	722.22	727.99	*0.80*
16	501.92	504.59	*0.53*
64	187.72	188.49	*0.41*
256	54.36	54.23	*-0.24*
1024	13.55	13.56	*0.07*

Table 3: Throughput in HTTP ops/sec

In our WebStone experiments, throughputs reported are the average of 10 runs. In all of our tests, 90 percent confidence intervals are less than 1 percent of the raw numbers, which means all changes that cause a greater than 1 percent differences in the numbers are *statistically significant*.

4.1 Evaluating Proposed Socket Functions

As described earlier in Section 2, `acceptex()`combines the `accept()`, `getsockname()`and `recv()`system calls. The kernel implementation was simple, merely combining the `soaccept()` and `sorecv()` kernel primitives. Modifying Flash to use `acceptex()`was also straightforward. Table 3 shows the server throughput for the Flash server with and without the `acceptex()`system call. The function makes a small difference of about 1 percent for small transfers.

The `send_file()`system call, also described in Section 2, had a more complex kernel implementation. The first version of `send_file()`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 `send_file()`implementation also supports the header buffer option, which simply calls `m_copym()` on the passed user buffer before reading the file, and the socket close option, described in more detail in Section 4.3.

Adding the `send_file()`support to Flash involved slightly more work than `acceptex()`. We removed Flash's cache of `mmap()`'ed files and instead called `send_file()`on each request, rather than using `writev()`on files that have been `mmap()`'ed. We did, however, retain Flash's mechanism to cache open file descriptors, in order 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.

Table 4 shows the change in throughput for the Flash server after adding support for `send_file()`. Here the header option is used, rather than incurring an additional `write()`call. As we can see, performance degrades by up to 18 percent. Due to an artifact of the way AIX is implemented on the PowerPC architecture, AIX does not possess an integrated I/O system. In other words, the network subsystem and the file system buffers are in separate

File Size (KB)	Flash Poll	Flash Poll SendFile	Diff (%)
1	1140.11	1081.81	*-5.11*
2	1059.26	1012.09	*-4.45*
4	904.15	882.65	*-2.38*
8	722.22	707.47	*-2.04*
16	501.92	485.73	*-3.23*
64	187.72	180.65	*-3.77*
256	54.36	44.41	*-18.30*
1024	13.55	11.01	*-18.75*

Table 4: Throughput in HTTP ops/sec

File Size (KB)	Flash Poll SendFile	Flash Poll SendFile Mbuf	Diff (%)
1	1081.81	1158.91	*7.13*
2	1012.09	1110.48	*9.72*
4	882.65	998.89	*13.17*
8	707.47	842.05	*19.02*
16	485.73	632.47	*30.21*
64	180.65	271.36	*50.21*
256	44.41	82.22	*85.14*
1024	11.01	20.03	*81.93*

Table 5: Throughput in HTTP ops/sec

kernel address spaces. In-kernel measurements found that copying data from the file system is actually slower than copying from user space. While we did not investigate this extensively, we believe it is because copying requires invoking VM primitives, which are typically expensive. Experiments with `send_file()`that did not exploit the header option (not shown for space reasons) performed even more poorly, both due to the extra `write()`call and because of interactions with the Nagle algorithm [28]. For simplicity, readers should assume all further results reported with `send_file()`use the header option.

Thus, a single-copy `send_file()`seems to offer no performance benefit over a `mmap()/writev()`combination.

4.2 Evaluating Per-Byte Optimizations

As described earlier, buffers in the network subsystem and in the file system reside in different address spaces in AIX. Due to this limitation, `send_file()`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 a `send_file()`implementation that caches mbufs. 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 modified the `send_file()`implementation to include a caching mechanism within the kernel that is separate from the VM system. On each `send_file()`, the kernel checks whether 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.

Table 5 shows the throughput for the Flash server using the single-copy and mbuf caching versions of `send_file()`. Here we see substantial performance improvements of up to 85 percent, with caching of mbufs making progressively larger differences for files up to 1 MB. Removing a copy may have particular performance benefits on our platform, since the PowerPC 604e has a write-through cache and the 43P workstation has a direct-mapped second-level cache. Data thus must be written all the way to main memory, as opposed to a write-back cache where writes can proceed at cache speed. Eliminating writes can thus remove "cache-busting" behavior, especially in experiments where large files are requested.

File Size (KB)	Flash Poll SendFile Mbuf	Flash Poll SendFile Mbuf Cksum	Diff (%)
1	1158.91	1178.34	*1.68*
2	1110.48	1120.70	*0.92*
4	998.89	1019.06	*2.02*
8	842.05	878.48	*4.33*
16	632.47	673.03	*6.41*
64	271.36	294.23	*8.43*
256	82.22	89.32	*8.64*
1024	20.03	21.82	*8.94*

Table 6: Throughput in HTTP ops/sec

While eliminating the copy reduces data-touching operations, it does not completely eliminate them, since the data must still be read into the CPU to calculate the Internet checksum. Thus, each byte of data is still read, but not written. AIX allows disabling the Internet checksum for a specific interface, which provides us a close approximation of how performance would change if the checksum was offloaded to the adaptor. (The checksum offload feature is available on certain ATM and gigabit Ethernet interfaces for AIX; however, we did not have access to these newer adaptors.) The Ethernet driver we used supports DMA of the data from the hosts' memory onto the network interface cards' memory, thus with the checksum disabled, the host CPU does not touch the data at all.

Table 6 shows the change in throughput for the Flash server when the host CPU does not perform the checksum. Again, we see performance improving with larger file transfers, with the largest gains in performance of just under 9 percent.

4.3 Per-Connection Optimizations

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

Figure 2 shows the sequence of TCP packets exchanged in a typical HTTP transaction requesting a 1 KB file, taken from `tcpdump` [20]. 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

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

Figure 2: Original TCP Packet Exchange

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

Figure 3: Piggybacking the FIN

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 `send_file()`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 function pointer `pr_usrreq()`, 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 relatively simple matter to add an additional option, `PRU_SEND_DISCONNECT`, which appends data to the socket buffer, changes the TCP connection state to the TCP_FIN_WAIT_1 state, and then invokes `tcp_output()`. Thus, when the close option is set, our `send_file()`code calls `pr_usrreq()` with `PRU_SEND_DISCONNECT` when sufficient send buffer space is available. Figure 3 (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 [19], 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 the interrupt overhead, copying the packet from the network device, and protocol processing.

Table 7 shows the subsequent change in throughput between experiments with and without the close option on

File Size (KB)	Flash Poll SendFile Mbuf Cksum	Flash Poll SendFile Mbuf Cksum Close	Diff (%)
1	1178.34	1267.65	*7.58*
2	1120.70	1193.33	*6.48*
4	1019.06	1076.72	*5.66*
8	878.48	882.60	*0.47*
16	673.03	672.53	*-0.07*
64	294.23	294.40	*0.06*
256	89.32	89.37	*0.06*
1024	21.82	21.90	*0.37*

Table 7: Throughput in HTTP ops/sec

our 100 Mbit Ethernet testbed. In requests for small files, up to a 7 percent increase in HTTP throughput is observed. Note from Table 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 larger transfers in situations where the sender is not limited by the congestion window, which is a function of many things, including the segment size. On our testbed using Ethernet, which has a 1500 byte MTU, this is up to 4 KB transfers. On earlier experiments using ATM (not shown for space reasons), which had a 9 KB MTU in our testbed, the close option benefitted transfers of up to 16 KB. While this optimization only affects transfers for small files, calculations from our logs discussed in Section 3 showed that the median HTTP response for the `www.ibm.com` site was 265 bytes, and that 85 percent of transfers were for less than 1800 bytes. The median and 85th percentile transfers of the Deep Blue Chess site were 1584 and 13947 bytes, respectively. 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.

Examining the packet exchanges in Figure 3, it can be seen that further reductions in packets are possible, given the redundant information being communicated. For example, in Figure 3 (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 TCP requires every packet to contain an ACK (except for the initial SYN packet), and acknowledgments in TCP are cumulative. 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 [7, 8, 32, 33].

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

Figure 4: 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

Figure 5: Delaying ACK of SYN-ACK

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 4 shows the TCP packet exchange after removing the ACK of the server FIN when the TCP state is `TCPS_ESTABLISHED`. This was enabled by changing a line in `tcp_input()` which forces `TF_ACKNOW` to be set when a FIN is received. Instead, `TF_DELACK` is set, enabling the normal 200 ms. delayed ACK timer, so that the ACK will be piggybacked on the next outgoing packet, which in this case is the client's FIN. Table 8 shows the change in performance, between experiments with and without the delayed ACK of the FIN, with increases in throughput of up to 5 percent.

Figure 5 shows the exchange after removing the ACK of the server's SYN-ACK. Again, this was achieved by changing a line in `tcp_input()` which sets `TF_ACKNOW` when a SYN-ACK is received to setting `TF_DELACK`, and by preventing `needoutput` from being set in that case. Table 9 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 5 percent additional increase in performance.

File Size (KB)	Flash Poll SendFile Mbuf Cksum Close	Flash Poll SendFile Mbuf Cksum Close DelAckFin	Diff (%)
1	1267.65	1341.34	*5.81*
2	1193.33	1254.58	*5.13*
4	1076.72	1110.84	*3.17*
8	882.60	904.31	*2.46*
16	672.53	686.54	*2.08*
64	294.40	295.55	*0.39*
256	89.37	89.30	*-0.08*
1024	21.90	21.80	*-0.46*

Table 8: Throughput in HTTP ops/sec

File Size (KB)	Flash Poll SendFile Mbuf Cksum Close DelAckFin	Flash Poll SendFile Mbuf Cksum Close DelAckFin DelAckSyn	Diff (%)
1	1341.34	1407.55	*4.94*
2	1254.58	1320.16	*5.23*
4	1110.84	1166.91	*5.05*
8	904.31	937.51	*3.67*
16	686.54	708.30	*3.17*
64	295.55	298.16	*0.88*
256	89.30	89.15	*-0.17*
1024	21.80	21.79	*-0.05*

Table 9: Throughput in HTTP ops/sec

File Size (KB)	Flash Poll	Flash Poll SendFile Mbuf Cksum Close DelAckFin DelAckSyn	Diff (%)
1	1140.11	1407.55	*23.46*
2	1059.26	1320.16	*24.63*
4	904.15	1166.91	*29.06*
8	722.22	937.51	*29.81*
16	501.92	708.30	*41.12*
64	187.72	298.16	*58.83*
256	54.36	89.15	*64.00*
1024	13.55	21.79	*60.81*

Table 10: Throughput in HTTP ops/sec

We have not yet evaluated how delaying these ACKs might affect the performance of other applications that use TCP, such 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 for use with port 80.

4.4 Evaluating Performance with SURGE

Table 10 presents the total change, as measured by WebStone, in HTTP throughput as a result of our optimizations. Across a range of transfer sizes, improvements can be seen of up to 64 percent. However, as discussed in Section 3, it is important to know not only the benefits across a set of file sizes, but how the optimizations benefit aggregate

Configuration	SURGE Ops/sec	Diff (%)
Flash-Poll	437.72	
+ send_file()	418.05	*-05*
+ Mbuf Caching	519.83	*+20*
+ Checksum Offload	555.14	*+06*
+ FIN Piggyback	560.66	*+01*
+ Delayed Ack of FIN	571.60	*+02*
+ Delayed Ack of SYN	581.56	*+02*
Total Improvement:		*+25*

Table 11: HTTP Throughput in ops/sec (SURGE)

server performance for a statistically representative workload. To that end, we employ SURGE. Table 11 shows the incremental changes in throughput in HTTP operations/second as measured by SURGE. Throughput is the average over a 10 minute sampling period after a 2 minute warmup interval. Here, SURGE is configured using 2000 unique documents, 2 processes and 250 threads per machine, for a total workload of 1500 user-equivalents. The aggregate performance improvement is 25 percent, with most of the benefits coming from the data-touching optimizations but the per-connection optimizations still having a statistically significant impact.

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. We summarize our conclusions as follows:

- *new socket functions.* We find little or no increase in performance using the `acceptex()`function, on either process-based or thread-based WWW servers. Profiling using UTLD, an IBM-internal AIX kernel profiling tool, shows that servers spend a relatively small amount of time in the `accept()`, `getsockname()`, and `read()`system calls. A `send_file()`implementation that incurs a single copy provides no advantage over a combination of `mmap()`and `writev()`, even when the header option is exploited.
- *per-byte optimizations.* A `send_file()`implementation tied to an integrated I/O system, which does not copy data, provides substantially better performance. In our mbuf caching testbed, we observe an increase in throughput of up to 51 percent. We find that offloading the checksum to the network device can improve WWW server performance by up to 9 percent. Our mbuf cache mechanism can also be enhanced to allow caching of the checksum values in the mbufs, for network interfaces that do not support the checksum offload.
- *per-connection optimizations.* We show how the close option to `send_file()`provides the semantic support to enable piggybacking the FIN on the last data segment, eliminating one packet and improving throughput for small transfers by 7 percent. We also show how delaying acknowledgments for the FIN and SYN-ACK packets can eliminate 2 more packets, increasing performance an additional 11 percent in these cases. In total, we reduce the packets in small HTTP exchanges from 9 to 6, reducing network utilization and raising server throughput by up to 20 percent in these scenarios.
- *aggregate benefits.* Using SURGE as a macrobenchmark, we showed that the combination of techniques improved aggregate server performance by 25 percent.

We should point out that, while we have evaluated these optimizations in the context of a WWW server, they have utility for other programs as well. Reducing packet exchanges should help other TCP-based applications, and `send_file()`is a general function that can be used by other network servers, such as NFS, FTP, or SMB. In addition, it is easier for an application programmer to use `send_file()`than to implement a custom mechanism, such as a cache of `mmap()`'ed files. Finally, a `send_file()`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.

As a result of our findings, IBM's AIX division has released these features in AIX 4.3.2.

For future work, we plan to evaluate these 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 Paul Barford for the SURGE code, Yiming Hu for his Apache modifications, Chij-Mehn Chang for the original prototypes of `acceptex()`and `transmitfile()`, and Daisy Chang for her industrial-strength implementation of `send_file()`. Daisy Chang, Herman Dierks, Roch Guerin, Yiming Hu, Arvind Krishna, Dave Marquardt, David Mosberger, Rich Neves, Vivek Pai, and Satya Sharma all improved this work through discussions and/or 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 per-

formance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–646, Oct 1997.

[3] Gaurav 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] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] M. Frans Kaashoek, Dawson Engler, Gregory R. Ganger, Hector Briceno, Russell Hunt, David Mazieres, Tom Pinckney, Robert Grimm, John Jannotti, 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.

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

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

[24] 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.

[25] 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.

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

[27] David Mosberger and Tai Jin. httperf – a tool for measuring web server performance. In *Proceedings 1998 Workshop on Internet Server Performance (WISP)*, Madison, WI, June 1998.

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

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

[30] 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.

[31] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel.

I/O Lite: A copy-free UNIX I/O system. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

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

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

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

[35] 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.

A 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 [28]. Researchers have presented evidence [14, 30] 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. Otherwise, the segment will wait until the remote end acknowledges all outstanding data. 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 [14]. 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 `send_file()` 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.

Erich Nahum (A 91/ ACM 91) received his B.A. in Computer Science from the University of Wisconsin and his M.S. and Ph.D. from the University of Massachusetts. He is a Research Staff Member at the IBM T.J. Watson Research Center in Yorktown Heights, New York, USA. His research interests focus on network software performance, including WWW servers, TCP, clusters, and multiprocessors. His email address is nahum@watson.ibm.com

Tsipora Barzilai

Dilip Kandlur

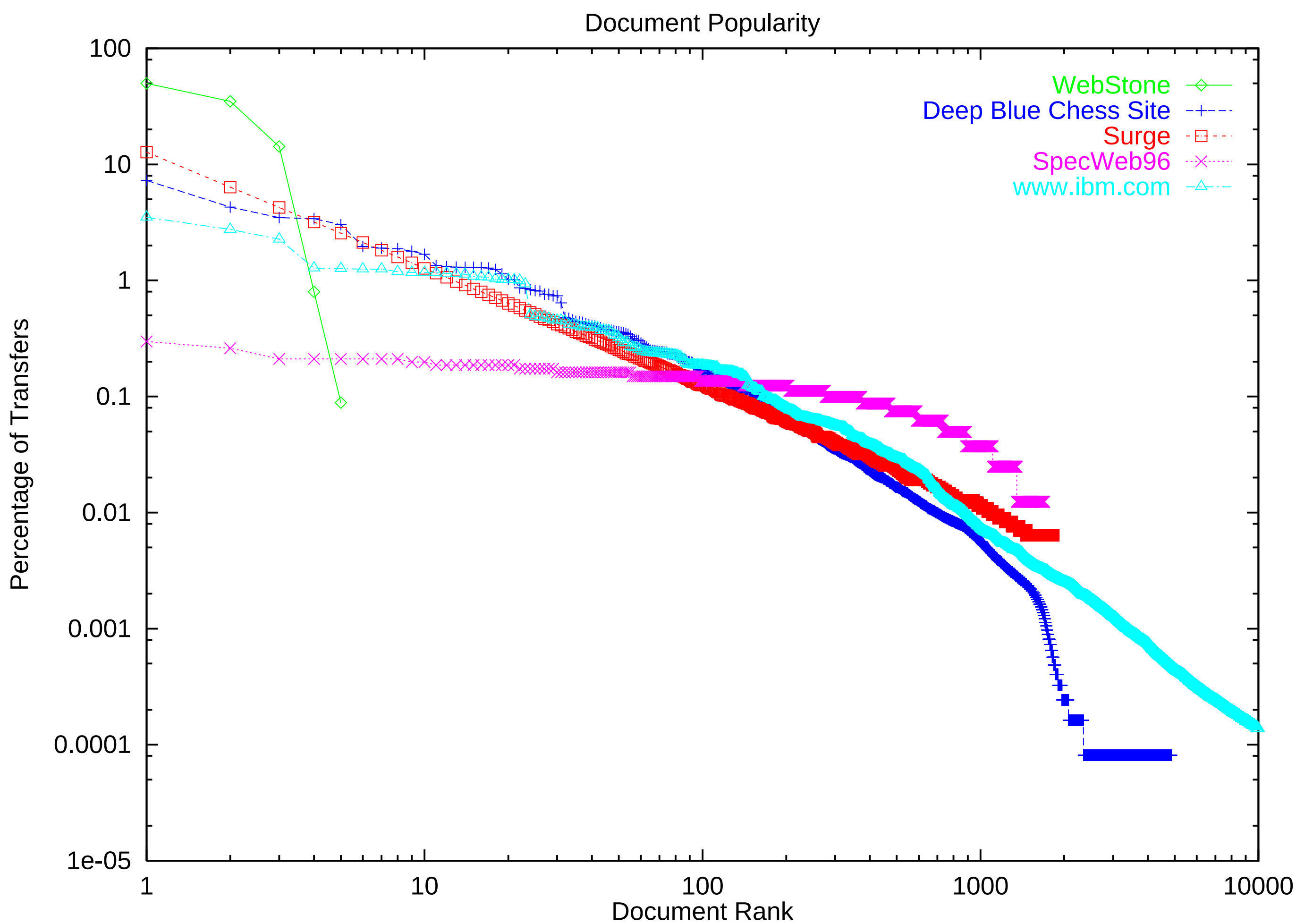

Figure 1: Document Popularity for Several Workloads

Step #	Web Server Operation	User-level Optimization	Operating System Optimization
1.	`accept()`connection		`acceptex()`
2.	`getsockname()`for peer		`acceptex()`
3.	`read()`request		`acceptex()`
4.	`setsockopt()`Nagle off		inherit socket option
5.	`gettimeofday()`	fast clock reads	
6.	HTTP request parsing	URI cache	
7.	`stat()`for requested file	cache `stat()`info	
8.	`open()`requested file	cache `mmap()`'ed files	file descriptor cache
9.	`read()`file into server	cache `mmap()`'ed files	`send_file()`
10.	`write()`HTTP header	`writev()`	`send_file()`header option
11.	`write()`file data	`writev()`	`send_file()`
12.	`close()`file	cache `mmap()`'ed files	file descriptor cache
13.	`close()`socket		`send_file()`close option
14.	`write()`log entry	buffered write	

Table 1: Web server operations and optimizations

File Size (KB)	Apache 1.2	Apache Cache	Apache 1.3	Zeus 3.1.2	Flash	Flash Poll
1	329.35	439.82	295.58	768.60	1086.50	1140.11
2	314.98	422.15	294.33	715.43	981.60	1059.26
4	273.82	369.47	285.65	628.23	849.27	904.15
8	245.57	334.50	263.37	535.60	688.92	722.22
16	198.70	274.72	218.92	384.10	480.52	501.92
64	99.48	138.45	111.82	154.32	179.47	187.72
256	33.90	41.25	39.97	48.30	51.70	54.36
1024	8.87	10.72	10.92	8.52	13.05	13.55

Table 2: Throughput in Operations/sec