Scalable kernel performance for Internet servers under realistic loads*

Gaurav Banga gaurav@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX, 77005

Jeffrey C. Mogul mogul@pa.dec.com
Digital Equipment Corp. Western Research Lab., 250 University Ave., Palo Alto, CA, 94301

Abstract

UNIX Internet servers with an event-driven architecture often perform poorly under real workloads, even if they perform well under laboratory benchmarking conditions. We investigated the poor performance of event-driven servers. We found that the delays typical in wide-area networks cause busy servers to manage a large number of simultaneous connections. We also observed that the *select* system call implementation in most UNIX kernels scales poorly with the number of connections being managed by a process. The UNIX algorithm for allocating file descriptors also scales poorly. These algorithmic problems lead directly to the poor performance of event-driven servers.

We implemented scalable versions of the select system call and the descriptor allocation algorithm. This led to an improvement of up to 58% in Web proxy and Web server throughput, and dramatically improved the scalability of the system.

1 Introduction

Many Web servers and proxies are implemented as as single-threaded event-driven processes. This approach is motivated by the belief that an event-driven architecture has some advantages over a thread-per-connection architecture [17], and that it is more efficient than process-per-connection designs, including "pre-forked" process-per-connection systems. In particular, event-driven servers have lower context-switching and synchronization overhead, especially in the context of single-processor machines.

Unfortunately, event-driven servers have been observed to perform poorly under real conditions. In a recent study of Digital's Palo Alto Web proxies, Maltzahn et. al. [11] found that the Squid (formerly Harvest) proxy server[5, 22] performs no better than the older CERN proxy[10]. This is surprising, because the CERN proxy forks a new process to handle each new connection, and process creation is a moderately expensive operation. This result is also in sharp contrast with the study by Chankhunthod et al.[5], which concluded that Harvest is an order of magnitude faster than the CERN proxy.

Maltzahn et. al. [11] attribute Squid's poor performance to the amount of CPU time Squid uses to implement its own memory management and non-blocking network I/O abstractions. We investigated this phenomenon in more detail, and found out that the large delays typical of wide-area networks (WANs) cause Squid to have a large number of simultaneously open connections. Unfortunately, the traditional UNIX implementations of several kernel features used by event-driven single-process servers do not scale well with the number of active descriptors in a process. These are the *select* system call, used to support non-blocking I/O, and the kernel routine that allocates a new file descriptor. (We refer to the descriptor-allocation routine as ufalloc(), as it is named in Digital UNIX, although other UNIX variants use different names, e.g., fdalloc().) A system running the Squid server spends a large fraction of its time in these kernel routines, which is directly responsible for Squid's poor performance under real workloads.

We designed and implemented scalable versions of select() and ufalloc() in Digital UNIX, and evaluated the performance of Squid and an event-driven Web server in a simulated WAN environment. We observed throughput improvements of up to 43% for the Web server, and up to 58% for Squid. We observed dramatic reductions in CPU utilizations at lower loads. We also evaluated these changes on a busy HTTP proxy server, which handles several million requests per day.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the working of a typical event-driven server running on a UNIX system. We also describe the dynamics of typical implementations of select() and ufalloc(). Section 3 describes our quantitative characterization of the performance problems in select() and ufalloc(). In Section 4 we present scalable versions of select() and ufalloc(). In Sections 5 and 6 we evalu-

*This paper will appear in the Proceedings of the USENIX Annual Technical Conference, New Orleans , LA, June 1998.

ate our implementation. Finally, Section 7 covers related work and and offers some conclusions.

2 Background

In this section we present a brief overview of the working of a typical event-driven server. We will also describe classical implementations of select() and ufalloc(). This will provide necessary background for the discussion in the following sections.

2.1 Event-driven servers

An event-driven server typically has a single thread which manages all connections to the server. The thread uses the select() system call to simultaneously wait for events on these connections.

When a call to select() returns, the server's main loop invokes event handlers for each of the ready descriptors. These handlers perform a variety of tasks depending on the nature of the particular event. For example, when a socket being used to listen for new connections becomes ready, the corresponding handler calls accept() to return a file descriptor for the new connection. Handlers invoked when a connection becomes ready for reading or writing perform the actual read or write to the appropriate descriptor. The execution of handlers may cause the addition or removal of descriptors from the set being managed by the server.

Event-driven servers are fast because they have no locking or context switching overhead. The same thread manages all connections, and all handlers are executed synchronously. A single-threaded server, however, cannot exploit any true concurrency in the stream of tasks. Thus, on multiprocessor systems, event-driven servers have as many threads as processors. Examples of event-driven servers include Squid[5, 22] and its commercial version NetCache[16], Zeus[25], thttpd[24] and several research servers[2, 8, 18].

2.2 select()

The *select* system call allows a user process to wait for events on a set of descriptors. A process can indicate interest in three types of events on a descriptor: events that make a descriptor *readable*, those that make it *writable*, and *exception* events. This information is passed to the kernel using three bitmaps. In each bitmap the kth bit indicates interest in events of that type for the kth descriptor. These bitmaps are value-result parameters, and the returned bitmaps indicate the sets of ready descriptors. Stevens[23] describes the select() interface in detail.

We describe the Digital UNIX implementation of select(). However, the classical BSD implementation of select() is similar to the Digital UNIX implementation. The main differences are related to the multithreaded nature of the Digital UNIX kernel. Thus our discussion is fully applicable to 4.3BSD and most BSD-derived implementations. Also, we discuss how select() works for descriptors that represent sockets, but our discussion and algorithms can be trivially extended to include descriptors that refer to other kinds of objects, such as vnodes. (Vnodes are kernel data structures used to represent files and devices.)

In Digital UNIX, the select() function in the kernel starts by creating internal data structures containing summary information about sockets that are marked in at least one input bitmap. Subsequently, select() calls do_scan(), which calls selscan() to check the status of each of the entities (vnodes or sockets) corresponding to the selected descriptors.

For each selected socket, selscan() enqueues a record referring to the current thread on the *select queue* of the socket. This is done so that the thread can be identified as waiting inside select() for events on the socket. selscan() then calls soo_select() for each socket, which checks to see if the condition that the process is interested in (i.e. the socket is readable, writable, or has pending exceptions) is true. If none of the conditions that the user process is selecting on are true, then do_scan() goes to sleep waiting for any of these to become true.

Note that the linear search in selscan() covers every socket of potential interest to the selecting process, independent of how many are actually ready. Thus, the cost is proportional to the number of file descriptors involved in the call to select(), rather than to the number of events discovered by the call.

When a network packet comes in, protocol processing may cause a condition on which do_scan() is blocked to become true. The thread that performs protocol processing for an incoming packet calls select_wakeup(), which wakes up all threads that are blocked in do_scan() awaiting this condition.

A thread that is woken up in do_scan() calls selscan(), which calls soo_select() for *all* the sockets that the corresponding call to select() specified in its three bitmaps. do_scan() also calls undo_scan() to remove this thread from select queues of the selected sockets.

2.3 ufalloc()

The kernel function ufalloc() is called to allocate a new file descriptor for a process. This function is called as a result of the open(), socket(), socketpair(), dup(), dup2() and accept() system calls.

UNIX semantics for file descriptor allocation require that the kernel allocate the lowest-numbered available descriptor. This prevents the use of a straightforward scalable implementation, such as a free list. Instead, all of the UNIX variants that we know of, including BSD-derived systems such as Digital UNIX, and System V Release 4 systems such as Solaris, use a linear search of the file descriptor table. The search starts with file descriptor

0 and continues to the first NULL entry. The cost of this search is roughly proportional to the number of open file descriptors, although it might complete before checking all of the possible descriptor table slots.

3 Problems in select() and ufalloc()

As we observed in section 1, Maltzahn et. al. [11] found that the Squid proxy server performs no better than the older CERN proxy under real workloads, contradicting the study by Chankhunthod et al.[5], which concluded that Harvest is an order of magnitude faster than the CERN proxy. Indeed, a simple LAN-based experiment using a simulated client load does show a big performance difference between Squid and the CERN proxy.

In an attempt to explain this peculiar result, we tried to understand why Squid's performance under real load is so much worse than under ideal conditions. One factor that is different in the two scenarios is that under real load Squid manages a much larger number of simultaneous connections than in a LAN-based test scenario. This is because of much larger delays experienced in WANs. Because WAN environments have larger round-trip times (RTTs), and are more likely to exhibit packet losses, HTTP connections tend to last much longer in WAN environments than in simple LAN environments. Therefore, for a given connection arrival rate, a WAN-based HTTP server will have more open connections than a server in a LAN environment.

Richardson's measurements of Digital's Palo Alto Web proxies [19] show between 30 and 950 simultaneously open connections, depending on time of day. Richardson's measurements also show that while the median response time is about 250 msec., the mean is 2.5 seconds: some connections stay open for a very long time. The large ratio of mean to median holds over a wide range of response sizes (although the 10:1 ratio only holds when all response sizes are considered together). This implies that at any given time, most of the open connections are *cold* (idle for long intervals), and only a few are *hot*.

Following this intuition, we tried to evaluate the effect of a large number of cold connections on Squid performance. We used DCPI [1] to profile a system running the Squid proxy under a carefully designed request load. To simulate the effect of large WAN delays, we set up a dummy HTTP client process on a client machine. This process opened a large number (100-2000) of connections to the Squid server but subsequently made no requests on these connections. We refer to this process as the *load-adding client*. Another process on the client machine simulated a small number (10-50) of HTTP clients, which repeatedly made HTTP requests of the proxy. Each request retrieved a 1259-byte response. We used the scalable client (S-Client) architecture from Banga and Druschel [3].

In our tests, we ran the Squid server process on an AlphaStation 500 (400Mhz 21164, 8KB I-cache, 8KB D-cache, 96KB level 2 unified cache, 2MB level 3 unified cache, SPECint95 = 12.3) equipped with 192MB of physical memory. The server operating system was Digital UNIX 4.0B, with the latest patches that were available at the time. The client machine was a 333Mhz AlphaStation 500 (same cache configuration as above, SPECint95 9.82) with 640MB of physical memory, running DUNIX 3.2C. The Squid version used was Squid-1.1.11. The client and server were connected using a 100Mbps FDDI network.

This experiment indicates that up to 53% of the system's CPU time is being spent inside select() (and its various components – selscan(), soo_select(), etc.). Up to 11% of the CPU is being spent by the user process in collating information from the bitmaps returned by select().

Our detailed results are shown in Figure 1. The x-axis represents the number of cold connections. Curves are plotted, for both 10 hot connections and 50 hot connections, showing the percentage of CPU time spent in kernel-mode functions related to select(), and the percentage of CPU time spent in the user-mode select() loop.

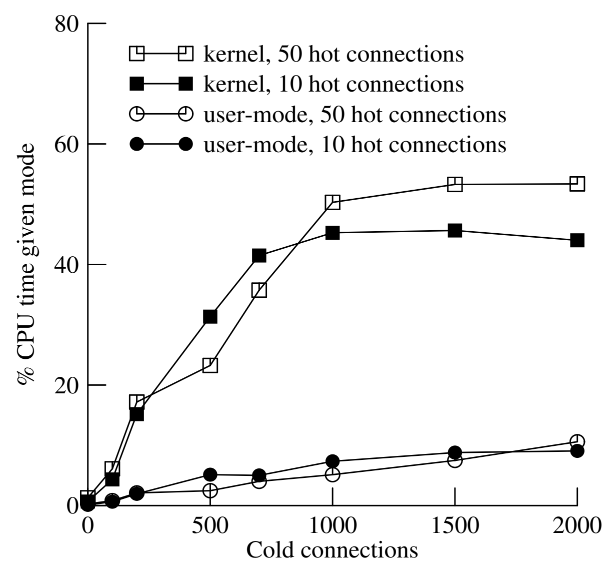

Figure 1: select() costs in unmodified kernel

Figure 1 shows that the costs of both the kernel select() implementation and the user-mode select() loop rise significantly with increasing numbers of cold connections. Also, these costs are relatively independent of the number of hot connections, up to about 1000 cold connections.

The costs are initially linear in the number of cold connections, but eventually they flatten out. As the number of cold connections increases, the system spends more CPU time in each call to select(), and so the calls to select() come less often. This causes the number of pending events returned by select() to increase (at low loads, select() usually returns just one pending event, but when called infrequently, it often returns several). The cost of each select() call is thus amortized over a larger number of interesting events. Thus, the total CPU cost of select(), which is proportional to the number of select()s per second times the cost of each select, tends to level off.

These numbers were generated with a request load of about 100 requests/second. At higher rates, select() is still important, but ufalloc() also consumes significant CPU time, because of its linear search algorithm. A typical DCPI profile for the system above, with 750 cold connections, 50 hot connections, and 220 new connections/second, is shown in Table 1.

CPU %	Procedure	Mode
21.91%	*all kernel* select *functions*	kernel
8.31%	soo_select()	kernel
7.56%	selscan()	kernel
4.82%	undo_scan()	kernel
1.22%	select()	kernel
17.79%	ufalloc()	kernel
4.23%	comm_select()	user
1.71%	_Xsyscall()	kernel
1.68%	_doprnt()	user
1.32%	idle_thread()	kernel
1.20%	memset()	user
1.15%	cache_lookup()	kernel
1.10%	namei()	kernel

750 cold connections, 50 hot connections, 220 requests/second

Table 1: Example profile for unmodified kernel

In summary, the current implementations of select() and ufalloc() do not scale well with the number of open connections in a server process. Both algorithms do work that is linear in the number of connections being managed by the process, and proxies in WAN environments tend to have many open connections. In the next section we will describe our implementation of scalable versions of these functions.

4 Scalable select() and ufalloc()

In this section we describe our design for scalable versions of select() and ufalloc(). We also describe our prototype implementation of these designs in Digital UNIX.

4.1 select()

Consider an event-driven server process waiting for activity on any of a few thousand sockets. Recall from Section 2 that select() always performs a full scan through all of these sockets, either to find those few that are currently ready, or to indicate that a thread is waiting for events on each of the sockets.

A full scan is also performed after the protocol code processes an incoming packet and calls select_wakeup() to unblock a thread waiting inside select(). The full scan is performed even though only a few of the sockets are actually ready. This wasted effort is expended because, between the call to select_wakeup() and the invocation of do_scan(), we throw away the information about the identity of the socket that has become ready. selscan() then does a significant amount of work to rediscover the set of ready sockets.

The key idea of our design is to preserve information about the change in the state of a socket between select_wakeup() and do_scan(). We use this information to prune both the initial scan, and the scan after the select_wakeup(), to inspect only those sockets that need inspection. These are the sockets either about which we have no prior information, or for which we have state-change hints from the protocol-processing layer.

We changed the Digital UNIX kernel to keep track of three sets for each thread, named READY, INTERESTED, and HINTS. (The first two of these sets actually consist of three component sets, one for read-ready descriptors, one for write-ready descriptors and one for exceptions.) The INTERESTED set is the subset of sockets that the thread is currently interested in selecting on. The READY set is a subset of the INTERESTED set and includes those sockets which the kernel thinks are *ready*. The kernel maintains state-change information about sockets in the INTERESTED set, rather than for the full set of sockets open for a thread. This state-change information is maintained as the HINTS set. The HINTS set includes sockets that might have become ready since the last call to select(), and is updated by the protocol layer when a packet arrives for a socket.

Each call to select() specifies a SELECTING set for the thread, which is used to compute the new values of the READY and INTERESTED sets. select() uses the HINTS and READY sets to prune its initial scan. It checks only those sockets which are in the SELECTING set and either:

1. are not in the old INTERESTED set, or
2. are in the old READY set, or
3. are in the HINTS set

Mathematically, we can express the computation of these sets as:

$$INTERESTED_{new} = SELECTING \cup INTERESTED_{old}$$

$$READY_{new} = \mathcal{C}(INTERESTED_{new} \cap (\overline{INTERESTED_{old}} \cup READY_{old} \cup HINTS))$$

where $\mathcal{C}$ expresses the computation of checking the status of descriptors in its argument set.

The computation of $\mathcal{C}$'s argument set above appears to have complexity proportional to the size of the SELECTING set. We took care to optimize this computation and its data-cache footprint. The resulting code has a very small cost relative to other parts of select().

The set returned from select() is:

$$READY_{to\_user} = SELECTING \cap READY_{new}$$

A descriptor must be removed from the INTERESTED sets of *all* threads in a process at some point between the time that the descriptor is closed and the time that it is next allocated by *any* thread in the process.

For each socket, we record the set of processes that have a reference to the socket. In the protocol processing code, when a packet comes in for a socket, sowakeup() records a hint in the HINTS sets of each of the threads in the referencing processes for which this socket is present in the INTERESTED set of the thread. sowakeup() also wakes up all such threads that are blocked in select(). After a thread is woken up in select(), it scans only those sockets in its HINTS set.

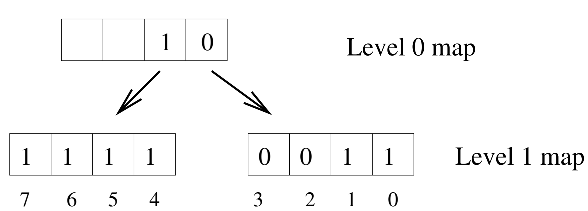

Figure 2: Two-level ufalloc bitmap

4.2 ufalloc()

The existing ufalloc() implementation uses a linear search to find the lowest-numbered free descriptor. We converted this into a logarithmic-time algorithm by adding an auxiliary data structure, a two-level tree of bitmaps. The collection of all the level-1 nodes can be thought of as a single bitmap; each bit in this bitmap describes the allocation state of one file descriptor. One-valued bits in this bitmap correspond to allocated descriptors. The level-1 bitmap is stored as an array of nodes.

Each bit in the level-0 bitmap describes the state of an entire level-1 node. One-valued bits in this bitmap correspond to level-1 nodes with no zero bits; a zero-valued bit in the level-0 bitmap corresponds to a level-1 node with at least one zero bit.

Figure 2 shows an example of such a tree. For simplicity, this figure depicts the nodes as 4-bit integers, although our actual implementation uses 64-bit integers. We use the Alpha's little-endian bit-order in this example. The example tree shows that descriptors 0, 1, and 4 through 7 are allocated, while descriptors 2 and 3 are free.

When a process wants to allocate a new file descriptor, the level-0 bitmap is searched for the first zero bit. The index of this bit is used as an index into the array of level-1 nodes, and the indexed node is then searched to find the first zero bit. Efficient algorithms exist for finding the first zero bit in a word, but we have found that a simple linear search is sufficiently fast, since the dominant cost on modern CPUs is the number of data-cache misses, not the number of instructions executed.

When a descriptor is deallocated, the appropriate bits are cleared in both bitmaps. This leads to a constant-time cost for deallocation.

With the level-1 nodes and the entire level-0 bitmap represented as 64-bit words, this algorithm directly supports 4096 descriptors per process. A straightforward generalization to a deeper tree would support an enormous number of descriptors, even if a smaller word size were used.

5 Experimental Evaluation

We evaluated the effects of our implementation of select() and ufalloc() on the performance of two event-driven Internet servers: the Squid proxy, and the thttpd [24] Web server (we used a modified version of thttpd with numerous performance improvements [18]). These experiments were performed using the same server and client systems describe in Section 3. We also measured the effect of our changes on the performance of Digital's Palo Alto proxies.

5.1 Scalability with respect to connection rate

The S-Client architecture introduced by Banga and Druschel [3] allows the generation of high HTTP request rates, using a small number of client machines. We used S-Clients to vary the load on the server. At the lowest load, the server is underutilized; at the higher loads, the server is the bottleneck.

For each request rate, we ran two kinds of benchmarks. In the naive benchmark, we used only enough S-

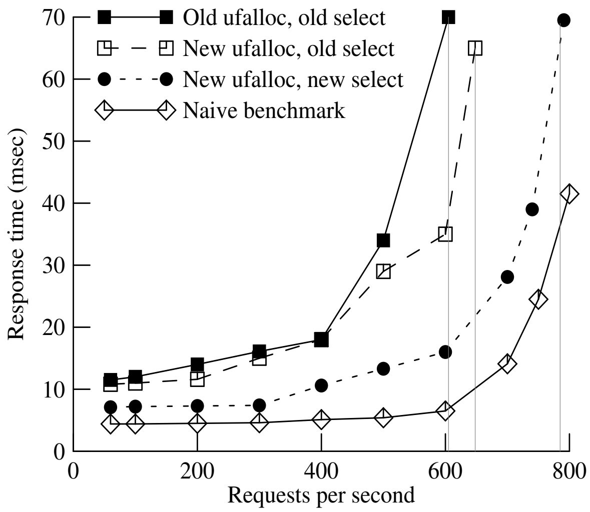

Figure 3: Squid response times – 1259-byte files

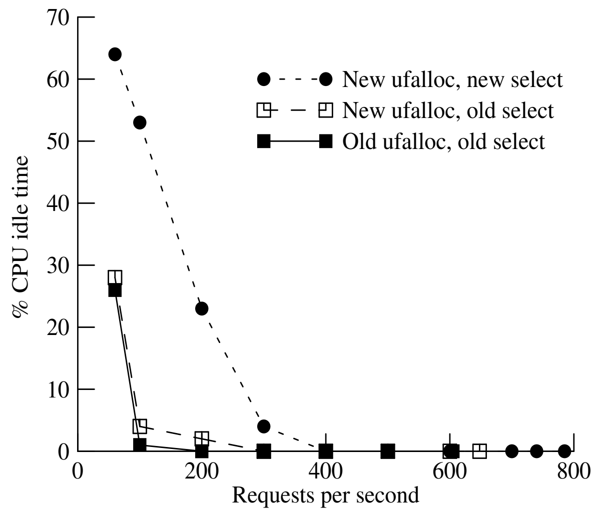

Figure 4: Squid idle time – 1259-byte files

Clients to generate the desired request rate. In the more realistic benchmark, we also used a load-adding client, to simulate the presence of long-delay connections. The load-adding client was run with 750 infinitely slow connections. (We show the effect of varying the number of slow connections in Section 5.2.)

All clients, in all of the experiments, repeatedly requested a single file of a fixed sized. In some experiments, we used an 8192-byte file; this is within the range of typical response sizes reported for the Web. In other experiments, we used a 1259-byte file; the shorter file size places more emphasis on per-connection overheads.

For our experiments using the Squid proxy server, we arranged things so that each request received by the proxy would generate an "If-Modified-Since" message from the proxy to the origin server, but the actual data would be served from the proxy's cache. The origin server ran on identical hardware (a 400Mhz AlphaStation 500), using the thttpd server program; we ensured that the origin server was never the bottleneck.

Figure 3 shows how the response time of the Squid proxy varies with request rate, for 1259-byte files. The results for all kernels on the naive benchmark are effectively identical; for the realistic benchmark, we plot different curves for the different kernels. For each curve, the final point shows the "saturation throughput" for the given kernel; beyond this point, increasing the offered load did not increase throughput. This figure clearly shows that the presence of adding slow connections in the realistic benchmark drastically reduces the throughput achieved with the unmodified kernel relative to the naive benchmark. It also shows that our new implementations of select() and ufalloc() solve this performance problem. The performance of the fully modified kernel is nearly independent of the presence of many slow connections.

Figure 4 shows the effect of the new versions of select() and ufalloc() on server CPU idle time, also for 1259-byte files. At lower request rates, where the server is underutilized, our modifications greatly increase idle time for the realistic benchmark. The increase in idle time reflects the improved scalability of the system in the presence of cold connections.

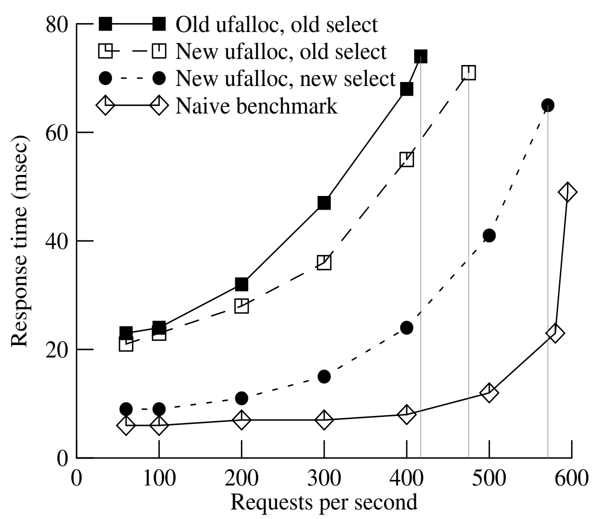

Figure 5: Squid response times – 8192-byte files

Figure 5 shows the response time of the Squid proxy for 8129-byte files. As in Figure 3, the fully modified kernel provides a higher saturation request rate than the original kernel, and yields lower response times at all request rates. However, the new kernel's performance on the realistic benchmark does not come quite as close to the performance of the naive benchmark; this may be due to data-cache collisions between the larger packets and the kernel's data structures. In these tests, the unmodified kernel showed no idle time for all request rates, while the new kernel showed some idle time up to 300

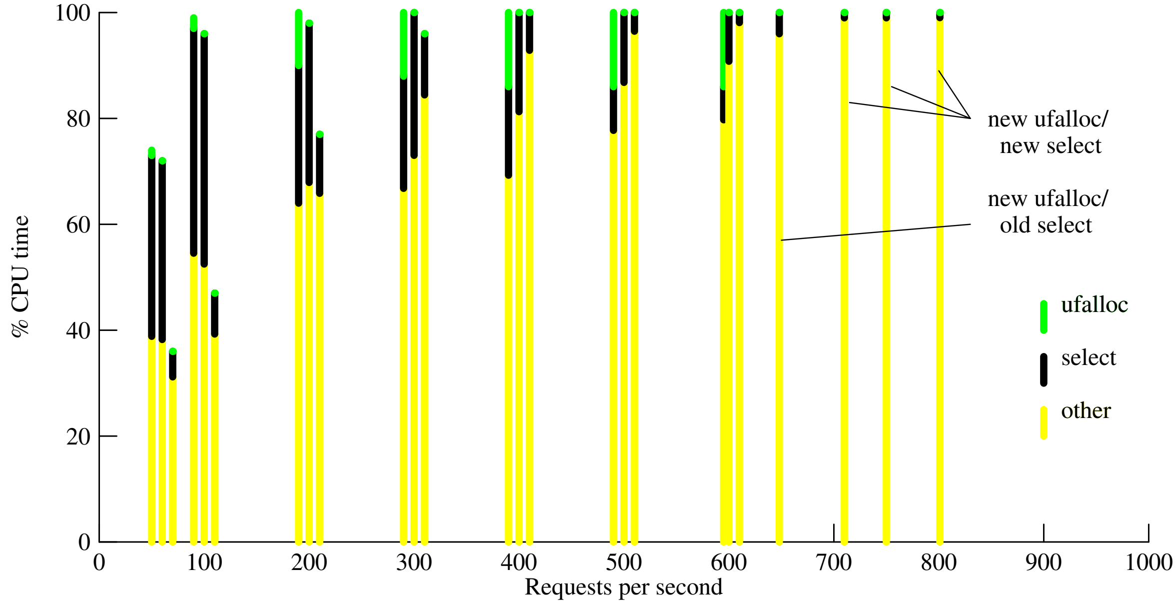

Figure 6: CPU share of ufalloc() and select(), Squid Proxy – 1259-byte files

requests/sec.

We used DCPI to obtain CPU time profiles of the server. Figure 6 shows the fraction of CPU time used in select() and in ufalloc(), for various request rates, using 1259-byte files. (The results for tests using 8192-byte files are analogous.) In each group of three bars, the leftmost bar represents the unmodified kernel, the center bar represents the kernel with the new ufalloc(), and the rightmost bar represents the kernel with new versions of both select() and ufalloc(). At rates above 600 requests per second, each bar is independently labelled. The top section of each bar shows the CPU time spent in ufalloc(), and the middle section shows the CPU time spent in select(). The bottom section of each bar ("others") shows the CPU time used for all other components of the server, including user-mode code. Idle time is not shown; it corresponds to the space above the bar, if any.

Figure 6 shows that the new ufalloc() almost entirely eliminates the CPU costs of descriptor allocation in all of the tested configurations. The new select() also costs much less than the old select().

When the server is underutilized, at rates below about 200 requests per second, the CPU profiles show that the new select() provides an additional performance impact: although we have not changed the implementation of any code covered by the "others" part of the profile, and the total throughput has not changed, the CPU costs of the "others" components has been reduced, relative to the unmodified kernel. We attribute this to better data-cache behavior, because the new select() has a much smaller data-cache footprint than the original implementation. The modified ufalloc() may also have a similar effect on cache performance. The improved data-cache footprint of select() is probably responsible for some of the throughput gains in the server-bound configurations.

CPU %	Procedure	Mode
21.96%	*all idle time*	kernel
11.49%	*all kernel* select *functions*	kernel
11.24%	select()	kernel
0.15%	new_soo_select()	kernel
0.10%	new_selscan_one()	kernel
16.37%	comm_select()	user
2.61%	tcp_slowtimo()	kernel
1.73%	tcp_fasttimo()	kernel
1.39%	_doprnt()	user
1.21%	_Xsyscall()	kernel
1.10%	_XentInt()	kernel
1.00%	bcopy()	kernel
0.91%	read_io_port()	kernel
0.90%	memset()	user

750 cold connections, 50 hot connections, 220 requests/second

Table 2: Example profile for modified kernel

As can be seen in Figure 3, even with our kernel modifications, the realistic benchmark still causes a small performance degradation compared to the naive benchmark. We attribute this to the inherently poor scalability of the select() programming interface. This interface passes information proportional to the total number of active connections on each call to select(). Moreover, when select() returns, the user process must do work proportional to the total number of active connections to discover which descriptors have pending events. Finally, select() overwrites its input bitmaps, thus requiring additional user-mode work to create these bitmaps on each call. These costs cannot be eliminated with the current interface. In a separate publication [4], we propose a new, scalable interface to replace select()

Table 2 shows a profile of the modified kernel, made under the same conditions as the profile of the original kernel shown in Table 1. The new kernel spends 22% of the time in the idle loop, compared to almost no idle time for the original kernel. The original kernel spent about 22% of the CPU in select() and its subroutines, and 18% of the CPU in ufalloc(). The modified kernel spends 11% of the CPU in select(), and virtually none in ufalloc(). However, the busiest function in the system is now the user-level comm_select() function, using 16% of the CPU. The almost 28% of the CPU together consumed by the kernel select() and user-mode comm_select() functions is a result of the poorly scaling bitmap-based select() programming interface.

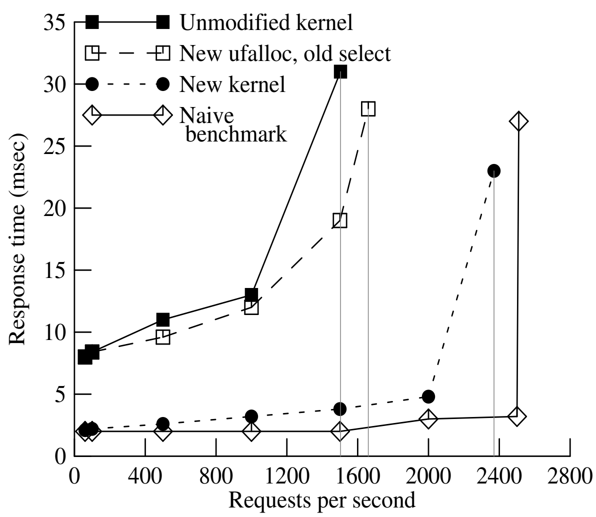

Figure 7: Response time for thttpd – 1259 byte files

Our experiments using the thttpd [24] Web server gave similar results. Using our modified kernel (with new implementations of both select() and ufalloc()), server throughput (at server saturation) improved by 58% for 1259-byte files, as shown in figure 7. For 8192-byte files, throughput increased by 37%; further improvement may have been limited by the available network bandwidth, rather than by the server. At lower request rates, the modified kernel showed much more idle time. For example, at 100 requests/sec. for a 1259-byte file, the unmodified kernel showed 16% idle time; the modified kernel showed 88% idle time. At at 100 requests/sec. for an 8192-byte file, the unmodified kernel had no idle time, but the modified kernel still showed 73% idle time.

5.2 Scalability with respect to connection count

To demonstrate that our implementations of select() and ufalloc(), unlike the original code, does scale well as the number of cold connections increases, we performed another series of experiments. In these experiments, we varied the number of connections from the load-adding client, between 0 and 2000 connections, and then increased the request rate until the server was saturated.

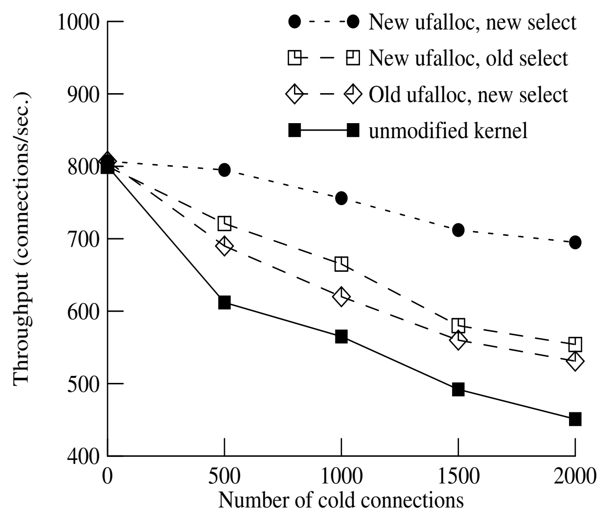

Figure 8: Performance of Squid Proxy – Scalability

Figure 8 shows that the throughput of the original kernel drops by 44% as the number of of cold connections increases from zero to 2000. The figure also shows that the kernel with our scalable ufalloc() has a somewhat smaller dependency on the number of cold connections, and for the kernel with our implementations of both select() and ufalloc(), its throughput drops by only 14% over the same range. We believe that the remaining dependency results from the user-level costs of the programming interface for select().

6 Performance of a live system

Digital Equipment Corporation operates a Web proxy system, in Palo Alto, California, that serves a large fraction of Digital's internal users. During a typical weekday, the system handles as many as 2.6 million HTTP requests, from at least 5570 individual client hosts.

We installed our modified kernel on the proxy server, a 500 MHz AlphaStation 500 system (21164A processor, SPECInt95 = 15.0) with 512 MBytes of RAM. We then ran the system using either the unmodified kernel or our modified kernel, each for an entire calendar day (midnight to midnight, Pacific Time), and collected extensive monitoring information.

During these these tests, the proxy server used version 3.1.2c-OSF of the NetCache software [16] from Network Appliance, Inc. Like Squid, NetCache was based on the Harvest Cache software, although NetCache and Squid

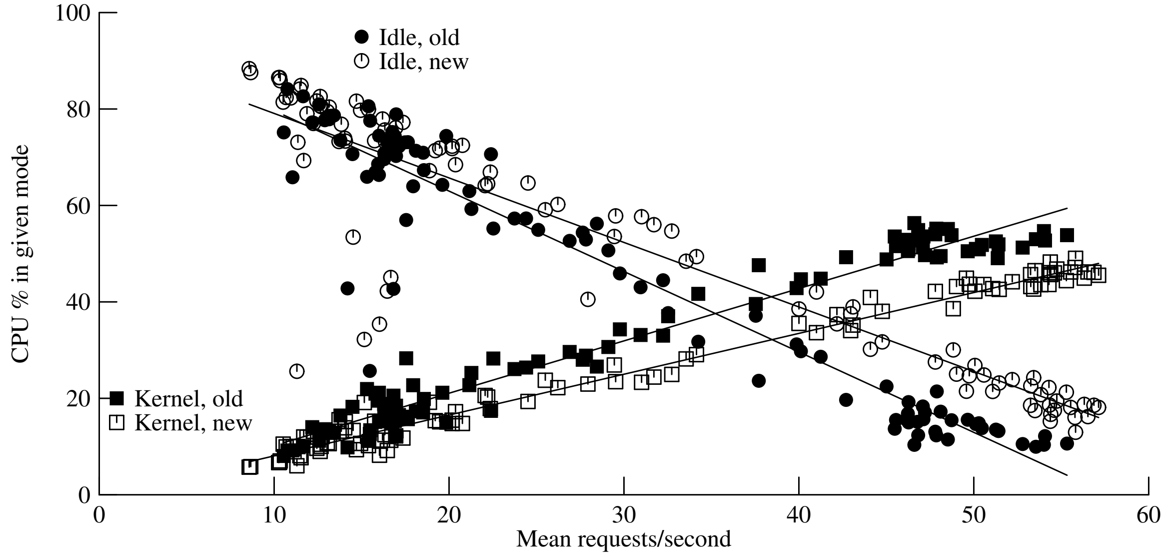

Figure 9: CPU costs as a function of request rate

Date	Kernel version	Requests handled	Max. alloc. fds	Peak req. rate
1998-04-16	old	2581113		107
1998-04-23	new	2602448	755	116

Table 3: Statistics for live tests

have since evolved separately. Because caching tends to reduce the number of simultaneous network connections, during our trials we operated this software with caching disabled. This increases the load on the system, but for various reasons does not significantly increase response time as seen by the users.

Table 3 shows some statistics for each of the trials. The "Max. alloc. fds" column shows the largest number of file descriptors allocated to a single process at any one point during the trial; the "Peak req. rate" column shows the largest number of requests logged during a single second over the course of the day.

6.1 Effect of request rate on CPU load

The operating system maintains counts of the number of clock interrupts that occur in each system mode (user-mode, kernel-mode, and idle). During the course of each trial, we logged these counters every 15 minutes, which allowed us to reconstruct the mean time spent in each mode during the 15 minutes prior to each log entry. The proxy software creates a timestamped log entry for each HTTP request it receives, so we can also count the number of requests handled in each 15 minute period, and then compute the mean request rate over that period.

Figure 9 shows how CPU idle time, and CPU kernel-mode time, vary as a function of the mean request rate. Each point on the scatterplot represents one 15-minute sample. The circles correspond to idle time; the squares correspond to kernel-mode time. The filled marks show performance with the old versions of both select() and ufalloc() (the trial of 1998-04-16). The open marks show the performance of the new implementations (the trial of 1998-04-23).

We then computed linear regressions for each set of samples. The regression lines are shown in Figure 9; the numeric results are given in Table 4. (User-mode regressions are given in the table, but not shown in the figure.) Each sample set includes 96 points (24 hours of 15-minute samples). The correlation between kernel-mode time and request rate is quite close; the correlation for idle time is not quite as good, probably because of some outliers caused by daily "housekeeping" tasks done during periods of low request rate. Because the outliers all occur at low request rates (that is, late at night), we re-calculated the regressions after excluding samples taken at rates below 20 requests/second. These regressions, shown in Table 5, show higher correlation coefficients for idle time and user-mode time.

The regressions for idle time and kernel-mode time show significantly steeper slopes for the unmodified kernel, compared to those for the new implementations of select() and ufalloc(). The regressions for user-mode time suggest that the new kernel performs slightly better, perhaps because of better data-cache utilization, but the difference might not be significant.

Although one cannot necessarily expect linear behavior at very high request rates, a linear extrapolation of the idle time regressions from the full data sets gives X-intercepts of 58 requests/sec. for the unmodified kernel,

Date	Kernel version	CPU mode	Slope	Corr. coeff.
1998-04-16	old	idle	-1.67	-0.96
1998-04-23	new	idle	-1.34	-0.92
1998-04-16	old	kernel	1.09	0.98
1998-04-23	new	kernel	0.85	0.99
1998-04-16	old	user	0.58	0.77
1998-04-23	new	user	0.49	0.66

N = 96

Table 4: Linear regressions: full 1-day data sets

Date	Kernel version	CPU mode	Slope	Corr. coeff.
1998-04-16	old	idle	-1.69	-0.97
1998-04-23	new	idle	-1.46	-0.98
1998-04-16	old	kernel	1.02	0.96
1998-04-23	new	kernel	0.85	0.99
1998-04-16	old	user	0.68	0.97
1998-04-23	new	user	0.65	0.99

N = 54

Table 5: Linear regressions: above 20 requests/second

and 69 requests/sec. for the new implementation. Using the truncated data sets (Table 5), the calculated X-intercepts are 57 and 68 requests/sec., respectively. This suggests that the modified kernel might support a peak request rate about 19% higher than the unmodified kernel, in this application.

Note that our samples were averaged over 15-minute intervals. The actual one-second peak rates experienced during these trials (see Table 3) were 107 requests/sec. for the unmodified kernel, and 116 requests/sec. for the modified kernel. Clearly, the systems can support rates higher than the extrapolation of idle time implies. The main significance of our performance improvements may be not the increase in peak throughput, but the decrease in queueing delay (and response time) at high throughputs.

6.2 Profile results

We obtained CPU-time profiles, using DCPI, for the proxy server during periods of heavy load, for both the original kernel (Table 6) and our modified kernel (Table 7). Each profile covers a period of exactly one hour. The tables include all procedures accounting for at least 1% of the non-idle CPU time.

The first column in each profile shows the fraction of CPU time spent in each function or group of procedures.

CPU %	Non-idle CPU %	Procedure	Mode
10.77%		all idle time	kernel
89.23%	100.00%	all non-idle time	kernel
35.27%	39.53%	all select functions	kernel
13.51%	15.14%	selscan	kernel
12.56%	14.08%	soo_select	kernel
7.48%	8.38%	undo_scan	kernel
1.64%	1.83%	select	kernel
12.64%	14.17%	commSelect	user
1.74%	1.95%	all TCP functions	kernel
1.49%	1.67%	malloc-related #1	user
1.39%	1.56%	malloc-related #2	user
1.09%	1.22%	mutex_unblock	user
1.03%	1.16%	read_io_port	kernel
0.95%	1.07%	bcopy	kernel
0.94%	1.05%	memGrep	user

Profile on 1998-04-16 from 10:00 to 11:00 PDT
mean load = 54 requests/sec.
peak load ca. 98 requests/sec

Table 6: Profile of unmodified kernel on live proxy

As the first row in each table shows, even during periods of heavy load, some time is spent in the kernel's idle thread and its children. Therefore, the second column shows the fraction of non-idle CPU time spent in all non-idle procedures; this is a more useful basis for comparing the two kernels. Note that the profiles include a mixture of kernel-mode and user-mode procedures.

The modified kernel spends 30% of the non-idle CPU time in select() and related procedures, compared to almost 40% spent in such procedures by the unmodified kernel. However, kernel-mode select() processing is still a significant burden on the CPU. As in Figure 2, considerable time is spent in the user-mode commSelect() procedure (Squid and NetCache apparently use slightly different names for the same procedure). These observations support our belief that the bitmap-based select() programming interface leads to unnecessary work, and probably to significant capacity misses in the data caches.

In experiments with simulated loads, we observed that NetCache on our kernel calls select() about 7 times as it does on the unmodified kernel. We believe this is because our faster select() causes a NetCache thread to return from select() with usually only one ready descriptor[1]. Before the next event arrives, other NetCache threads call select() to discover this event again. In the unmodified kernel, each call to select() takes

[1] NetCache uses multiple event-driven threads, presumably for exploiting the parallelism available on SMP machines.

CPU %	Non-idle CPU %	Procedure	Mode
16.29%		all idle time	kernel
83.71%	100.00%	all non-idle time	kernel
25.11%	30.00%	all select functions	kernel
11.23%	13.42%	new_soo_select	kernel
7.73%	9.24%	new_selscan_one	kernel
5.67%	6.77%	select	kernel
0.04%	0.05%	new_undo_scan	kernel
15.33%	18.32%	commSelect	user
2.70%	3.23%	all TCP functions	kernel
2.56%	3.05%	in_pcblookup	kernel
1.09%	1.30%	mutex_unblock	user
1.01%	1.21%	bcopy	kernel
1.00%	1.19%	read_io_port	kernel
0.97%	1.16%	malloc-related #1	user
0.93%	1.12%	memGrep	user
0.91%	1.09%	malloc-related #2	user

Profile on 1998-04-23 from 10:00 to 11:00 PDT
mean load = 55 requests/sec.
peak load ca. 116 requests/sec

Table 7: Profile of modified kernel on live proxy

longer, and returns multiple events. This may account for the heavy use of select() in Table 7.

In this application, even the unmodified kernel spends very little time in ufalloc() (0.20%). However, the modified kernel spends even less time in ufalloc() (0.03%). For this proxy, the total number of open file descriptors is relatively small. However, one might expect this fraction to become more significant at higher request rates.

We are not entirely sure what caused the significant increase in time that the modified kernel spends in in_pcblookup. This may be the result of an unfortunate collision in the direct-mapped data caches.

We note that in this real-world environment, for both versions of the kernel, just over 1% of the non-idle CPU time is spent in all kernel-related data movement (the bcopy()). Even less time is spent computing checksums. A moderate amount of time (between 2% and 3%) is spent in TCP-related functions (which have been highly optimized in Digital UNIX). These measurements reinforce the emphasis placed by Kay and Pasquale[9] on "non-data touching processing overheads"; however, they failed to recognize that the poor scalability of select() would ultimately dominate the other costs.

6.3 Data cache effects

We have speculated in several places that our kernel modifications affect data cache utilization. DCPI allows us to estimate the mean cycles per instruction (CPI) for each procedure in a profile, and to estimate the fraction of dynamic stalls caused by data-cache misses. We found that the CPI for the user-mode commSelect() procedure declined from 1.69 to 1.62 as a result of our kernel changes, mostly because of fewer data-cache misses.

We also found that the CPI for in_pcblookup() increased from about 1.28 to 11.15 as an apparent result of our kernel changes, even though we did not change the code for this kernel procedure. This suggests that we somehow created a particularly unlucky collision in the data caches between the data structures for in_pcblookup() and those for select().

7 Related Work

Operating system researchers and vendors have devoted much effort to improving Internet server performance. One early experience that lead to published results was the 1994 California election server [14, 15]; another early study was performed at NCSA [12]. Operating system vendors responded to complaints of performance problems by improving various kernel mechanisms, especially by replacing BSD's linear-time PCB lookup algorithm [13, 21], and by changing certain kernel parameter values. Vendors also provided tuning guides for systems being used as Web servers [6].

In response to observations about the large context-switching overhead of process-per-connection servers, recent servers [5, 16, 22, 24, 25] have used event-driven architectures. Measurements of these servers under laboratory conditions indicate an order of magnitude performance improvement [5, 20].

Maltzahn et. al. [11] reported the poor performance of Squid under real conditions. Fox et al. [7], in describing the Inktomi system, also briefly mention that their event-driven front-ends spend 70% of their time in the kernel, and attribute this to the state-management overhead of a large number of simultaneous connections. However, neither of these papers analyzed the reason for this phenomenon in any detail.

8 Conclusion

We presented a detailed analysis of the effect of WAN delays on the performance of event-driven servers, and showed that linear scaling in the select() and ufalloc() implementations leads to excessive kernel CPU consumption.

We described scalable versions of select() and ufalloc(), and evaluated their impact on the performance of event-driven servers. We showed that these changes improve the performance of Web servers and proxies on realistic benchmarks, and on a live proxy, without harming performance on naive benchmarks.

Our results show the need for a new, scalable interface to replace select(). We are currently working to develop this.

Acknowledgments

We are grateful to Kathy Richardson of Digital's Network Systems Laboratory, for providing data on the performance of the Palo Alto Web proxies, and to Kathy and to Jessie Stickgold-Sarah for helping us evaluate our changes in the context of these proxies. We also thank the USENIX referees for their comments.

References

[1] J. Anderson, L. M. Berc, et al. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

[3] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Dec. 1997.

[4] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. To be presented at the Workshop on Internet Server Performance, June 1998.

[5] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[6] Digital UNIX Tuning Parameters for Web Servers. http://www.digital.com/info/internet/document/ias/tuning.html.

[7] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[8] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *1996 SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[9] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of the ACM Communications Architectures and Protocols Conference (SIGCOMM)*, pages 259–268, San Francisco, CA, Sept. 1993.

[10] A. Luotonen, H. F. Nielsen, and T. Berners-Lee. CERN httpd 3.0A. http://www.w3.org/pub/WWW/Daemon/, July 1996.

[11] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[12] R. E. McGrath. Performance of Several HTTP Demons on an HP 735 Workstation. http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html, Apr. 1995.

[13] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming tcp packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–280, Aug. 1993.

[14] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[15] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[16] Network Appliance, Inc., *NetCache*. http://www.netapp.com/level3/netcache/datasheet.html.

[17] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference. http://www.scriptics.com/people/john.ousterhout/threads.ps.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. Technical Report TR97-294, Rice University, CS Dept., Houston, TX, 1997.

[19] K. J. Richardson. Personal communication, 1997.

[20] S. E. Schechte and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/ stuart/Tarantula/FirstPaper.html.

[21] Solaris 2 TCP/IP. http://www.sun.com/sunsoft/solaris/networking/tcpip.html.

[22] Squid. http://squid.nlanr.net/Squid/.

[23] W. Stevens. *Unix Network Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[24] thttpd. http://www.acme.com/software/thttpd/.

[25] Zeus. http://www.zeus.co.uk/.