Critical Performance Factors in Web Server Design: Experience Implementing CoW, a Cooperative Multithreading Web Server*

Matthew D. Roper, Takashi Ishihara, and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, California 95616-8562 U.S.A
{roper, ishihara, olsson}@cs.ucdavis.edu

Abstract

The web is becoming an increasingly popular medium of information exchange and content distribution. Popular web sites commonly receive thousands of requests per second, so high performance web servers are in high demand. Many different HTTP server implementations have attempted to achieve high performance using varying techniques, but most of these implementations only address a single specific aspect of the web serving model. Little work has been done to enumerate and explore all of the factors that play a significant role in web server performance. In this paper, we explore the key design decisions that we have found to play a critical role in the performance of a web server. We also introduce a new thread-based server, CoW, that offers both higher performance than popular web servers as well as a simple, clean architecture.
Keywords: web server, concurrency models, event notification models, thread-based programming, cooperative multithreading

1 Introduction

HTTP servers are a widely expanding area of development. Although it is likely impossible to develop a single, perfect server for all platforms (including mainframes, workstations, general PCs, and embedded systems), there has been significant work in searching for a suitable model on which high performance web servers can be built. Most modern web servers fall into one of three categories: process-based servers (e.g., Apache 1.x [1]), event-driven servers (e.g., BOA [2]), and preemptive thread-based servers (e.g., Apache 2.x or Lancer [3]). Thread-based servers have become significantly more popular over the last couple of years now that most operating systems finally have stable thread implementations with high performance, but so far they have not been able to overtake event-driven servers in terms of web server performance.

In this paper, we identify three critical components of web server design that significantly affect server performance: concurrency model, event-notification model, and I/O strategy. We examine how various popular web servers address these issues and compare them in terms of performance and resource requirements. We also introduce our own web server, CoW (Cooperative Multithreading Web Server), which was designed with these factors in mind and outperforms the other popular servers in our experiment. CoW utilizes a cooperatively threaded architecture to maintain a clean and maintainable code structure while providing excellent performance.

*This work is partially supported by Z-World, Inc. and the University of California under the MICRO program. The National Science Foundation partially supported the computing equipment used in our work through grant EIA-0224469.

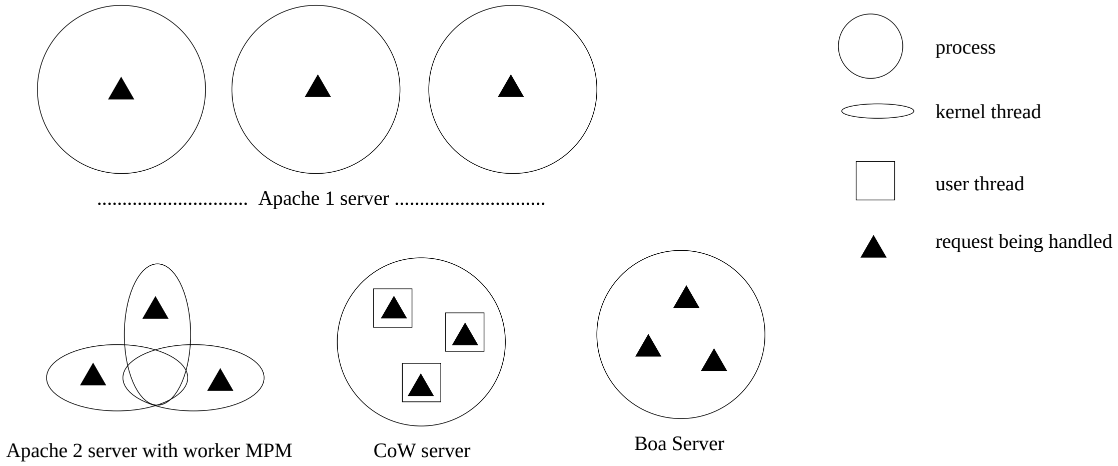

Figure 1: Concurrency models used by popular web servers

The rest of this paper is organized as follows. Section 2 presents design models for web servers, including concurrency models, event notification models, and I/O models. Section 3 explores different styles of multi-threading in more detail. Section 4 introduces the specific design and implementation of our CoW web server. Section 5 describes and analyzes the performance experiments we ran on CoW and the other web servers. Section 6 discusses the results of the preceding section. Finally, Section 7 concludes the paper.

2 HTTP Server Design Issues

Although concurrency models are the most visible and most heavily explored facet of HTTP server development, it is important to remember that they are still only a single piece of the web server performance puzzle. Achieving optimal web server performance also requires a high-performance I/O strategy and scalable event notification model.

2.1 Concurrency Models

There are three major categories of concurrency models for HTTP servers: process-based servers, single-process event-driven servers, and thread-based servers. Figure 1 illustrates the various models used by popular web servers.

2.1.1 Process-based Servers

A process-based server handles each incoming request in a separate process, either by forking off a new child process in response to each incoming request or by assigning the request to a process from a pre-forked pool of processes. This model is considered to be heavy-weight since process context switching is expensive. However process-based servers have the advantage of leveraging the operating system's scheduler, which not only simplifies implementation, but also allows closer integration of scheduling and I/O. An example of this type of server is the 1.x branch of the well-known Apache Web Server [1]. Apache's 2.x branch, although capable of supporting other concurrency models via Multi-Processing Modules (MPM's) [4], also uses a process-based model by default.

2.1.2 Single Process Event-driven Servers

Event-driven servers run as a single process and use non-blocking I/O and event notification mechanisms such as `select()` or `poll()` to internally multiplex control between active connections.

Because there is no context switching, even-driven servers are light-weight and perform very well with low memory and CPU usage, making them ideal for use on low-resource embedded systems. The primary limitation of these servers is that most operating systems limit the number of file descriptors available to a single process. Each active network connection uses up a file descriptor, and descriptors must also be used to open the web document files being served. Since event-driven servers run entirely inside one OS process, they will quickly exhaust their supply of file descriptors if the connection rate is too high. Another factor that can limit the performance of event-driven servers is the event notification model used; most event-driven servers use `select()`, which does not scale well over large numbers of connections (although it is the most portable mechanism). Event notification models are described more thoroughly in Section 2.2. BOA [2] is an example of a popular event-driven server.

2.1.3 Thread-based Servers

Thread-based servers have been considered for many years, but very few mature implementations existed until the last couple of years. The main problem that held thread-based servers back for so long was lack of stable OS support, but almost all modern operating systems now have good thread support. The thread-based server category can be further subdivided into servers based on preemptive multithreading and servers based on cooperative multithreading packages. Most existing multithreaded web servers, such as Apache 2.x (when using the `worker` MPM [5]) and Lancer [3], are based on *preemptive multithreading*: the thread scheduler can force a context switch at any time in the same way the operating system can force context switches between processes. An alternative model is *cooperative multithreading*: each thread has uninterrupted use of the processor until it invokes a function that passes control to another thread.

Thread-based servers are efficient in that the context switching between threads within a process is generally less expensive than context switching between processes. In addition, by using explicit yield operations, a server can efficiently pass execution control to a thread that is ready to run, thus minimizing the latency. Section 3 discusses more details about styles of threading.

2.1.4 Other Models

The SEDA model is a hybrid of the thread-based and event-driven models. The Haboob web server, built with the SEDA model, performs better [6] than Apache [1] and Flash [7]. SEDA, however, has a very different structure from other HTTP servers in that it is divided into stages based on functionality, and each stage is connected via pipeline. The requirement is that the application must be well-conditioned, meaning the behavior must resemble a simple pipeline [6]. We believe this conditioning is difficult to achieve in the general case because it is impossible to tell whether or not reading from the network socket is completed until the HTTP message header is parsed [8]. In addition, features such as caching and keep-alive require feedback to earlier stages of the pipeline and turn the control flow into a nested loop, rather than a simple pipeline.

2.2 Event Notification Models

The choice of event-notification model has a significant (and often underestimated) effect on web server performance. Event-notification models are necessary for servers that process more than one concurrent request per kernel-scheduled unit (i.e., servers that are event-driven or use user-space threading libraries such as Pth [9]). Event notification frameworks are not an issue for process-based servers or kernel thread-based servers since the operating system scheduler takes care of event notification and process/thread activation internally.

The most portable (and also most widely used) event notification mechanisms are `select()` and `poll()`; unfortunately both have serious scalability problems. `select()`, which is used by Boa and also internally by Pth, is a poor choice for servers that need to handle a large number of simultaneous connections:

- There is an upper-limit on the number of file descriptors that `select()` can monitor simultaneously; this limit is independent of the file descriptor limit that the operating system imposes on each process [10].

- Both the internal implementation of `select()` and the application-level loop necessary to parse the results scale linearly as the number of file descriptors being monitored increases. This results in very poor performance compared to other event notification mechanisms that scale in terms of the number of events that actually occurred.

`poll()`, another highly portable and relatively popular mechanism, also does not scale well — `poll()` requires large arrays of file descriptors to be copied between user-space memory and kernel-memory, which results in very poor performance [11].

Several newer event notification mechanisms (such as Linux 2.6's `epoll` and FreeBSD's `kqueue`) run in constant time, but unfortunately these mechanisms are very platform-specific and are unsuitable for servers that require portability. Fortunately, some frameworks, such as libevent [12] and enot [13] are being developed that wrap the functionality of all of these notification mechanisms and will automatically pick the best one available on any platform for which they are compiled.

Figures 2 and 3 illustrate the difference between `select()`-based application code and `epoll`-based application code. Error checking and setup are omitted for brevity. Note that the loop in the `epoll` code runs once for each event that actually occurred whereas the `select()` code must run once for every file descriptor being monitored.

```
select(numfds,
       &readfd_set,
       NULL, NULL, NULL);
for (i = 0; i < numfds; ++i)
    if (FD_ISSET(i, &readfd_set))
        process_event(i);
```

Figure 2: `select()`-based code

```
numev = epoll_wait(epfd, &events,
                   numfds, -1);
for (i = 0; i < numev; ++i)
   process_event(events[i].data.fd);
```

Figure 3: `epoll`-based code

2.3 Input/Output Models

A third critical factor for web server performance is how I/O is handled. I/O handling strategies raise two important questions:

- What OS mechanisms are used to read and write web documents? How many memory copies are involved in each?

- How is network transmission handled? Is application-level buffering performed to minimize network packets or system calls?

Copying data between memory buffers is a relatively expensive, yet widely overlooked, source of overhead for web servers. Opening a file, reading into a user-space memory buffer, and writing to a network socket requires multiple (unnecessary) copies of data around memory. One way to overcome part of this overhead is to use the `mmap` system call to map the file directly into user-space memory; this eliminates the intermediate buffer and one level of copying. Several operating systems also provide special system calls (such as Linux's `sendfile()`) that, given a file descriptor and a network socket, will perform a zero-copy transmission of the file contents over the network — this provides optimum performance in terms of memory transfers.

Another important I/O factor is whether the web server performs any application-level buffering. Although such buffering actually increases the number of memory copies performed while serving an HTTP request, it can cut down on network costs and ultimately result in better performance than an

I/O strategy based on zero-copy mechanisms. Buffering is useful both for writing to and reading from network sockets. When writing, a buffering output layer can collect several small writes into a single large write, which the OS will then transmit as a single TCP packet. This reduces packetization costs in the kernel as well as general network congestion. Although operating system kernels will generally try to prevent very small packets from being transmitted (via the Nagle algorithm), this method is not as effective as an application-level buffer that understands the semantics of the web server; in fact, the Nagle algorithm has actually been found to damage performance in web servers and other similar applications since it can force unnecessary delays [14]. Buffering is also useful for processing input. Although the operating system will buffer unread socket data, using large read buffers and application-level buffers can avoid extra calls to `read()`; since system calls are more expensive than regular function calls, this can improve performance if several small reads need to be performed.

3 Threading Implementations

Many different thread designs and implementations are possible based on variables such as kernel support, mapping of threads onto processes, and use of machine-specific assembly language vs. portable C. Kernel-space threads, also called light-weight processes, use a preemptive model for context switches in which the application is not aware of thread context switch. One advantage of kernel-space threads is that the operating system is aware of the individual threads so only a single thread will be blocked when synchronous I/O system calls (such as `read()` and `write()`) are made. In contrast, user-space threads have their own scheduler in which the kernel is not aware of context switches. Because user-space threads are not visible to the operating system, any call to a synchronous I/O system call will block all threads of execution in the process; to alleviate this problem, most user-space threading libraries provide replacement functions that provide the semantics of blocking I/O system calls but are implemented using non-blocking I/O routines.

Another feature that distinguishes threading models is the mapping between kernel-space threads and user-space threads. These mappings can be represented as M:N, where M is the number of user-space threads, and N is the number of kernel-space threads (or light-weight processes). Solaris, AIX, Tru64 Unix, and IRIX are M:N; Pth is M:1; and LinuxThreads is 1:1 [15, 9].

3.1 GNU Pth (Portable Threads)

Pth is a user-space POSIX/ANSI-C threading library with non-preemptive thread scheduling [15, 9]. Because Pth has no assembler code, it is highly portable but lacks platform-specific fine-tuning available in some other threading libraries. Concurrency is achieved internally by using non-blocking I/O, which is called low concurrency, as opposed to high or true concurrency which requires multiple processors. Pth provides cooperative multithreading (described in Section 2.1.3). In addition to functions like `pth_yield()` and `pth_wait()`, which are used to explicitly yield to other threads, Pth also yields automatically anytime a thread must block on I/O (i.e., when a thread calls a function such as `pth_read()` or `pth_write()`). Such functions generally return control to the Pth scheduler, which will pick the next thread to run based on thread priority and position in the ready queue. As an alternative, a thread may explicitly name the thread that it wants to run next by passing that thread's ID as a parameter to the `pth_yield()` function; this is known as a named yield (a general, unnamed yield is also possible by passing NULL).

When the scheduler takes over the execution control, the `select()` system call is used to detect if any I/O is ready and then control is passed to one of threads associated with a ready I/O stream. The Pth scheduler also handles signals and intra-application message passing at this point. Because Pth provides a Pthread API, platforms that do not support POSIX threads can use it, taking advantage of the high portability of Pth. The GNU project claims that at version 1.1, Pth was already a more complete implementation of the Pthread API than FreeBSD's native implementation [16]. Because of non-preemptive scheduling, Pth requires that application code be written with threadsafe functions, but does not require reentrant functions, a great benefit when working with 3rd party libraries. A thread context switch is achieved by either a `makecontext(3)/switchcontext(3)` pair or a `sigsetjmp(3)/siglongjmp(3)` pair [15]. One shortcoming of the Pth library is that the user-space nature of its threads prevents utilization of multiprocessors; this shortcoming was addressed in the NGPT library (see Section 3.3), which was based on Pth.

Pth can be viewed as a framework for abstracting the details of event-driven applications (Section 2.1.2); the threads provided by Pth are an abstraction of the state machines present in regular event-driven applications. A major advantage of using Pth is that application programmers do not need to write their own scheduling code when developing event-driven applications. This can significantly speed-up development time by eliminating the coding and debugging of a custom scheduler. Furthermore, the source code of an event-driven application written with Pth is expected to be more readable than that of traditional event-based applications, which tend to have scheduling code scattered everywhere.

3.2 LinuxThreads

POSIX threads on Linux are implemented using processes. By using processes, LinuxThreads is able to easily implement most of the requirements of the POSIX thread standard. Linux's kernel-level threads are created by the `clone()` system call, which, like `fork()`, creates a new process. Processes created with `clone()` have a unique process ID and are scheduled directly by the kernel, but differ from processes created by `fork()` in that they share part of their execution context (memory, file descriptors, and signal handlers) with the parent process. LinuxThreads includes user-space wrapper functions (in glibc) around the kernel-level `clone()` function to provide the POSIX thread interface.

Competing POSIX threads implementations for Linux were replaced by LinuxThreads mainly because of its superior performance; in other words, although process based, LinuxThreads has optimized the use of processes as threads to mitigate the negative aspects of using processes, such as expensive process context switching. Because of its kernel-level implementation, LinuxThreads can take advantage of multiprocessor architectures. Its other strengths include robustness and simple library code. The ability to use blocking system calls enhances the robustness of LinuxThreads; since it is process based, using blocking I/O does not cause other threads of a program to block. The latest versions of LinuxThreads are closely coupled with libc 6 (also known as glibc 2) for better support than previous versions in terms of performance and robustness [17].

3.3 Next Generation POSIX Threads (NGPT) and Native POSIX Thread Library (NPTL)

During the development of Linux 2.5, kernel developers began looking for ways of improving thread performance under Linux. Although LinuxThreads worked moderately well for regular system loads, it did not scale well when used with applications that used hundreds or thousands of threads. Furthermore, LinuxThreads had some problems with POSIX compliance, especially in terms of signal handling. Two new threading libraries, Next Generation POSIX Threads (NGPT) [18] and Native POSIX Thread Library (NPTL) [19], arose to address these problems.

NGPT was the first to become available and receive exposure on the Linux kernel mailing list. NGPT, which is based on an M:N threading model, was a radical change to the kernel's thread support. By switching to an M:N model, NGPT developers had to split thread scheduling code between kernel code and user-space library code. Pth was used as the base for NGPT and was extended to communicate with the kernel scheduler. Use of a user-space scheduler allows NGPT to create and switch between user space threads without having to make any (relatively expensive) switches between user-space and kernel-space. NGPT achieved its goal of improving Linux thread performance and consistently outperformed the original LinuxThreads implementation of POSIX threads by a factor of two [20].

Shortly after NGPT became available, the first version of NPTL was released. NPTL was primarily the work of two kernel developers who felt that NGPT's departure from the 1:1 threading model was unnecessary. These developers argued that switching to an M:N model would require re-implementing many complicated features (e.g., real-time scheduling guarantees) in user-space that had been present in the kernel scheduler for a long time. An M:N scheduler would be significantly more complicated and very tricky to maintain because changes would have to be made to both halves of the scheduler. Furthermore, many user-space tools such as debuggers and profilers would have to be updated to work properly. The NPTL developers argued that pure user-space context switches are relatively rare; most context switches are induced by the kernel so an M:N thread model is likely to add more overhead than it alleviates. Recent benchmarks have shown NPTL to perform about four times better than NGPT, or about eight times better than traditional LinuxThreads [20]. NPTL is expected to be the official threading library for Linux when kernel 2.6 is released.

4 CoW's Design

CoW is an HTTP server based on a cooperative multithreading (CM) model and implemented using a slightly modified version of Pth [9]. CoW uses a thread-per-request architecture in which each incoming HTTP request is paired up with a handler from a pool of idle threads.

CoW consists of three types of threads: the accept thread, the depot thread, and handler threads. The accept thread, or main thread, is responsible for spawning the depot thread and a pool of handler threads. After this initial setup, it goes into the accept loop where it will pass incoming connections to the depot where they will be routed to a handler thread. The depot thread coordinates between the accept thread and the handlers via message passing. It is responsible for dispatching incoming connections delivered by the accept thread to an available handler thread for processing. Each handler thread processes a single client connection to completion (which may span multiple requests if an HTTP keep-alive connection is used) and then returns to the thread pool. Figure 4 gives a pictorial overview of CoW.

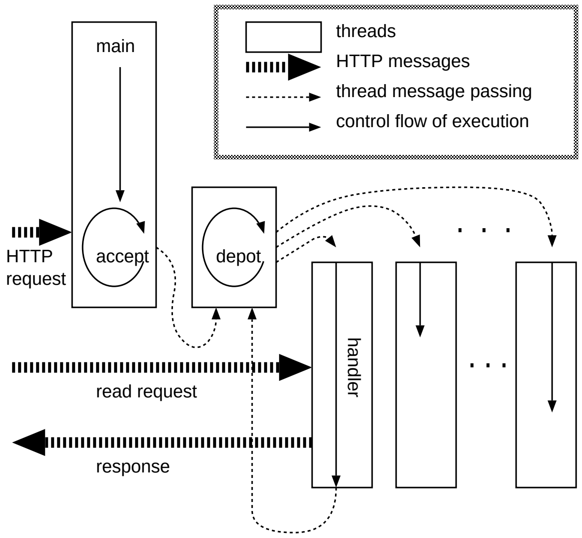

Figure 4: Overview of CoW's internal structure

Inter-thread communication in CoW is accomplished via Pth message ports. Message ports function as FIFO queues of messages and are manipulated with the asynchronous commands `pth_msgport_put()` and `pth_msgport_get()` to send and receive messages respectively. Pth messages are defined by the structure `pth_msg_t`, but additional application-specific information may be added by simply defining a new structure that contains a `pth_msg_t` as the first field and has other information following; a pointer to such a structure can be cast to type `pth_msg_t*` and then sent using the regular message port commands. CoW uses three different types of messages:

- Messages from the accept thread to the depot thread. These contain a file descriptor associated with an incoming request; after finding an available handler, this message will be copied and dispatched to the handler.
- Messages from the depot thread to a handler thread. These are simply the "accept to depot" messages being forwarded on to the request's handler thread. They provide the handler with the file descriptor to the network connection to be processed.
- Messages from a handler thread to the depot thread. These messages indicate that the handler is finished processing a request and is ready to handle a new one. They contain a reference to the handler's message port, which the depot can use to dispatch new requests to the handler. If the depot thread receives one of these messages when there are no outstanding requests, it will place the message port on a queue (so that it can be assigned to a future request) and then suspend the thread. This is an additional benefit of including a depot thread in our implementation – Pth does not allow handler threads to suspend themselves.

In order to improve performance and avoid dynamic memory allocation, CoW uses a pre-allocated pool of message structures for message passing between threads. When a new connection is established, the accept thread will remove a message structure from the pool and send it to the depot, where it will be dispatched to the first available handler. The message structure is returned to the pool as soon as the handler has extracted the important information (i.e., the file descriptor for the socket connection). If the accept thread detects that the pool is empty, it will repeatedly yield until some message structures are freed up; no new connections will be accepted during this time. While this may cause some clients' connections to be rejected, we do not see this as a problem; the limited (but configurable) number of handler threads already limits the maximum concurrency of the server and not accepting additional connections avoids allocating resources to connections that will probably timeout anyway.

One problem we encountered with using Pth's message ports was that if the message queue is processed in FIFO order, the server could suffer from response timeout or run out of file descriptors under heavy loads. The problem arose from the operating system's limit on the number of file descriptors available to a process (as described in Section 2.1.2). There were two troublesome situations. Suppose the message queue is filled by "new connection" messages from the accept thread, and all handlers are busy except for one handler that just finished its job. In this case the depot will not know the availability of that handler until processing all pending messages. Thus, it is important to give higher priority to the messages from the handler. Also, given the previous situation, further suppose the handler has not yet opened the requested file. If the new network connections waiting to be dispatched use up all of the available file descriptors, already-running handlers will fail to serve the request because `open()` fails to allocate another descriptor. To overcome this problem, we added a `pth_msgport_push()` function to Pth that prepends a message to the message port (like a stack) rather than append it (like a queue). Since Pth message ports are internally implemented as circular linked lists, both appending and prepending are constant time operations so this change incurs no performance penalty. We used this new function to send "handler done" messages to the depot so that they always appear at the beginning of the message port and are processed first.

Another difficult implementation decision we had to make was whether to continually reuse a pool of pre-spawned threads to handle requests or whether to spawn a new thread for each incoming request. Creating a new stack for a thread is expensive because it uses malloc, which makes the thread pool approach more attractive. The downside of this strategy is that the Pth scheduler performs a linear scan of its internal queues (i.e., walks through the wait queue to find out if any threads are runnable and, if so, moves them to the ready queue), which introduces unnecessary overhead if there are a large number of idle threads on the wait queue. Fortunately, Pth also provides `pth_suspend()`, which moves specified threads to a separate suspend queue. The Pth scheduler ignores threads on that queue, so suspending unused handlers eliminates the scheduling overhead mentioned above and makes a thread pool the best design model.

Although Pth is built on top of the inefficient `select()` event notification model, we modified the implementation of Pth to use Linux's new `sys_epoll` notification system as an alternative to `select()` (see Section 2.2). This modification required two changes to Pth. First, I/O functions such as `pth_write()` and `pth_read()` had to be modified to update the global epoll "interesting event set." Second, the Pth scheduler had to be updated to perform an `epoll_wait()` call on the interest set instead of calling `select()`. We tested a version of CoW running on this modified Pth and found performance (as measured by the number of successful responses) to increase by about 10% when running under a heavy load; performance under light load was unaffected.

When we first implemented CoW, we did not realize the importance of application-level I/O buffering and found CoW to be performing significantly worse than Apache. Our network traces indicated that the first bytes of CoW's response arrived much more quickly than Apache's, but the overall transmission time was far greater for CoW. We replaced CoW's existing I/O code (which used either `mmap()` for single copy I/O or, if available, zero-copy `sendfile()`) with an adapted version of Apache's I/O buffering code. We found that the performance gained at the networking layer more than made up for the additional intra-memory copies resulting from this code replacement.

Due to the user-space nature of its threading, Pth faces the same file descriptor exhaustion problem that single process event-driven servers face (as described in Section 2.1.2). To work around this problem, we added a command line option to CoW, which allows it to fork a small number of identical processes immediately after the main server socket is opened. Because the operating system will dispatch incoming requests to only one of the identical server processes, this strategy allows us to multiply the number of file descriptors available to CoW. Forking one additional, identical process, for example, effectively doubles the number of file descriptors that can be used for serving requests. The only disadvantages of this strategy are the increased memory usage (each process has its own set of threads taking up memory) and that a unified log file cannot be used without inter-process synchronization (although each process can log to a different file). This method of expanding file descriptors can also be used to make CoW scale to multi-processor systems since the kernel-scheduled threads can be scheduled on separate processors, unlike the user-space threads they contain.

5 Performance Results

Although we modified the version of Pth used by CoW to support `epoll` as described in Section 4, kernel-level support for `epoll` is only present in 2.6 kernel pre-releases or patched 2.4 kernels. We did not have root access on our final benchmark machines to make these kernel modifications, so we used the version of CoW built with a `select()`-based version of Pth. We set CoW's thread pool size to 1024 for these tests because we found this setting allowed us to scale better at higher connection rates and incurred no penalty at lower rates. Also, three additional processes were forked at startup to prevent file descriptor depletion.

We compared this CoW prototype to four other web servers: Apache 1.3.27 [1], Apache 2.0.47 [1] (using the thread-based "worker" multiprocessing module (MPM)), BOA 0.94.13 [2], and Xitami 24d9 [21]. In order to measure the performance of each server, we used httperf [22], a rate-based benchmark. Rate-based benchmarks more closely model real-world workloads because they make it possible to measure the snowballing effect of request backlog on server performance [23]. Although the numbers presented here represent a single set of test runs, we performed benchmarks on all servers hundreds of times over the course of a few months while we were tuning our own CoW server and the server performance trends remained constant. For our tests, we had all clients request the same file (of size 12500 bytes) repeatedly — this ensured that the file was buffered in the operating system's page cache and that we were measuring actual web server performance rather than hard drive performance.[1] All requests were performed using the HTTP/1.1 protocol and exactly five requests were performed on each persistent connection (so total requests is 5 times

[1] We initially tested with a large test set of over 7500 files of varying sizes and found almost all servers to perform

Table 1: Successful responses over 10 minutes

Connections per second	Number of Successful Replies over 10 minutes CoW	BOA	Apache 1	Apache 2	Xitami	Ideal
10	29680	29995	29938	29965	29764	30000
25	74277	74166	74131	74595	74578	75000
50	124960	117970	83545	84973	-	150000
60	125750	129725	84067	84405	-	180000
70	133468	133278	81759	78375	-	210000
75	126626	128804	82045	84420	-	225000
80	138333	130757	81825	75181	-	240000
90	148123	128835	81499	77330	-	270000
100	140899	125752	81351	75842	-	300000
110	139043	127189	82092	74100	-	330000
120	137796	126767	77936	76442	-	360000

Table 2: Web servers' average response times

Connections per second	Average Response Times (ms) CoW	BOA	Apache 1	Apache 2	Xitami
10	164	87	99	78	199
25	230	316	357	301	360
50	230	1849	1825	1711	-
60	2237	2275	1758	1700	-
70	2697	2858	1773	1707	-
75	2774	3127	1810	1679	-
80	3055	3304	1801	1719	-
90	3133	3404	1740	1745	-
100	3568	3544	1781	1797	-
110	4149	3464	1828	1810	-
120	4493	3507	2018	1659	-

number of connections). We benchmarked each server for 10 minutes at varying connection rates and measured the number of responses returned. As described in [22], this long benchmark period is necessary for the server to reach its steady state. We only performed our benchmark on static content.

We used 11 identical machines (one server and ten clients) to perform our tests. All machines had Intel Pentium 4 2.0 GHz processors, 512 MB main memory, and an Intel 8255x-based Ethernet adaptor. Linux kernel 2.4.20 i686 (as distributed by Redhat[2]) was used on all machines and the e100 kernel module was used as a network driver. Benchmark files were served from a local IDE hard drive with DMA enabled. Logging facilities and all dynamic content options were disabled for all servers. Tables 1 and 2 show the performance results of the various web servers. Figures 5 and 6 present graphs of this data. Response time is measured as the amount of time between transmission of a request to the server and when the final byte of the response finally arrives at the client.

We originally planned to include results for the Lancer HTTP server [3], a multi-threaded

identically — the I/O overhead of the slow IDE drives in our test system significantly outweighed any performance gains of the individual servers.

[2]The kernel shipped with Redhat 9 includes backports of several 2.6 kernel features (such as NPTL support) that are not present in the stock 2.4.20 Linux kernel. However, informal tests performed on a Debian Linux machine running a stock 2.4.20 kernel indicated that these Redhat-specific features did not significantly influence the relative performance results.

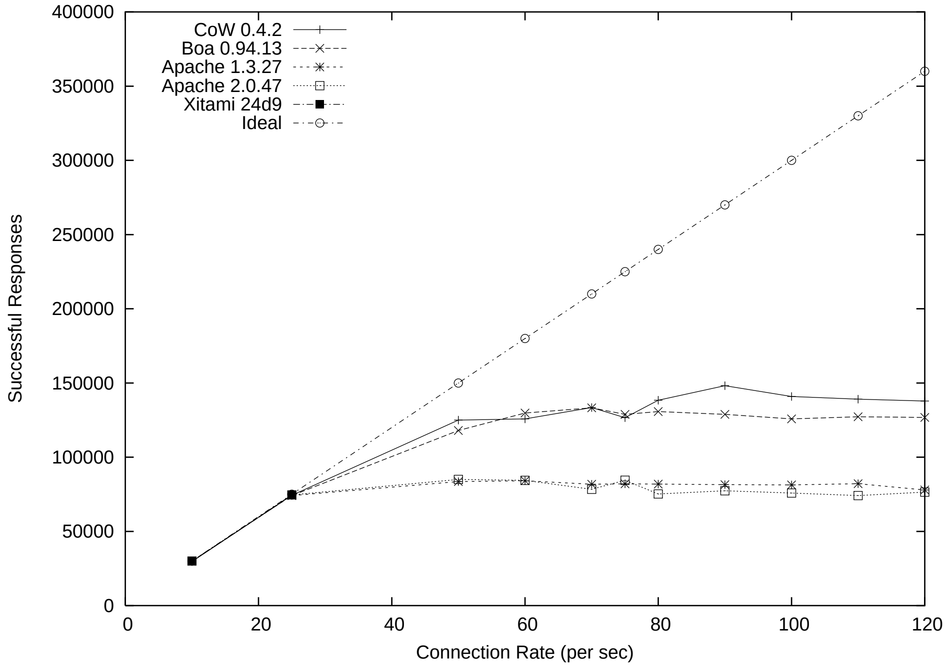

Figure 5: Successful responses over 10 minutes

webserver based on Pthreads. Our hope was that including Lancer would help illustrate just how much performance Apache 2.x (also based on Pthreads) sacrifices in exchange for its highly extensible framework. Unfortunately, Lancer does not currently support HTTP/1.1 keep-alive connections; although Lancer seemed to perform decently when faced with several isolated web requests, it was unable to scale with the other servers that implemented the keep-alive mechanism so we decided to drop it from our results.

We also planned to include benchmark results for the Haboob webserver based on the SEDA framework. Unfortunately Haboob did not perform as well as we expected based on Reference [6], which showed that Haboob outperformed the other popular web servers. We found the latest version of Haboob to be somewhat buggy and no longer under development. Even after fixing some bugs to prevent Haboob from crashing with our benchmark, the performance was still disappointingly low and we decided to drop the results from this paper.

6 Discussion

As the results in Section 5 show, CoW and Boa were the leading performers in terms of number of responses returned among the servers we tested; both version of Apache were noticeably worse. We were surprised to discover that Xitami, a multi-threaded server that uses a custom cooperative multithreading library called SMT [24], would display errors about unsafe memory assertions failing and would dump core when faced with higher connection rates. This is unfortunate because Xitami performed well at low connection rates and it would have been interesting to compare competing user-space threading implementations such as Pth and SMT.

CoW and Boa met our expectations and delivered the best performance of the servers tested. We believe that the excellent performance can be attributed mainly to the light-weight, single process concurrency models they employ since all servers tested utilized at least simple application-level buffering. Furthermore, event-notification mechanisms are not an issue for process-based servers or

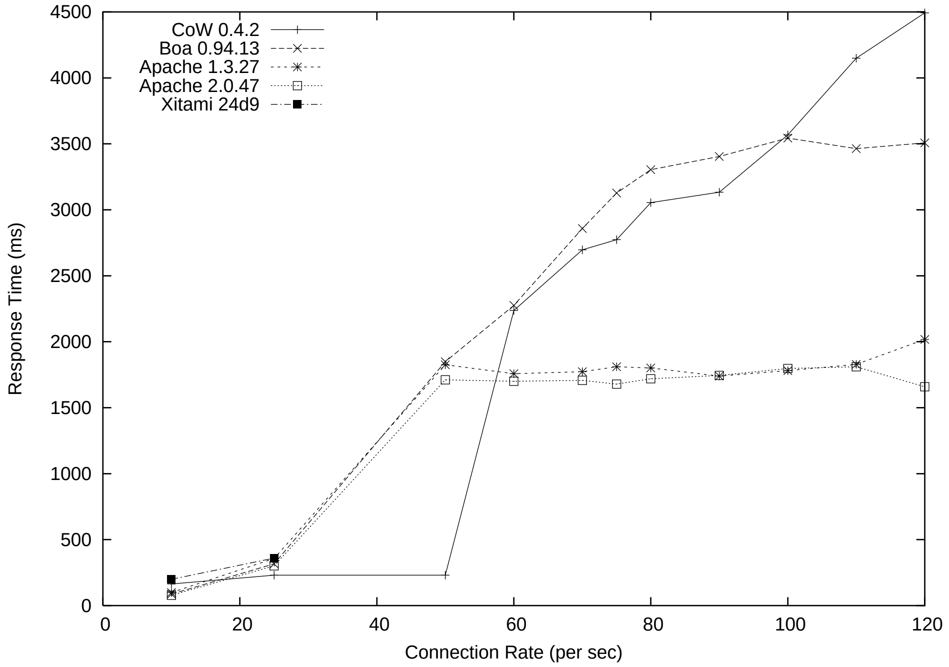

Figure 6: Web servers' average response time

kernel thread-based servers since the kernel's concurrency handling is already directly integrated with event notification. We have already observed that CoW can provide even better performance when running on top of an `epoll`-based version of Pth, and it is likely that similar performance improvements could be achieved by adding `epoll` support to Boa. Although CoW and Boa manage to return more responses than the Apache servers, Figure 6 shows that their average response time is significantly higher under heavy loads. The graph shows that Apache's response time levels off relatively quickly while CoW and Boa's average response times continue to grow as the server load increases. We believe that this behavior is caused by the use of `select()` by these servers. The quicker requests are sent to the server, the more file descriptors will be open at any given time; increasing the number of monitored file descriptors increases the overhead of the `select()` call and thereby increases the average response time. Boa's response time graph does level off at the very highest connection rates, but we suspect that this is the point where Boa has exhausted all of its available file descriptors and the overhead of `select()` cannot grow any further; since CoW uses the forking technique described in Section 4 to multiply the number of available file descriptors, it is able to maintain more open file descriptors and its response time continues to deteriorate. Since Apache only monitors a single network file descriptor per process, it does not need to use `select()` and does not face this penalty at higher connection rates.

Although the threading abstraction provided by Pth should make CoW slightly heavier weight than Boa, CoW manages to outperform Boa at higher request rates. We believe this is because CoW uses a relatively complicated buffering library (adapted from Apache 1.x) that works very hard to provide effective buffering with minimal memory copies; Boa's buffering code is much simpler and not as heavily optimized. For example, Apache's I/O buffering code can use functions such as `writev` to combine chunks of data from multiple memory locations in order to avoid unnecessary memory copying.

One important point in Boa's favor is that it uses even less resources than CoW while providing almost comparable performance. This is largely due to the fact that CoW spawns a pool of threads at startup and places them on the suspend queue; although the presence of suspended threads does not affect the scheduling algorithm at all, each thread still requires a fixed amount of memory to store its state. If memory is limited, CoW could easily be modified to spawn new threads only when requests arrive; however, spawning threads requires relatively slow `malloc()` calls which would add to the response latency.

One disadvantage of using a cooperative multithreading model for CoW is that serving dynamic content becomes more complicated. Servers like Apache are able to integrate interpreters for scripting languages directly into the web server as modules. This is easy because the preemptive multitasking nature of these servers allows web serving to continue even if a script being used to process one request gets into a runaway loop. However, in a CM model, such a script would never yield control of the processor back to the other threads of the server and all web serving would halt. To overcome this problem, it is necessary to execute each script handler in a separate kernel-level process or thread in order to separate normal web server execution from potentially unsafe user scripts. This type of model has been used for many years in other web servers to implement CGI scripts, but is generally avoided now because it introduces several extra sources of latency (forking, inter-process communication, etc.) that are not present when the dynamic content generator is incorporated directly into the main web server process.

Apache's primary goal has never been performance, but rather robustness. Apache also strives for security, flexibility, and extensibility which makes it ideal for large, resource-rich servers that need to use add-on modules. Because Apache 2 supports the pthread API (with the "worker" MPM), we tested it with both the LinuxThreads [17] and Pth [9] implementations of the API. Both versions compiled and ran without problem (although the Apache build system had to be modified slightly to build with Pth instead of LinuxThreads), but the performance of the two versions was nearly identical. This implies that the latency of Apache's general framework overshadows the effects of kernel-based versus userspace-based thread scheduling.

7 Conclusion

We have exposed the key design decisions that influence web server performance, namely concurrency models, event-notification models, and I/O strategies. We have also shown that cooperatively multi-threaded servers can provide comparable or better performance than purely event-driven servers, which are generally regarded as the optimal web server model. The use of cooperative multi-threading is advantageous because not only does it provide high performance, it also results in more readable and maintainable code. This is largely due to the fact that handler thread logic can be written as an individual unit. Synchronization mechanisms such as semaphores are unnecessary in cooperatively threaded code and it is easier to separate application logic from I/O details.

Another advantage of cooperative multi-threading for web server design is that scheduling code is provided by the thread API and does not need to be developed from scratch. Event-driven servers must develop their own scheduling mechanisms, which is both a time-consuming and error-prone process. In general, cooperative multi-threading provides the ease of development of multi-process code with performance comparable to that of event-driven servers.

A copy of our CoW web server for Unix-like systems can be downloaded from `http://www.cs.ucdavis.edu/~roper/cow`. We are also developing an embedded version of CoW for Z-World's [25] 8-bit Rabbit devices. Rabbits are often used as controllers for mechanical systems (e.g., manufacturing plants), so embedded web servers of interest as a means of monitoring and controlling such systems. An overview of our embedded CoW server can be found in [26].

Acknowledgments

We thank Aaron W. Keen, Justin T. Maris, Eric Wohlstadter, Fritz J. Barnes, Gene Fodor, and Pedram Abolghasem for providing us with helpful suggestions.

References

[1] The Apache web server, 2002. http://www.apache.org/.

[2] BOA web server, 2002. http://www.boa.org/.

[3] Lancer web server. http://sourceforge.net/projects/lancer.

[4] Multi-processing modules (MPM's). http://httpd.apache.org/docs-2.0/mpm.html.

[5] Apache MPM worker. http://httpd.apache.org/docs-2.0/mod/worker.html.

[6] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. *Symposium on Operating Systems Princples (SOSP-01)*, pages 230–243, October 2001.

[7] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server, 1999. http://www.cs.rice.edu/~vivek/flash99/.

[8] R. Fielding, J. Gettys, J. Mogul, L. Masinter, P. Leach, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1, June 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[9] Ralf S. Engelschall. GNU Pth — the GNU portable threads, January 2002. http://www.gnu.org/software/pth/.

[10] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Annual Technical Conference*, Monterey, California, June 1999. USENIX.

[11] Felix von Leitner. Scalable network programming; or: The quest for a good web server (that survives slashdot), October 2003. http://bulk.fefe.de/scalable-networking.pdf.

[12] libevent - an event notification library. http://www.monkey.org/~provos/libevent/.

[13] Event notification framework. http://www.fefe.de/fnord/enot.html.

[14] G. Minshall, Y. Saito, J. Mogul, and B. Verghese. Application performance pitfalls and TCP's Nagle algorithm, 1999. http://citeseer.nj.nec.com/minshall99application.html.

[15] Ralf S. Engelschall. Portable multithreading — the signal stack trick for user-space thread creation. In *Annual Technical Conference*, San Diego, California, June 2000. USENIX.

[16] Georg C. F. Greve. Brave GNU world. Technical report, 1999. http://www.gnu.org/brave-gnu-world/issue-7.en.html.

[17] S. Walton. *LinuxThreads*, 1997. http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/.

[18] Next generation POSIX threading. http://oss.software.ibm.com/developerworks/oss/pthreads.

[19] Ulrich Drepper and Ingo Molnar. The native POSIX thread library for linux. http://people.redhat.com/drepper/nptl-design.pdf, 2003.

[20] Ulrich Drepper. first NPT vs. NGPT vs. LinuxThreads benchmark results. http://lwn.net/Articles/10741/.

[21] Xitami web server. http://www.imatix.com/html/xitami/index.htm.

[22] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998. http://www.hpl.hp.com/personal/David_Mosberger/httperf.ps.

[23] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. *Symposium on Internet Technologies and Systems (USITES 97)*, pages 66–71, 1997.

[24] The SMT kernel. http://www.imatix.com/html/smt/.

[25] Z-World. http://www.zworld.com.

[26] Matthew D. Roper, Takashi Ishihara, and Ronald A. Olsson. CoW: an embedded, cooperative multithreading web server. In *2003 UC Davis Student Workshop on Computing*, pages 36–37, November 2003. http://www.cs.ucdavis.edu/~roper/papers/swc2003.pdf.