CoW: A Cooperative Multithreading Web Server*

Takashi Ishihara, Aaron W. Keen, Justin T. Maris, Eric Wohlstadter, and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, California 95616-8562 U.S.A
{ishihara, keen, maris, wohlstad, olsson}@cs.ucdavis.edu

Abstract

HTTP servers have evolved over the last decade. The use of threads within HTTP servers has been suggested, mainly for performance gain due to less overhead than the use of processes. However, major players have mostly avoided using threads in their servers partially because thread programming is not as easy as commonly believed and good debugging facilities are lacking. We introduce a new web server to explore the possible benefits of using a thread-based server. Our server offers a cleaner architecture than some servers as well as better performance than popular web servers.

Keywords: cooperative multithreading, web server, performance, thread-based programming

1 Introduction

Many HTTP servers are available. Yet, many HTTP servers are still in development and the demand for HTTP servers for new platforms continues. Diversity of platforms, including mainframes, workstations, general PCs, and embedded systems — which covers PDAs to wireless phones — prevents us from offering an ideal web server for all platforms [1, 2]. There may not be a panacea, but the use of threads in HTTP servers has been suggested as having two potential advantages. First, the use of threads can reduce execution costs, as context switching among threads is often less expensive than context switching among processes. Second, the use of threads can be a better match for embedded systems with limited resources. HTTP servers for embedded systems are often written in-house, and there are no widely available HTTP servers with full support of threads. (The exception is Apache's new version (2.x series), which will support the use of threads; however, its beta version was just announced [3].)

Most HTTP servers (e.g., Apache 1.x [3]) spawn a process for each incoming request. Other, event-driven servers (e.g., BOA [4]) effect concurrent handling of requests by internally maintaining state information on each request [4, 1]. In this paper, we compare the different models of HTTP servers in performance and resource requirements. We then discuss various thread-based web servers and consider the impact of different thread scheduling mechanisms. This paper introduces CoW (<u>Co</u>operative Multithreading <u>W</u>eb Server). Our goal is to achieve better performance than commonly used web servers while maintaining the code clean by taking advantage of thread architecture.

The rest of this paper is organized as follows. Section 2 presents design and implementation alternatives in thread packages and event-based model. Section 3 describes existing web servers. Section 4 introduces the 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. Finally, Section 6 concludes the paper.

2 Thread Background

Many different thread designs and implementations are possible based on kernel support, mapping (of threads onto processes), use of assembly language, and other factors. Kernel-space threads are usually called light-weight processes, and the application is not aware of thread context switch. On the other hand, user-space threads have their own scheduler in which the kernel is not aware of context switches. Mapping of thread models is shown 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). **M:N** model is found in Solaris, AIX, Tru64 Unix, and IRIX. GNU Pth is **M:1** model and LinuxThreads is **1:1** [5, 6].

*This work is partially supported by Z-World, Inc. and the University of California under the MICRO program.

2.1 GNU Pth

Pth (portable thread) is a user-space POSIX/ANSI-C threading with non-preemptive thread scheduling[5, 6]. Because Pth has no assembler code, its portability is high at the price of fine tuning for each platform. Concurrency is achieved by using non-blocking I/O, which is called low concurrency as opposed to high or true concurrency, which requires multiple processors. Pth uses cooperative multithreading, which means each thread must explicitly yield the execution control to other threads or the Pth thread scheduler. The former is called named yield, and it is done by `pth_yield(target_thread_id)`. The latter is the general case, and `pth_yield(NULL)` is called. When the scheduler takes over the execution control, the `select()` system call is used to detect if any I/O is ready, then the control is passed to the thread with one of thread associated with the ready I/O. The Pth scheduler also handles signals at this point. Because Pth provides Pthread API, platforms that do not support POSIX thread can use it taking advantage of high portability of Pth. However, since Pth is a user-space thread, multiprocessors are not utilized. Because of non-preemptive scheduling, Pth requires thread-safe functions, but not reentrant functions. Thread context switch is achieved either `makecontext(3)/switchcontext(3)` pair or `sigsetjmp(3)/siglongjmp(3)` pair [5].

Pth can be viewed as a scheduler portion of event driven applications. Application programmers do not need to write their own scheduling when they write an event driven application using Pth. This might speed-up development time by freeing up the scheduler coding time and debugging associated with the scheduler. In addition, the source code is expected to be more readable than event based application codes, which tend to have scheduling code scattered everywhere. In our case, unfortunately, both developing time and readability are not as good as we expected because reading from the socket cannot be separated from parsing, and to achieve high performance, additional Pth library calls are made. Further details appear in Section 4.

2.2 LinuxThreads

POSIX thread implemented on Linux is not a true thread but processes. By using processes, LinuxThreads meets requirements of POSIX thread such as preemptive scheduling. Naturally it is kernel-level threads created by clone() system call, and the scheduling is in the kernel. Other POSIX threads projects on Linux was replaced by LinuxThreads mainly because of its superior performance; in other words, although process based, LinuxThreads has optimized the use of processes as threads so that negative aspects of using processes are compensated, which include expensive process context switching. Because the kernel-level implementation, LinuxThreads can take advantage of multiprocessor architectures. Robustness and simple library code is another strength. Blocking system call enhances the robustness, but since it is process based, for example, blocking I/O system calls do not hurt the performance very much. 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 [7].

2.3 Event Driven

We explain an event driven application based on the Boa web server. For simplicity, we deal with only three parts: getting a request as `accept`, reading and parsing the request as `read`, and the response as `write`. All events are assumed to be I/O events handled by the `select()` system call. The portion checking if I/O events happened since the last time is denoted as `cpt` for check point. Each request is associated with a struct `z`, which contains the file descriptor returned by `accept()`, the amount of `read()` from socket, a buffer for reading, a pointer to the buffer up to parsed location, and other elements.

Concurrency is achieved by non-blocking I/O. For example, assume there are two pending requests associated with `z[0]` and `z[1]`. After reading a portion of HTTP request header into `z[0]`, it can be parsed while reading another request for `z[1]`. Figure 1 shows how the stack behaves in this model for the following sequence:

```
t1:  initialization phase.
t2:  scheduler found a new request.
t3:  the request was accepted and assigned to z[1].
t4:  z[1] is reading, then reached cpt.
...  no other activitie is found, so it keeps going.
t5:  z[1] is reading and parsing.  cpt.
...  two new requests are pending.  z[1] returns.
t6:  scheduler found accept is ready.
t7:  accepted both requests as z[2] and z[3].
t8:  scheduler found z[2] is ready.
t9:  z[2] reads.  cpt.  z[3]'s I/O is ready.
ta:  scheduler passes the execution control to z[3].
tb:  z[3] wrote completely, so return and leave.
tc:  scheduler found z[2]'s I/O is ready.
...  z[3]'s I/O is not ready yet.
td:  z[2] reads more.  cpt.  z[3] is not ready.
te:  z[2] finished reading.  z[3] is not ready yet.
```

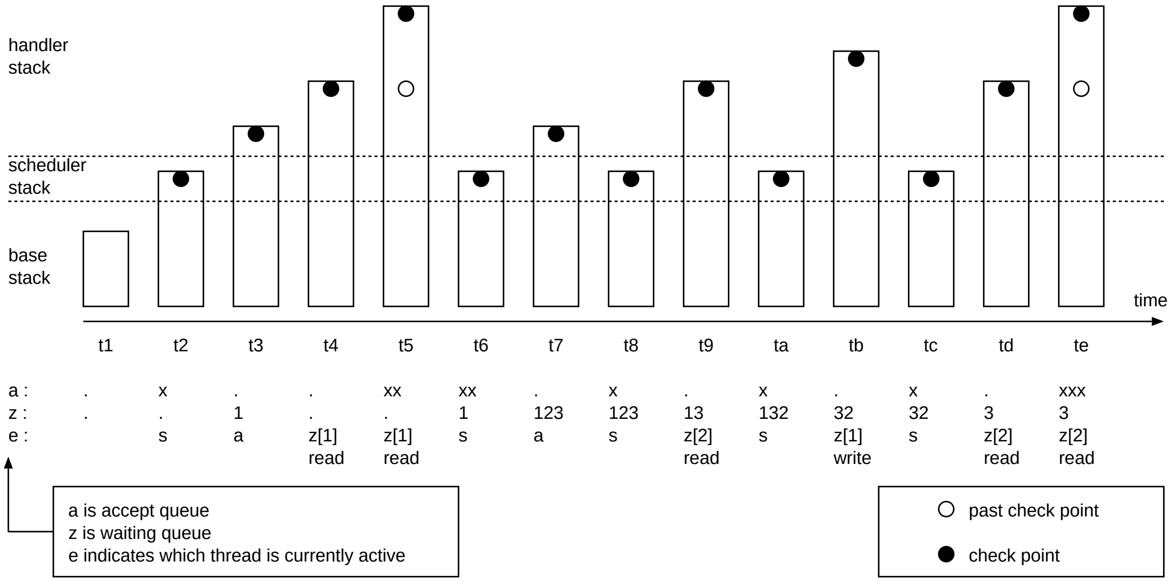

Figure 1: Event driven application behavior

3 Other Web Servers

There are three major categories of HTTP servers: process-based servers, event-driven servers, and thread-based servers. A process-based server forks off a child process in response to each incoming request. Typically, one process per request is created; this model is considered to be heavy-weight as process context switching is expensive. An example of this class is the well-known Apache Web Server [3]. Event-driven servers use the `select()` (or `poll()`) system call to internally multiplex the control of threads to handle multiple requests in a single process. Because there is no context switching, this type of server is light-weight and performs reasonably well with low memory and CPU usage [4]. Even though event-driven servers work well in less intensive environments, they scale poorly because the number of file descriptors handled by `select()` is limited [8]. In addition, the implementation of `select()` inherently limits the performance and scalability [9]. Examples of this category are Flash [10] and BOA [4]. Thread-based servers have been mentioned in many papers, but we are unaware of any previous work to implement and rigorously test this type of server. Haboob, using SEDA [11], is an example of the thread-based model, but it is actually a hybrid of event-driven and thread. The details of SEDA are discussed later in this section.

Thread-based servers are efficient in that the context switching between threads within a process is less expensive than context switching between processes. In addition, by using explicit yield, the server efficiently passes execution control to a thread that is ready to run, minimizing the latency.

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 [11] than Apache [3] and Flash [10]. 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 [11]. 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 [12]. In addition, caching and keep-alive makes the control flow a nested loop, which is not a simple pipeline.

4 Overview of CoW

CoW is a thread-based HTTP server built using GNU Pth with slight modification. Pth achieves concurrency via `select()` system calls taking advantage of non-blocking I/O [5, 6, 8].

Figure 2 gives a pictorial overview of CoW. The main thread spawns a depot and request handlers before it enters the accept loop. Its job is just to accept a new request, then pass it to a handler via the depot. The depot is a thread that coordinates between the main thread and the handlers. It receives messages from both main and handlers. A

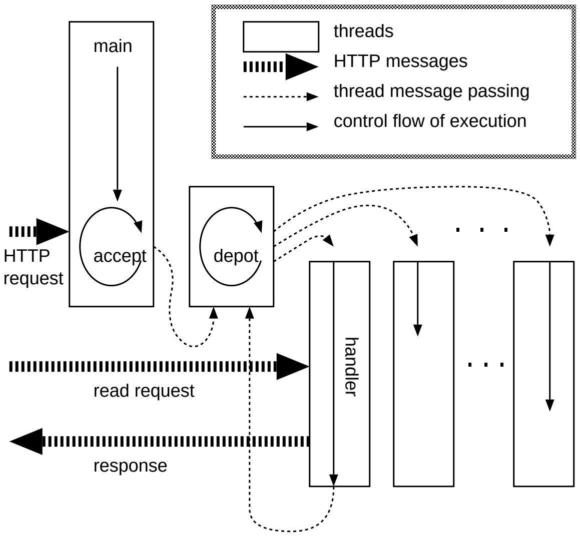

Figure 2: Overview of CoW's Internal Structure

message from main contains a file descriptor of an accepted request, which is then passed to an available request handler. A message from a handler contains the handler's thread ID so that the depot knows who is available and is able to suspend spare handlers when the request rate is low. Because the depot knows about both incoming messages and available handlers, it is a good place to control the thread scheduling. The lack of self-suspension facility in Pth is another reason to employ the depot as the control center. A handler receives one message at a time. It then reads the request from the socket, parses the HTTP request header, and takes appropriate action. After that, if *keep-alive* is not enabled or has expired, the handler sends its own thread ID to depot so that the handler can work on the next pending request or enter into the suspend mode.

The number of file descriptors available to a process is a big problem in thread applications like web servers. Even though it could be easily increased by modifying the kernel, we tested with the default kernel setting (i.e., 1,024 maximum file descriptors per process). This is partially because the bottleneck of performance is I/O, but more importantly, management of Pth messages, which contain accepted file descriptors, can not only prevent "Too many open files" errors but also eliminate undesirable "hold and wait" state on a message queue. In particular, depot is the center of message passing, and it needs to find available request handlers as quickly as possible to serve the accepted request. If the message queue is processed in FIFO order, the server may suffer from response timeout or run out of file descriptors. There are two bad situations. Suppose the message queue is filled by messages from the main thread, and all handlers are busy except for one handler that just finished its job. In this case the depot would 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, suppose the previous situation, and further suppose handlers have not opened the requested file. If the accepted request sent from main and still in depot's message queue used up all file descriptors, handlers will fail to serve the request because `open()` fails.

The reuse of pre-spawned threads as opposed to spawning a new thread per request prevents thread creation overhead. Creating a new stack for a thread is expensive because it uses malloc. The downside of this strategy is that Pth scheduler checks its internal queues (i.e., walks through the wait queue to find if any threads can be runnable and, if so, moves them to the ready queue), which introduces significant overhead. Fortunately, Pth provides `pth_suspend()`, which moves specified threads to the suspend queue. The Pth scheduler ignores threads on that queue, so suspending unused handlers eliminates the scheduling overhead mentioned above.

5 Performance

We compared four representative web servers: Apache 1.3.22 (and the beta version Apache 2.0), BOA 0.94.12-pre1, Haboob, and our CoW prototype. The benchmark program used is Gaurav Banga's benchmark [9] modified by Fritz Barnes [13]. This benchmark is important in that it can create request rates at which exceed the rate that the server can respond to requests. The benchmark handles only static contents. Both server and client (benchmark) were run on Linux kernel version 2.4.2-2 i686, with Intel Pentium III 733MHz, 256 MB main memory, RTL8139-based 100Mb/s Fast Ethernet adaptor, and NFS. Logging facilities were disabled in all servers. Table 1 shows the performance results of the benchmark platform for the representative web servers. Figure 3 presents the graph of these data. The reasons for the drop off at high request rates are different for each server: CoW's degradation is primarily caused by thread scheduling overhead, BOA suffers from the `select()` limitation, and Apache suffers from excessive process context switching.

As Apache's main goal has never been performance, it is not the fastest HTTP server available. Instead, Apache provides robustness in the first place, then performance, security, flexibility,

Table 1: Web servers' performance data

Request Rate per second	Number of Successful Replies over 30 sec Interval: ideal	CoW	BOA	Apache 1	Apache 2	Haboob
50	1500	1497	1492	1494	1484	792
100	3000	2991	2985	2968	2925	1016
200	6000	5957	5777	5713	5219	1017
300	9000	6705	5972	6001	5625	1016
400	12000	6643	5709	5966	5498	1016
500	15000	6224	5842	5937	5536	1017

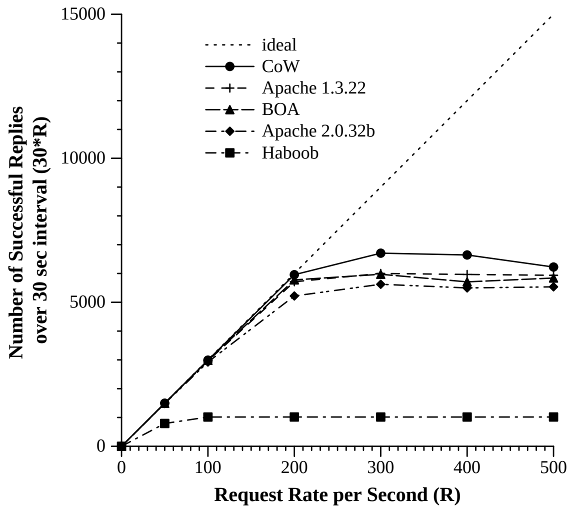

Figure 3: Web servers' performance graph

and extensibility.

BOA, on the other hand, seeks performance, but good performance under limited resource availability. In fact, the author of BOA is aware of restrictions on embedded systems and old computers. As a result, BOA is one of the most popular HTTP servers in the niche of embedded community. Because our test-bed does not have resource restrictions, BOA failed to show its strength. Instead, negative effects caused by the limitation of `select()` at high request rates were observed. It is important to note that %cpu at high request rate never exceeds 10%, typically around 5%, as opposed to CoW which uses 10-15% CPU cycle at high request rates.

As mentioned in Section 4, CoW's performance is about 10% better than Apache and BOA at high request rates although the performance at low request rates is very close. This performance gain is due to efficient implementation of the Pth library as well as fine management of the number of file descriptors. Yielding only when needed, and discrimination of message treatment at the depot are other reasons for the efficiency. Because each thread can have its own set of file descriptors used by `select()`, the bottleneck observed in BOA is not apparent in CoW even when counting the additional thread overhead. The table and graph shows the results of CoW with 32 request handlers. The performance is roughly constant if the number of handlers is between 16 and 256. If it is too small (8 or less) or too big (512), the performance degrades due to the lack of parallelism and overhead of Pth scheduling, respectively.

Apache 2.0.32, the first beta version of the Apache 2.0 series, was released just at the time of this writing. Because Apache 2 supports pthread, we tested it with Linux pthread as well as Pth with pthread API. The pthread version compiled fine after few problems, but Pth version compilation was difficult, and several header files needed to be modified in order to compile against Pth with pthread API [6, 7, 5]. Because the testing environment was on NFS, both thread versions required locking in the local disk. However, even then, they are both unstable. The pthread version terminates after 400–800 replies, and the Pth version never replied. Hence, we are unable to provide meaningful comparison data, although we hope to when newer versions of Apache 2 become available. Apache 2 provides configuration option for threading; for example, *—with-mpm=worker* is a multiprocessing module with threading via pthread, *—with-mpm=prefork* is one child process per request (non-thread). The default configuration for UNIX variant is *prefork*. Pthreads and Pth versions were compiled with the *worker* option. The prefork version ran fine, but, as seen in Table 1, its performance is not as good as Apache 1.3.22's.

Haboob did not perform as well as we expected based on Reference [11], which showed that Haboob outperformed the other popular web servers. The differences are due to running different benchmarks (SPECWeb [14] vs. ours) with different measurement parameters in very different computing environments (enterprise server vs. PC server).

Table 2: Maximum number of handlers active at a time (request rate of 300 per second)

maximum pool size	16	32	64	128	256	512
max_conn	16	32	34–42	42–46	42–44	40–43

The maximum number of handlers concurrently active (call it max_conn) during the benchmark depends on the pool size of handlers. The code in main's `accept()` loop also constrains max_conn for adjusting the behavior of the server based on available file descriptors, size of accept queue, and pending messages for the depot. This is essential to keep the server responsive. Table 2 shows the summary for a request rate of 300. This number also varies according to the scheduling policy placed various points in CoW's source code.

6 Conclusion

Although threads can be used to write clean code, thread-based code is not as simple as process-based application code. Scheduling code is not needed in threading unlike in the event-driven model, however, knowledge of the internal thread mechanism is critical to achieve robust and efficient thread-based applications. Difficulty of debugging is another issue of this model. However, when considering performance, better I/O facilities are essential. SFIO could be helpful for better performance [15]. If I/O controller hardware improves, it is clear to see the server can utilize CPU more efficiently, thus giving better performance. In summary, even though CoW does not fully meet the goals mentioned in the introduction, its architecture is conceptually cleaner than BOA and its performance is better than Apache's and BOA's. We plan to further improve CoW by using other thread packages (e.g., SR's [16, 17] thread code) or port to other concurrent programming languages such as SR [16, 17], SiR [18] and Dynamic C [19]. Dynamic C, which provides CM-style synchronization mechanisms, is designed for and runs on embedded systems, thus testing under embedded systems to demonstrate better performance and less resource requirements is particularly interesting.

Acknowledgements

We thank Fritz J. Barnes and Gene Fodor for providing us helpful suggestions.

References

[1] February 2002. http://www.serverwatch.com/.

[2] The World Wide Web Consortium (W3C), February 2002. http://www.w3c.org/.

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

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

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

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

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

[8] 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.

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

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

[11] 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.

[12] R. Fielding (UC Irvine), J. Gettys (DEC/Compaq/W3C), J. Mogul (DEC/Compaq), L. Masinter (Xerox), P. Leach (Microsoft), H. Frystyk, and T. Berners-Lee (W3C/MIT/LCS). Hypertext transfer protocol — HTTP/1.1, June 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[13] Raju Pandey, J Fritz Barnes, and Ronald Olsson. Supporting quality of service in HTTP servers. In *Proceedings of the Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 247–256, Puerto Vallarta, Mexico, June 1998.

[14] SPECWeb. http://www.specbench.org/osg/web99/.

[15] Glenn Fowler, David Korn, and Kiem-Phong Vo. SFIO: A Safe/Fast I/O Library, 2002. www.research.att.com/sw/tools/sfio/.

[16] The SR programming language, version 2.3.2, August 1999. http://www.cs.arizona.edu/sr/.

[17] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, California, 1993.

[18] E. F. Fodor and R. A. Olsson. Interoperable concurrency support with SIR middleware. In *Workshop on Middleware, held in conjunction with the 19th IEEE International Conference on Distributed Computing Systems (ICDCS-99)*, pages 84–89, June 1999.

[19] Z-World, Inc. *Dynamic C 5.x Integrated C Development System Application Frameworks (Rev.1)*, 1998. Dynamic C 5.x.