RICE UNIVERSITY

An Efficient Threading Model to Boost Server Performance

by

Anupam Chanda

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Willy Brnaepoel
Dr. Willy Brnaepoel, Chair
Karl F. Hasselman Professor of Computer Science

Alan L. Cox
Dr. Alan L. Cox
Associate Professor of Computer Science

Peter Drischel
Professor of Computer Science

HOUSTON, TEXAS

MAY, 2003
An Efficient Threading Model to Boost Server Performance

Anupam Chanda

Abstract

Multi-threading is a popular choice for server architecture. Widely used servers, like the Apache web server and the MySQL database server, are written in a multi-threaded fashion. We investigate the effects of thread architecture on server performance from two angles: (1) number of user threads per kernel thread, and (2) use of blocking I/O vs. non-blocking I/O. We propose N-to-M threads with non-blocking I/O, a novel threading model, to provide higher performance for servers, and explain its advantages over other existing thread architectures, viz., 1-to-1 threads with blocking I/O, N-to-1 threads with non-blocking I/O, and N-to-M threads with blocking I/O. We demonstrate the efficacy of this threading model by showing performance improvement for Apache and MySQL. Results show that our threading model provides a performance improvement of 10-22% for Apache (for synthetic and real workloads), and 10-17% for MySQL (for TPC-W workload) over existing thread models.
Acknowledgments

I thank my advisors Dr. Willy Zwaenepoel and Dr. Alan L. Cox for their support, encouragement and guidance. Special thanks to Willy for coming all the way from Lausanne, Switzerland for my defense. Special thanks to Alan for providing ideas whenever I got stuck in any problem.

Many thanks to Dr. Peter Druschel for providing valuable suggestions regarding my thesis.

I thank my research group members, Khaled Elmeleegy, Romer Gil and Sumit Mittal for their help and suggestions.

Finally, I thank my family and friends for their support and encouragement.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi
List of Tables vii

1 Introduction 1
  1.1 Motivation ........................................... 1
  1.2 Contributions ........................................... 2
  1.3 Organization ........................................... 3

2 Thread Library Architecture 4
  2.1 Different Threading Models .............................. 4
  2.2 Threading Model and Application Performance ............... 6
  2.3 ServLib: An N-to-M Thread Library with Non-blocking I/O ....... 8

3 Handling Large Transfers: An Optimization 11
  3.1 Large Write in an N-to-1 Library .......................... 11
  3.2 Large Write in a 1-to-1 Library ............................ 13
  3.3 Optimization in ServLib .................................... 13

4 Performance Evaluation 18
  4.1 Apache Results .......................................... 18
    4.1.1 Synthetic Workload ................................... 19
    4.1.2 Experiment on Large Transfer Optimization ............... 23
4.1.3 Trace-based Workload ........................................ 23
4.2 MySQL Results ..................................................... 28

5 Related Work ......................................................... 31
  5.1 Hybrid Architectures ............................................ 31
  5.2 Thread Library for Servers .................................... 32

6 Future Work .......................................................... 33
  6.1 Effects of Preemption ........................................... 33
  6.2 Comparison with Event-driven Architecture ................. 33
  6.3 Out-of-memory Workloads .................................... 34

7 Conclusion ............................................................ 35

Bibliography ............................................................. 36
Illustrations

2.1 Blocking I/O vs. Non-blocking I/O .................................................. 9

3.1 Handling large writes in N-to-1 thread libraries ............................... 14
3.2 Handling large writes in ServLib ......................................................... 17

4.1 Network throughput with varying file sizes, 10 concurrent connections ... 20
4.2 Network throughput with varying file sizes, 30 concurrent connections ... 20
4.3 Network throughput with varying file sizes, 50 concurrent connections ... 21
4.4 Response time with varying file sizes, 10 concurrent connections ........... 21
4.5 Response time with varying file sizes, 30 concurrent connections .......... 22
4.6 Response time with varying file sizes, 50 concurrent connections .......... 22
4.7 Request distribution for CS trace with increasing file size .................... 24
4.8 Request distribution for NASA trace with increasing file size ............... 25
4.9 Network throughput for CS trace with varying number of concurrent
    connections ....................................................................................... 25
4.10 Response time for CS trace with varying number of concurrent connections 26
4.11 Network throughput for NASA trace with varying number of concurrent
    connections ....................................................................................... 26
4.12 Response time for NASA trace with varying number of concurrent
    connections ....................................................................................... 27
4.13 MySQL throughput with varying number of clients ............................ 29
4.14 MySQL response times with varying number of clients ....................... 30
## Tables

2.1 Thread Library Architectures ........................................... 5

4.1 Web Trace Characteristics .............................................. 27
Chapter 1

Introduction

1.1 Motivation

The two most popular architectures for server implementation are multi-threading and event-driven. The generally accepted view of these architectures is that multi-threaded servers are easier to develop, maintain, and extend but event-driven servers achieve higher performance. However, among large, complex, I/O intensive servers, like Apache [2] and MySQL [10], multi-threading is the more popular choice. We argue that with proper choice of a threading architecture such an application can be made to achieve higher performance.

We consider threading architectures with respect to two important design criteria, viz.

1. Number of user threads per kernel thread, and

2. Use of blocking I/O vs. non-blocking I/O.

With respect to the first criterion, we consider 1-to-1 threads (every user thread runs on top of a kernel thread), N-to-M threads (N user threads are multiplexed on top of M kernel threads), and N-to-1 threads (N user threads are multiplexed on top of one kernel thread).

With respect to the second criterion, thread libraries either transparently convert all application issued blocking I/O to non-blocking I/O or use application issued blocking I/O “as is”. Different combinations of these two criteria lead to many possible thread architectures. In this thesis we consider the effects the choice of one of these thread architectures has on the performance of a server application and demonstrate that N-to-M threading model with
non-blocking I/O outperforms other existing threading models for such applications.

1.2 Contributions

The contributions of this thesis are as follows. We explain the shortcomings of the following thread architectures:

1. 1-to-1 threads,

2. N-to-1 threads, and

3. N-to-M threads with blocking I/O,

from a performance viewpoint for server applications. We propose the N-to-M thread model with non-blocking I/O as a novel thread architecture, and explain its benefits over other thread architectures. To the best of our knowledge, there is no existing work on this thread model.

We have designed and implemented ServLib, modeled after this thread architecture; ServLib exposes the standard POSIX threads (pthreads) API. We demonstrate the efficacy of this thread architecture by boosting the performance of the Apache web server [2] and the MySQL database server [10]. This is done by transparently linking them against ServLib. Inspired by the event-driven architecture we have performed an optimization in ServLib that reduces the number of context switches for large I/O transfers. This is a novel optimization for threaded architectures. Our results show that for Apache, ServLib has a 10-22% performance improvement over other threading models for synthetic and real workloads, and for MySQL, ServLib attains 10-17% performance improvement for the TPC-W [4] workload. These results highlight the effectiveness of the N-to-M threading model with non-blocking I/O, and our optimization for large I/O transfers. Additionally,
the results strongly suggest the requirement for a scalable event notification mechanism like kevent.

Our ultimate goal is to match the performance of event-driven architecture but in a multi-threaded fashion. But such an experiment is beyond the scope of this thesis.

1.3 Organization

The rest of the paper is organized as follows. In Chapter 2 we compare the different thread architectures. In Chapter 3 we explain the I/O optimization we have added to ServLib for performance improvement. We explain our experiments and do a performance evaluation of the thread architectures in Chapter 4. We describe related work in Chapter 5 and future work in Chapter 6. Finally we conclude in Chapter 7.
Chapter 2

Thread Library Architecture

2.1 Different Threading Models

We consider thread architectures with respect to two important design criteria, viz.

1. Number of user threads per kernel thread, and

2. Blocking I/O vs. Non-blocking I/O

With respect to the first design criterion, existing thread libraries fall into one of the following categories:

1. \textit{1-to-1}: Each user thread runs on top of one kernel thread.

2. \textit{N-to-M}: N user threads are multiplexed on top of M kernel threads.

3. \textit{N-to-1}: N user threads are multiplexed on top of 1 kernel thread.

Thread libraries use either blocking I/O or non-blocking I/O for their operation. In the former case, all blocking I/O from the application are issued "as is" by the library, while in the latter case, the library transparently makes all blocking I/O of the application non-blocking. This is achieved by marking all sockets, file descriptors, etc. as non-blocking by the library. For the rest of this chapter, we refer to all sockets, file descriptors, pipes, etc. as simply \textit{descriptors}.

Combining the above two design criteria, we can summarize the existing thread libraries as shown in Table 2.1.
<table>
<thead>
<tr>
<th>User thds-to-Kernel thds</th>
<th>Blocking I/O</th>
<th>Non-blocking I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-to-1</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>N-to-M</td>
<td>X</td>
<td>*</td>
</tr>
<tr>
<td>N-to-1</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2.1: Thread Library Architectures

In Table 2.1, we have marked all logical combination of thread libraries with a 'X'; '-' stands for architectures which provide no benefit and hence have not been implemented. Linux’s native pthreads library is an example of 1-to-1 threads with blocking I/O, FreeBSD’s native pthreads library is an example of N-to-1 threads with non-blocking I/O, and Solaris’s (till version 8.0) native pthreads library is an example of N-to-M threads with blocking I/O. To the best of our knowledge, there is no existing N-to-M thread library using non-blocking I/O, and we have marked it with a '*' . We propose this threading model, and show its benefits over other architectures.

For doing non-blocking I/O, thread libraries typically use event notification mechanism like select, or poll to query which descriptors are ready for I/O. One thing has to be noted here that though operations like read and write have non-blocking counterparts (e.g. by marking the descriptor as non-blocking), there is no non-blocking open or stat. Furthermore events like page faults are essentially blocking. So a kernel thread that blocks at these system calls or due to page faults causes the application, running on top of the thread library, to block.

To handle blocking events the library can use something like Scheduler Activations [1] to spawn a new kernel thread to continue execution of the application. We argue that using non-blocking I/O mechanisms is better as it allows batching of information across
user/kernel boundary. For the same reason, we advocate greater support for non-blocking operations, e.g., support for non-blocking open or stat. Scheduler Activations [1] provide a solution to this problem but come with the added cost of creation and termination of kernel threads [13].

2.2 Threading Model and Application Performance

In this section we explain how different threading models affect server application performance.

- **1-to-1 thread library** - 1-to-1 thread libraries typically employ blocking I/O. Since one kernel thread performs I/O on only one descriptor at a given time, there is no reason for it to employ non-blocking I/O. When a kernel thread in such a library blocks, the application running on top of the library does not block because there are other runnable threads, and the CPU simply switches to one of them. The problems with this thread architecture are:

  1. Any descriptor becoming ready causes the corresponding thread to be put in the run-queue, by the kernel scheduler. The number of context switches increases with larger transfers (reads/writes) and increasing number of descriptors.

  2. Kernel-level context switches between different threads are a source of overhead.

- **N-to-1 thread library** - N-to-1 thread libraries cannot afford to use blocking I/O, as any one user thread blocking for I/O would cause the entire library to block, and the application’s performance would suffer immensely. Such libraries convert all blocking descriptors to non-blocking, and employ event notification mechanisms like
select/poll to be notified of readiness of the descriptors. The problems with this architecture are:

1. Select and poll don’t scale well with increasing number of descriptors [3].

2. If the kernel thread blocks, e.g. due to page-faults, the application has to block as well.

3. Multi-processor systems are underutilized as the library can use only one processor at a time.

- **N-to-M thread library/blocking I/O** - Existing N-to-M thread libraries typically use blocking I/O, and they employ a mechanism, like Scheduler Activations [1], to be notified of kernel threads blocking due to I/O. They get upcalls from the kernel when such events happen, and they spawn kernel threads to continue execution of the application. The problem with this architecture is:

  1. Because I/O is blocking, any time a descriptor becomes ready, the kernel scheduler has to put the (previously) blocking thread in the run-queue. Scheduling overhead increases with larger transfers and increasing number of descriptors.

- **N-to-M thread library/non-blocking I/O** - We advocate this thread library architecture which uses non-blocking I/O in an N-to-M library. All descriptors are converted to non-blocking, and the library scheduler is informed by some event notification mechanism about descriptors as they become ready for I/O. Scheduler Activations-like [1] mechanisms are used to spawn new kernel threads when a thread blocks, e.g., due to page faults.

  1. Having fewer kernel threads in this architecture reduces the number of kernel context switches compared to 1-to-1 thread libraries. Library context switches
are less expensive. Compared to N-to-M threads with blocking I/O, fewer kernel threads will be required to save contexts of unfinished I/O for large transfers. This number can be significantly less because it is not proportional to the number of threads performing I/O.

2. By converting all blocking descriptors to non-blocking ones, the library scheduler can get a batch of ready descriptors when it is run, whereas if the descriptors were blocking, only one thread could wait on it, and it would be scheduled every time the descriptor becomes ready. This is illustrated in Figure 2.1. The left hand side of the figure shows many kernel threads waiting on blocking descriptors (shown as narrow pipes), and the right hand side shows one kernel thread (the library scheduler) waiting on many non-blocking descriptors (shown as a fat pipe). Whenever a descriptor becomes ready, the corresponding kernel thread has to be scheduled, for the blocking I/O case. However, for the non-blocking I/O case, there is a good chance that when the library scheduler eventually runs, after being scheduled because of a descriptor becoming ready, a few other descriptors have also become ready, and hence it gets a batch of ready descriptors.

2.3 ServLib: An N-to-M Thread Library with Non-blocking I/O

We have implemented ServLib, a N-to-M thread library which transparently converts the application’s blocking I/O to non-blocking I/O whenever possible. ServLib uses FreeBSD’s kevent event notification mechanism to be notified of ready descriptors. Select and poll are inefficient because they don’t scale with increasing number of descriptors, kevent is both scalable and efficient [8]. ServLib employs Scheduler Activations-like [1] mechanism to
spawn a kernel thread when some thread blocks, e.g., due to page-faults or operations on descriptors that have no non-blocking counterparts. This is implemented as a simple extension to the already existing *kevent* system call.

This extension enables a thread (watcher) to register interest on another thread (watched) and receive events when the watched thread blocks, e.g., due to page faults. Then the watcher thread can spawn additional threads to handle execution of the application. This implementation required less than 10 lines of code in the FreeBSD 4.7 kernel.

In addition to the gains over other thread library architectures (as mentioned above), we do the following optimization in ServLib:

- For large transfers, we minimize the number of context switches. Large transfers are handled in existing thread libraries by doing I/O till the thread blocks, then saving the context of the thread and pending I/O and going to the scheduler, and by context switching back to the blocking thread when the descriptor for I/O becomes ready. So every time the I/O blocks and every time the descriptor becomes ready there is a context switch from and to the thread issuing the I/O, respectively. In ServLib, the library scheduler maintains information about pending I/O by the blocking thread,
and does I/O on behalf of the blocking thread in a non-blocking fashion when the descriptor becomes ready. Eventually, the I/O finishes, and a context switch is made to the blocking thread. If the large transfer blocks $n$ times, the number of context switches in traditional thread libraries will be $O(n)$, while it is only $O(1)$ in ServLib. This optimization is further illustrated in the next chapter.
Chapter 3

Handling Large Transfers: An Optimization

In this chapter we compare the operation of ServLib with respect to large blocking transfers (e.g. socket read/write) vis-a-vis an N-to-1 thread library, and a 1-to-1 thread library. We explain the difference in operation of ServLib from other libraries by a simple example. Consider that a large write has to be performed on a socket, and also that the socket has been marked as blocking by the application. We explain how an N-to-1 and a 1-to-1 library handles such a write, and why they have an overhead of higher context switches. Then we explain the optimization we have performed in ServLib to eliminate this overhead.

Event-driven servers set up continuations, contexts for unfinished I/O, when they block in the middle of large transfer. Subsequent transfers are performed when the sockets become ready for doing so. Our optimization is inspired by this feature of event-driven server architecture and is novel to thread architectures.

3.1 Large Write in an N-to-1 Library

An N-to-1 library transparently converts all blocking sockets to non-blocking ones. This is done through wrappers for system calls. A large write is performed in such thread libraries in the following fashion:

```c
write(s, buf, nbytes) {
    int bytes = 0;
    while (bytes < nbytes) {
```
int n;

n = sys_write(s,buf,nbytes-bytes);
/* partial write, until block */
bytes += n;
buf = (char *) buf + n;

if (bytes < nbytes)
    scheduler(s);
    /* call scheduler,
       * thread suspended
       */

    /* return from scheduler
    * socket s is ready
    */
}
return (nbytes);
}

In the above piece of pseudo C-code, s is the socket on which write is being performed, buf is the buffer to be written, and nbytes is the number of bytes to be written on the socket. The wrapper for the write system call makes traps to the kernel for partial writes in a non-blocking fashion. When further data cannot be written to the socket (e.g. because the socket buffer is full), the wrapper calls the library’s thread scheduler asking it to be awakened when further data can be written to the concerned socket. The scheduler monitors the socket for such conditions and eventually wakes up the concerned thread to perform the next chunk
of the transfer. This sequence of events goes on until the entire buffer has been written to the socket. Figure 3.1 illustrates the above operation.

Figure 3.1 shows that for each partial write that takes place, control is transferred between the stacks on which the scheduler and the user thread are running. This context switching between threads is an overhead.

### 3.2 Large Write in a 1-to-1 Library

In the case of a 1-to-1 thread library, the write is performed in a single system call. But, internally, the write blocks when the socket buffer becomes full. The kernel scheduler wakes up the thread when the socket becomes writeable again. This requires many context switches to wake up and suspend the thread performing the write, and the operation is similar to that of Figure 3.1, except that each write is handled by a kernel thread, and its waking up is handled by the kernel scheduler.

### 3.3 Optimization in ServLib

The number of context switches occurring in existing N-to-1 or 1-to-1 thread libraries (to handle large transfers) grows with the size of the transfer. Here is how ServLib avoids this problem. ServLib internally marks all blocking sockets to non-blocking ones. ServLib also provides a wrapper for the write system call, pseudo code and explanation for which are given below:

```c
write(s, buf, nbytes) {
    int bytes = 0;
    bytes = sys_write(s, buf, offset);
```
Figure 3.1: Handling large writes in N-to-1 thread libraries
if (bytes < nbytes) {
  /* set up how many
   * more bytes are to be
   * written to the socket
   */
  setup_write_context(s);

  scheduler(s);
  /* call scheduler,
   * thread suspended
   */

  /* return from scheduler
   * write has finished
   */
}

return (nbytes);

}

scheduler(socket s) {
  while (!write_finished) {
    while (sock_not_ready(s)) {
      //do other stuff
    }
    partial_write(s);
  }
} /* write finished */
return;
}

The wrapper makes a first trap for writing to the socket in a non-blocking fashion. If the write finishes at this point, it simply returns to the application, otherwise it sets up a context regarding the unfinished write on the socket (how many more bytes to write, where to write from, etc.) associated with the socket and calls the scheduler. The scheduler monitors the socket to see if further writes can be performed on the socket. If and when further writes are possible, the scheduler performs them on its own stack, without doing a context switch back to the thread that initiated the write. Finally when all required bytes have been written to the socket, the scheduler returns to the thread, and the thread can return to the application code. This operation is illustrated by Figure 3.2 which shows that at most one context switch takes place from the user thread to the scheduler and vice-versa.

With this optimization, ServLib overcomes the overhead of context switching associated with large transfers to or from sockets.
Figure 3.2: Handling large writes in ServLib
Chapter 4

Performance Evaluation

In this chapter we present experimental results that compare the performance of ServLib with other threading libraries. *I-to-1 threads* is Linux’s 1-to-1 pthread library ported to FreeBSD. *N-to-1 threads* is FreeBSD’s N-to-1 native pthread library. We use two popular multi-threaded applications, the Apache web server [2] and the MySQL database server [10].

4.1 Apache Results

We built Apache 2.0.43 [2] on top of ServLib, *I-to-1 threads*, and *N-to-1 threads* separately, and compared their performance under both real and synthetic workloads. For the rest of this section we refer to Apache with ServLib as *Apache-ServLib*, Apache with *I-to-1 threads* as *Apache-I-to-1*, and Apache with *N-to-1 threads* as *Apache-N-to-1*.

Apache 2.0.x with its *worker* multi-processing module (*mpm*) employs a hybrid architecture of multiple processes (address spaces) each having multiple threads (pthreads) to handle requests. Each process has one listener thread listening to the listen socket, and a few worker threads to process incoming connections accepted by the listener thread of the same process. In our server configuration, we keep the number of worker threads per process to 25, which is the default value. Regarding handling of persistent connections, we stick to the default configuration, which is to allow a maximum of 100 requests on a persistent connection.
For each of Apache-ServLib, Apache-1-to-1, and Apache-N-to-1 we ran the server with the same configuration parameters to enable a meaningful comparison among them. The TCP/IP sendbuffer size on the server machine was set to a default value of 32 Kbytes.

The experiments were performed with the Apache servers running on a 2.4 GHz Intel Xeon machine with 2 GB of memory running FreeBSD 4.7-RELEASE. The server and the client machines were connected by two direct Gigabit links (2 Gigabit/second of bandwidth). The client machine ran simulated HTTP clients which made requests as fast as the server machine could handle.

4.1.1 Synthetic Workload

In our first experiment, a set of clients repeatedly request the same file, and the file size is varied in each test. We also vary the number of clients in our tests. This simple test allows the servers to perform at their peak efficiency. We ensure that the client machine CPU and the network between the client and server machines do not become a bottleneck. In these experiments, the server CPU was 100% busy. We measured performance numbers in terms of network throughput, and response times. We collected kernel profile statistics for these experiments as well.

In Figure 4.1, Figure 4.2, and Figure 4.3 we show the network throughput obtained by Apache-ServLib, Apache-1-to-1, and Apache-N-to-1 with 10, 30, and 50 concurrent connections, respectively. We vary the file size from 8 Kbytes to 256 Kbytes in these experiments. Apache-ServLib outperforms the other two servers by 10-15%. We show response time in each of the above experiments in Figures 4.4, 4.5, and 4.6. As the figures show, Apache-ServLib attains 10-15% reduction in response time over Apache-N-to-1 and Apache-1-to-1.

We collected kernel profile statistics for the above experiments. We found that for
Figure 4.1: Network throughput with varying file sizes, 10 concurrent connections

Figure 4.2: Network throughput with varying file sizes, 30 concurrent connections
Figure 4.3: Network throughput with varying file sizes, 50 concurrent connections

Figure 4.4: Response time with varying file sizes, 10 concurrent connections
Figure 4.5: Response time with varying file sizes, 30 concurrent connections

Figure 4.6: Response time with varying file sizes, 50 concurrent connections
Apache-1-to-1 tests, the number of times a kernel thread was set runnable after a socket became ready was almost 140 times to the same number for Apache-ServLib, and Apache-N-to-1. The number of voluntary context switches (due to socket I/O) for Apache-1-to-1 was almost 40 times greater than that of Apache-ServLib, and Apache-N-to-1. This explains why Apache-ServLib performed better than Apache-1-to-1. For Apache-N-to-1 runs, the profile results showed that poll was the fourth most costly system call, while kevent calls in the Apache-ServLib profile results were much less expensive. This explains why Apache-ServLib performed better than Apache-N-to-1 as well. This shows the need for a scalable and efficient event notification mechanism like kevent for doing non-blocking I/O efficiently.

4.1.2 Experiment on Large Transfer Optimization

In this experiment we turned off the feature in ServLib that the scheduler resumes the big transfers on its current stack. Instead, we made the scheduler switch back to the thread issuing the transfer to resume the transfer whenever the descriptor is ready for more data to be written, and switch back to the scheduler again when the transfer blocks. This is how existing thread libraries work. We varied the file size from 8 Kbytes to 256 Kbytes and varied the number of clients from 10 to 50 and compared the performance of Apache-ServLib with and without this optimization. This experiment showed that turning this feature off causes Apache-ServLib to lose up to 4% in throughput, and response time goes up by 5%.

4.1.3 Trace-based Workload

The previous tests indicated the servers' maximum throughput on a cached workload. For performance under real workloads, we subject the servers to client requests by replaying traces from existing Web servers. The web traces used in this set of experiments were the
same as used by Kim, Pai, and Rixner [14]. We used the Rice Computer Science (CS) departmental traces, and web traces from NASA. Both of these traces have a heavy-tailed distribution of file sizes (large number of small files, and small number of large files). This is shown in Figure 4.7 and Figure 4.8.

In Figure 4.9 we show network throughput for the CS trace for the three servers and a varying number of concurrent connections. Figure 4.10 shows the corresponding response time numbers. Figure 4.11 and Figure 4.12 show network throughput numbers and response time numbers for the three servers. These figures show that Apache-ServLib outperforms the other servers by about 22% for trace based workloads.

Table 4.1 shows the characteristics of the CS and NASA web traces. The column Total stand for total bytes transferred for the trace. In this total bytes transferred, the fractions contributed by small files (size less than or equal to 8 Kbytes), medium files (size in be-
Figure 4.8: Request distribution for NASA trace with increasing file size

Figure 4.9: Network throughput for CS trace with varying number of concurrent connections
Figure 4.10: Response time for CS trace with varying number of concurrent connections

Figure 4.11: Network throughput for NASA trace with varying number of concurrent connections
Figure 4.12: Response time for NASA trace with varying number of concurrent connections

<table>
<thead>
<tr>
<th>Web Trace</th>
<th>Total</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>8 Gbytes</td>
<td>5.5%</td>
<td>20.2%</td>
<td>74.3%</td>
</tr>
<tr>
<td>NASA</td>
<td>89.5 Gbytes</td>
<td>3.1%</td>
<td>24.6%</td>
<td>72.3%</td>
</tr>
</tbody>
</table>

Table 4.1: Web Trace Characteristics
tween 8 Kbytes and 256 Kbytes), and large files (size greater than 256 Kbytes) are shown in
columns Small, Medium, and Large respectively. This table shows that for both these web
traces more than 70% of the transfer is contributed by large files. We have seen from the
synthetic workload results that large file transfers favor the performance of Apache-ServLib,
i.e., the gains are more pronounced. This happens because of fewer context switches in
ServLib as a result of the optimization described in Chapter 3.

4.2 MySQL Results

We built MySQL 3.23.55 [10] on top of ServLib, 1-to-1 threads, and N-to-1 threads separ-
rately, and compared their performance under the TPC-W [4] workload. For the rest of this
section, we refer to MySQL with ServLib as MySQL-ServLib, MySQL with 1-to-1 threads
as MySQL-1-to-1, and MySQL with N-to-1 threads as MySQL-N-to-1.

We collected traces of database queries by running a TPC-W [4] system. We drive
the MySQL server against this trace of database queries, and measure the throughput ob-
tained in terms of database queries processed per second. We vary the number of clients in
different runs of the experiments.

The experiments were performed with the MySQL server and clients each running on a
separate 2.4GHz Intel Xeon machine with 2GB of RAM, running FreeBSD 4.7-RELEASE.
Both machines were connected to each other by a single Gigabit link. For all the experi-
ments, we used default configuration parameters for the MySQL server.

Figure 4.13 shows the performance of MySQL-ServLib, MySQL-1-to-1, and MySQL-N-
to-1 on these traces with a varying number of clients. MySQL-ServLib shows a performance
improvement of about 15% over MySQL-1-to-1 and about 30% over MySQL-N-to-1. Fig-
ure 4.14 shows response time numbers for the three different servers for the above exper-
iments. MySQL-ServLib shows a performance improvement of up to 17% for the TPC-W
Figure 4.13: MySQL throughput with varying number of clients

workload.

We collected kernel profile statistics for the above experiments, which show that the number of times a kernel thread is set runnable for `MySQL-1-to-1` tests is almost 3 times to that for `MySQL-ServLib` tests. MySQL uses locks to synchronize access to the database, and the lock releaser makes the thread next in line waiting for the lock runnable. ServLib uses user level locks to enforce synchronization. User level synchronization is much cheaper than kernel-level synchronization. This accounts for the speedup of `MySQL-ServLib` over `MySQL-1-to-1`. For `MySQL-N-to-1` tests, the number of poll system call was about 20 times to that for `MySQL-ServLib`, and almost 7 times to that for `MySQL-1-to-1`. This explains why `MySQL-N-to-1` perform worse than `MySQL-ServLib` and `MySQL-1-to-1`. Again, this emphasizes the need for a scalable and efficient event notification mechanism like kevent for doing non-blocking I/O efficiently.
Figure 4.14: MySQL response times with varying number of clients
Chapter 5

Related Work

Related work in this field may be classified in the following two areas. They are work on hybrid architectures and work on thread library for servers.

5.1 Hybrid Architectures

Many of the recent works on high-performance servers, including Flash [11], StagedServer [7], and SEDA [12], have proposed new server architectures. Although these architectures have their advantages, none of them have achieved widespread use, in part, because existing servers, especially multi-threaded ones, would require adaptation to these architectures. In contrast, we do not propose a new server architecture, but propose a threading model to transparently boost the performance of multi-threaded servers.

Flash [11] is a hybrid architecture that has an event-driven core, and employs helper threads to handle blocking I/O. To handle page faults, Flash uses the mincore primitive to check if some page is resident in memory or not.

In SEDA [12] the server operation is broken into several stages, output and input of successive stages being connected by queues. SEDA aims at providing quality of service (QoS) guarantees to internet services.

StagedServer [7] batches the execution of similar operations arising in different server requests. It achieves better cache locality as a result of this batching of similar operations.
5.2 Thread Library for Servers

State Threads [9] is an N-to-1 thread library for internet applications. However, this library is not pthreads compatible, and is not preemptive. GNU Pth [5] is also an N-to-1 thread library that provides multiple threads of execution in event-driven applications. The threads are non-preemptive, and scheduled according to priorities. In contrast, ServLib is a pthreads compatible library, its threads are preemptive, and it targets multi-threaded applications.

The native pthreads library on Solaris (till version 8.0) used to be an N-to-M threading model with blocking I/O. Solaris used a flavor of scheduler activations to handle page faults and other blocking events. However, more recently, Solaris is moving towards a 1-to-1 thread library with blocking I/O [6].

Linux’s native pthreads library is a 1-to-1 thread model with blocking I/O. FreeBSD’s native pthreads library is a N-to-1 thread model with non-blocking I/O.
Chapter 6

Future Work

We intend to pursue the following directions as extensions to this work. These are to study the effects of preemption, to compare with event-driven architecture, and to experiment with out-of-memory workloads.

6.1 Effects of Preemption

It would be interesting to study the effects of preemption, particularly on 1-to-1 threads. Our belief is that 1-to-1 threads suffer from preemption at inopportune moments. Such preemptions will lead to performance degradation because of the associated cost of context switches. We intend to study what effect this might have on application performance.

6.2 Comparison with Event-driven Architecture

We intend to compare N-to-M threads with non-blocking I/O against an event-driven architecture. Our hypothesis is that the performance of N-to-M threads with non-blocking I/O will be very close to or as good as event-driven servers. This is because our threading model does many things similar to an event-driven model, viz., I/O continuations (the large I/O optimizations), batching of events, limited or controlled preemption, etc.
6.3 Out-of-memory Workloads

For the experiments described in this thesis, the workloads in all cases were in-memory. For out-of-memory workloads, N-to-1 threads (with non-blocking I/O) will suffer more because of page faults and other blocking operations. We want to do some experiments to see the effects of blocking on N-to-1 threads.
Chapter 7

Conclusion

In this thesis we have investigated threading architectures for high-performance, I/O intensive servers. We considered thread architectures from the standpoint of (1) number of user threads per kernel thread, and (2) use of blocking I/O vs. non-blocking I/O. We have pointed out the shortcomings of 1-to-1 threads with blocking I/O, N-to-1 threads with non-blocking I/O, and N-to-M threads with blocking I/O with respect to server performance. We have proposed a previously unexplored thread model of N-to-M threads with non-blocking I/O. It wins over the other architectures because of fewer context switches, and batching of information across user/kernel boundary. We have implemented ServLib, a thread library, in this model. Using two popular multi-threaded servers, Apache and MySQL, we have shown that this thread model achieves higher performance than the other thread architectures. For Apache, ServLib obtains a performance gain of 10-22% for synthetic and real workloads. For MySQL, ServLib obtains performance gain of 10-17%. Our results also demonstrate that an efficient and scalable event notification mechanism like kevent is required for efficient non-blocking I/O. We believe that with this threading model multi-threaded servers can match the performance of event-driven servers.
Bibliography


