An Efficient Threading Model to Boost Server Performance[1]

Anupam Chanda*, Khaled Elmeleegy*, Romer Gil*, Sumit Mittal*, Alan L. Cox*, and Willy Zwaenepoel†
* Rice University, Dept. of Computer Science, 6100 Main St. MS-132, Houston, TX-77005, USA
{*anupamc, kdiaa, rgil, mittal, alc*}*@cs.rice.edu*
† Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

***Abstract*—We investigate high-performance threading architectures for I/O intensive multi-threaded servers. We study thread architectures from two angles: (1) number of user threads per kernel thread, and (2) use of synchronous I/O vs. asynchronous I/O. We underline the shortcomings of 1-to-1 threads with synchronous I/O, N-to-1 threads with asynchronous I/O, and N-to-M threads with synchronous I/O with respect to server performance We propose N-to-M threads with asynchronous I/O, a novel and previously unexplored thread model, for such servers. We explain the architectural benefits of this thread model over the above mentioned architectures. We have designed and implemented ServLib, a thread library, modeled in this architecture. We optimize ServLib to reduce context switches for large I/O transfers. ServLib exports standard POSIX threads (*pthreads*) API and can be slipped transparently beneath any multi-threaded application using such API. We have evaluated ServLib with two applications, the Apache multi-threaded web server, and the MySQL multi-threaded database server. Our results show that for synthetic and real workloads Apache with ServLib registers a performance improvement of 10-25% over Apache with 1-to-1 thread and N-to-1 thread libraries. For TPC-W workload, ServLib improves the performance of MySQL by 10-17% over 1-to-1 and N-to-1 thread libraries.**

I. Introduction

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 servers, like Apache [2] and MySQL [10], multi-threading is the more popular choice. Our goal is to investigate thread architectures for higher performance for such applications.

In this paper we consider thread architectures with respect to two important design criteria, viz.

1) Number of user threads per kernel thread, and
2) Use of synchronous I/O vs. asynchronous 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 synchronous I/O to asynchronous I/O or use application issued synchronous I/O "as is". One can think of possible combinations of these two criteria for thread architecture. We consider such architectures from the viewpoint of performance of I/O intensive, and massively parallel applications.

The contributions of this paper 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 synchronous I/O,

from a performance viewpoint. Existing N-to-M thread libraries have used synchronous I/O. We propose the N-to-M thread model with asynchronous I/O as a novel thread architecture, and explain its benefits over the 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 standard POSIX threads (*pthreads*) API. We demonstrate the efficacy of this thread architecture by boosting performance of the Apache web server [2], and the MySQL database server [10], by transparently linking them against ServLib.

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 multithreaded 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.

There have been other thread library implementations. 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.

In addition to its architectural benefit, we have performed an optimization on ServLib to further augment the performance of multi-threaded servers. This optimization reduces the number of context switches for large I/O transfers. We use the scalable *kevent* [8] system call in lieu of *select* and *poll* which are inefficient and are not scalable. Our results show that for Apache, ServLib has a 10-25% performance improvement over the other threading models for synthetic and real workloads, and for MySQL, ServLib attains 10-17% performance improvement for the TPC-W [4] workload. The performance benefits are because of the architectural advantages of our threading model, and the optimization. Sun has a new 1-to-1 thread library [6] that replaces their old N-to-M library, and they argue that its more robust and faster. However, the results of this paper show that N-to-M threads with asynchronous I/O outperform 1-to-1 threads. On the basis of our results, we argue for greater support for asynchronous I/O, e.g., asynchronous *open* or *stat*. Use of asynchronous I/O enables batching of information at user/kernel interface causing fewer user/kernel boundary crossings.

The rest of the paper is organized as follows. In Section II we compare the different thread architectures. In Section III we explain an optimization we have added to ServLib for performance improvement. We explain our experiments and do a performance evaluation of the thread architectures in Section IV. We describe related work in Section V. Finally we conclude in Section VI.

II. Thread Library Architecture

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

1) Number of user threads per kernel thread, and
2) Synchronous I/O vs. Asynchronous I/O

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

1) *1-to-1*: Each user thread runs on top of one kernel thread.
2) *N-to-M*: N user threads are multiplexed on top of M kernel threads.
3) *N-to-1*: N user threads are multiplexed on top of 1 kernel thread.

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

Combining the above two design criteria, we can summarize the existing thread libraries as shown in table I.

User thds-to-Kernel thds	Synch. I/O	Asynch. I/O
1-to-1	X	-
N-to-M	X	*
N-to-1	-	X

TABLE I
THREAD LIBRARY ARCHITECTURES

In table I, we have marked existing thread libraries with a 'X'; '-' stands for architectures which provide no benefit and hence have not been implemented. To the best of our knowledge, there is no existing N-to-M thread library using asynchronous I/O, and we have marked it with a '*'. We propose this threading model, and show its benefits over other architectures.

For doing asynchronous 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*, and 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, unless the library uses something like Scheduler Activations [1] to spawn a new kernel thread to continue execution of the application. We argue that using asynchronous I/O mechanisms is better as it allows batching of information across user/kernel boundary. For the same reason we advocate greater support for asynchronous operations, e.g., support for asynchronous 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].

We describe different thread library architectures in detail below:

- *1-to-1 thread library* - 1-to-1 thread libraries typically employ synchronous 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 asynchronous 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. 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 synchronous 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) Not efficient on multi-processor systems as the library can use only one processor at a time.
- *N-to-M thread library/synchronous I/O* - Existing N-to-M thread libraries typically use synchronous I/O, and they employ 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 synchronous, 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/asynchronous I/O* - We propose this thread library architecture which uses asynchronous 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 [1] like mechanisms are used to spawn new kernel threads when a thread blocks, e.g., due to page faults.
  1) Fewer kernel threads in this architecture reduces the number of kernel context switches compared to 1-to-1 thread libraries, and library context switches are less expensive. Compared to N-to-M threads with synchronous I/O, fewer kernel threads will be required to save contexts of unfinished I/O for large transfers.

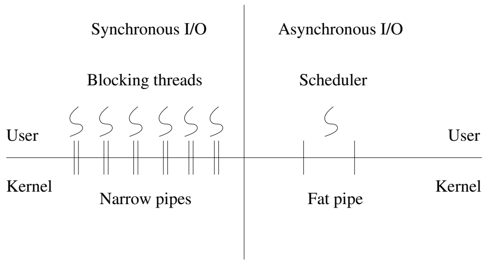

Figure 1. Synchronous I/O vs. Asynchronous I/O

  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 1. The left hand side of the figure shows many kernel threads waiting on synchronous descriptors (shown as narrow pipes), and the right hand side shows one kernel thread (the library scheduler) waiting on many asynchronous descriptors (shown as a fat pipe). Whenever a descriptor becomes ready, the corresponding kernel thread has to be scheduled, for the synchronous I/O case. However, for the asynchronous 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.

We have implemented ServLib, a N-to-M thread library which transparently converts the application's synchronous I/O to asynchronous I/O whenever possible. ServLib employs Scheduler Activations [1] like 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. 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 of 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 its only *O(1)* in ServLib. This optimization is further illustrated in the next section.

Additionally, we use 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].

III. Handling Large Transfers : An Optimization

In this section 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. An N-to-1 library transparently converts all blocking sockets to non-blocking ones. A large write is performed in such thread libraries in the following fashion:

```
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 N-to-1 library provides a wrapper for the *write* system call. The wrapper makes traps to the kernel

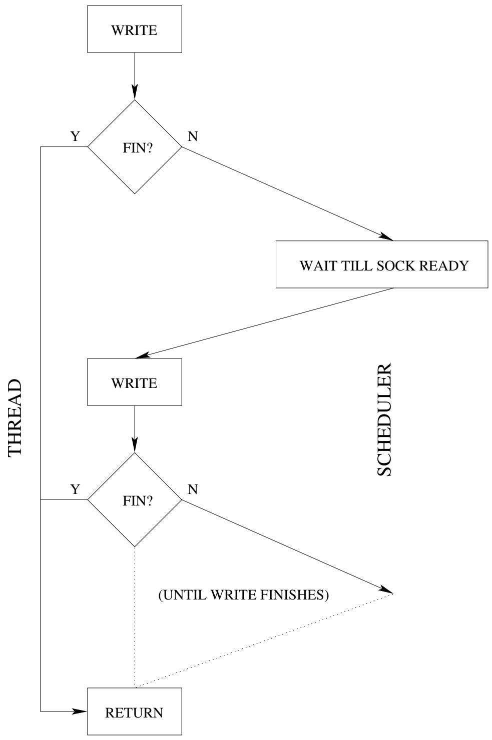

Figure 2. Handling large writes in N-to-1 thread libraries

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 thread scheduler of the library 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 transfer. This sequence of events goes on until the entire buffer has been written on to the socket. Figure 2 illustrates the above operation.

From figure 2 it is obvious 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.

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 2, except that each write is handled by a kernel thread, and its waking up is handled by the kernel scheduler.

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.

Let us now consider how we handle large transfers in ServLib. 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:

```
write(s, buf, nbytes) {
  int bytes = 0;
  bytes = sys_write(s, buf, offset);

  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 write 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 which shows that at most one context switch takes place from the user thread to the scheduler and vice-versa.

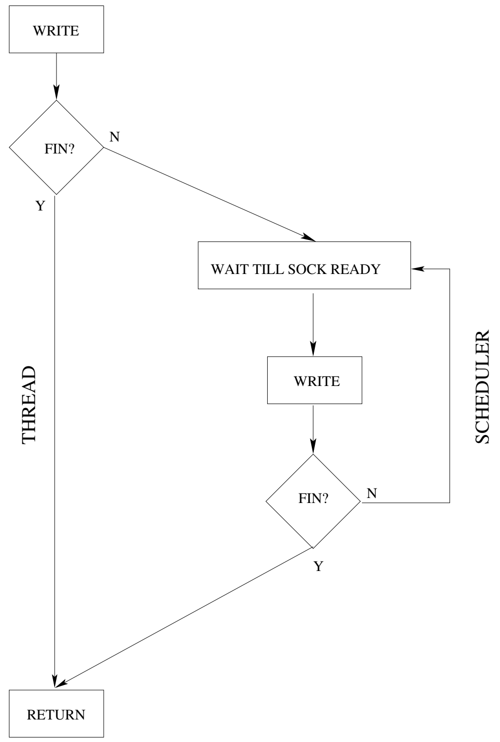

Figure 3. Handling large writes in ServLib

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.

With this optimization, ServLib overcomes the overhead of context switching associated with large transfers to or from sockets.

IV. Performance Evaluation

In this section we present experimental results that compare the performance of ServLib with other threading libraries. *1-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].

A. Apache Results

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

Apache 2.0.x 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.

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, figure 5, and figure 6 we show 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 7, 8, and 9. 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 *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 to that for *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

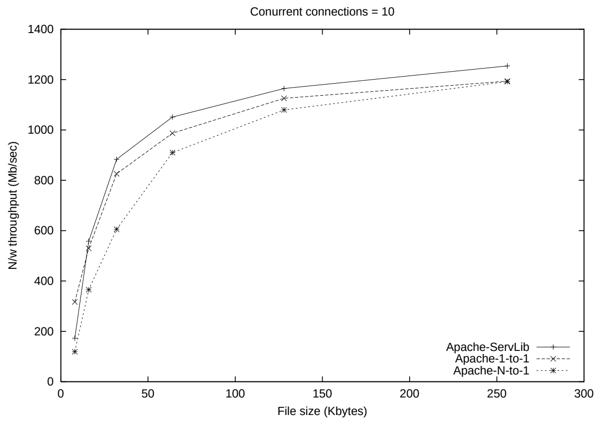

Figure 4. Network throughput with varying file sizes, 10 concurrent connections

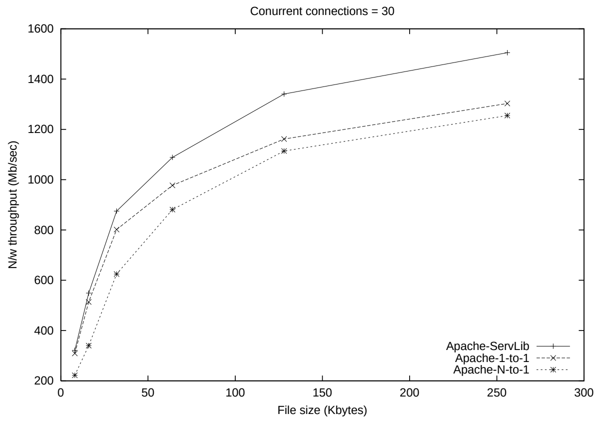

Figure 5. Network throughput with varying file sizes, 30 concurrent connections

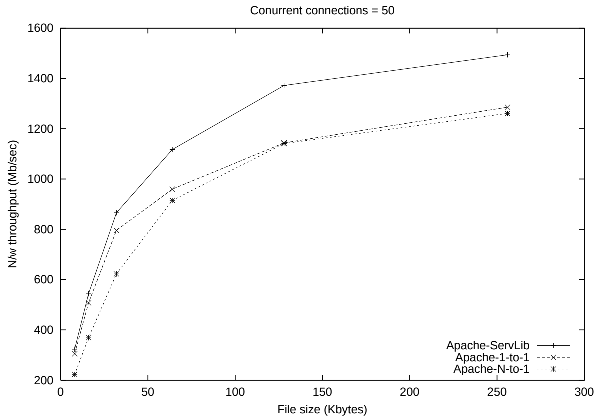

Figure 6. Network throughput with varying file sizes, 50 concurrent connections

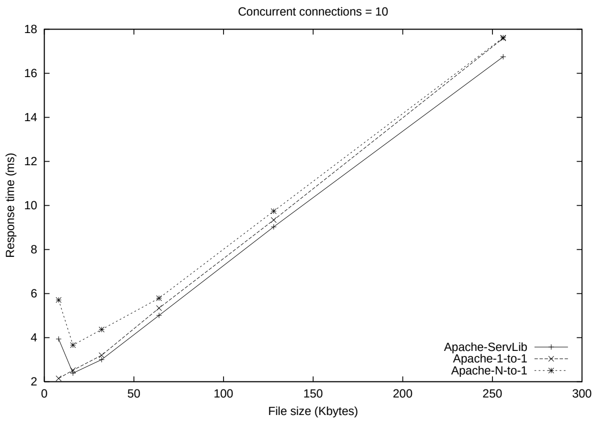

Figure 7. Response time with varying file sizes, 10 concurrent connections

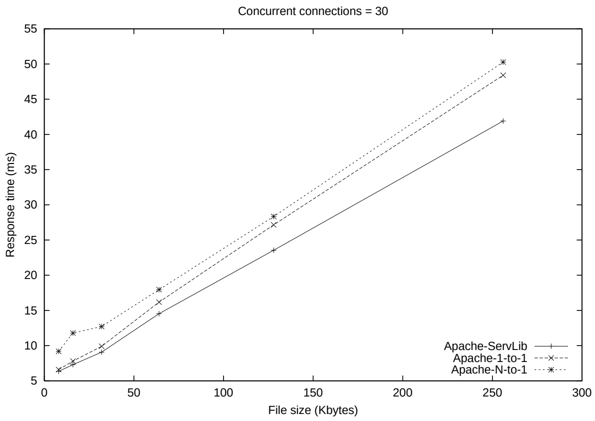

Figure 8. Response time with varying file sizes, 30 concurrent connections

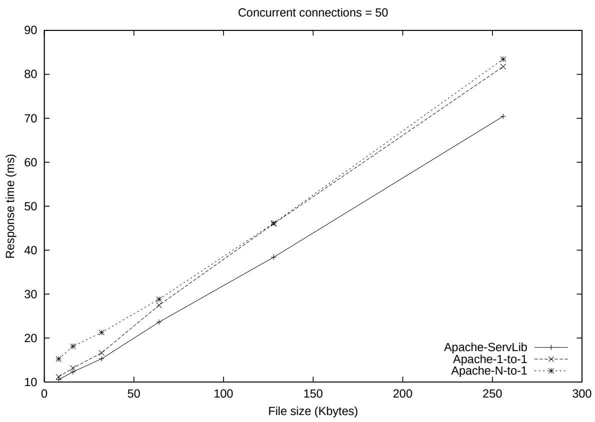

Figure 9. Response time with varying file sizes, 50 concurrent connections

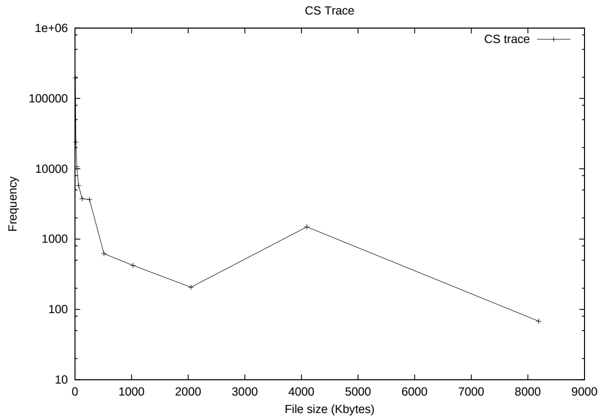

Figure 10. Request distribution for CS trace with increasing file size

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 asynchronous I/O efficiently.

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. 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%.

2) 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 in [14]. We used the Computer Science (CS) departmental traces, and web traces from NASA. Both these traces have heavy-tailed distribution (large number of small files, and small number of large files). This is shown in figure 10 and figure 11.

In figure 12 we show network throughput for CS trace for the three servers for varying number of concurrent connections, figure 13 shows the corresponding response time numbers. Figure 14 and figure 15 show network throughput numbers and response time numbers for the three servers. These figures show that *Apache-ServLib* outperforms the other servers by about 25% for trace based workloads.

Table II shows the characteristics of the CS and Nasa

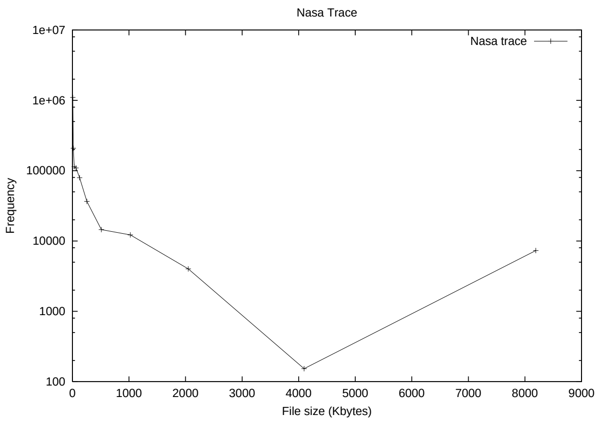

Figure 11. Request distribution for NASA trace with increasing file size

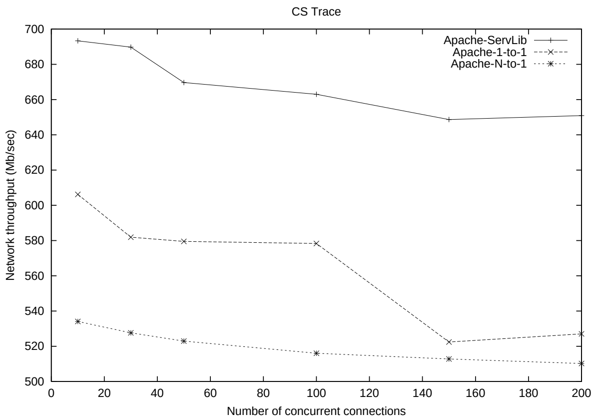

Figure 12. Network throughput for CS trace with varying number of concurrent connections

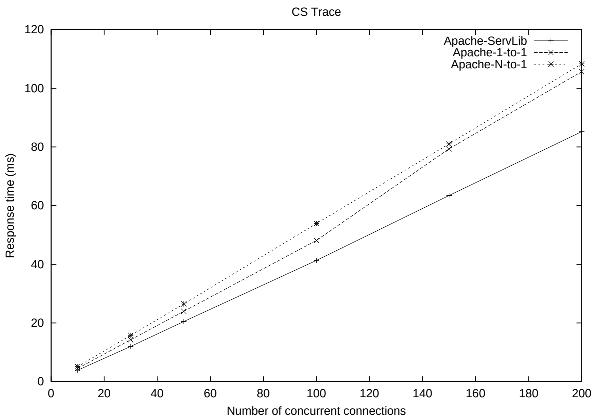

Figure 13. Response time for CS trace with varying number of concurrent connections

Web Trace	Total	Small	Medium	Large
CS	8 Gbytes	5.5%	20.2%	74.3%
Nasa	89.5 Gbytes	3.1%	24.6%	72.3%

TABLE II
WEB TRACE CHARACTERISTICS

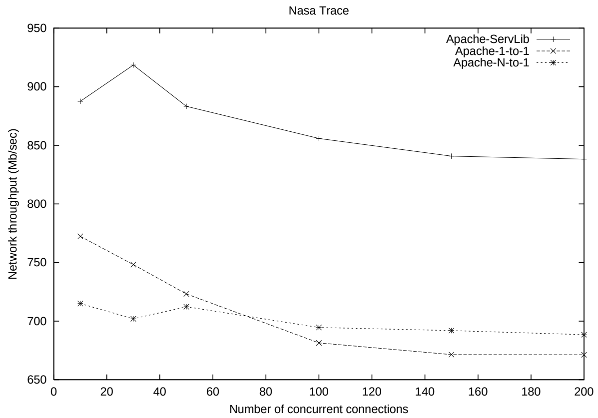

Figure 14. Network throughput for Nasa trace with varying number of concurrent connections

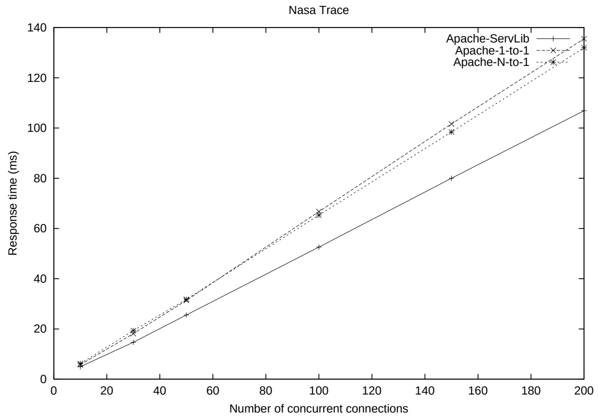

Figure 15. Response time for Nasa trace with varying number of concurrent connections

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 between 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 favour 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 Section III.

B. MySQL Results

We built MySQL 3.23.55 [10] on top of ServLib, *1-to-1 threads*, and *N-to-1 threads* separately, and compared their performance with 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*.

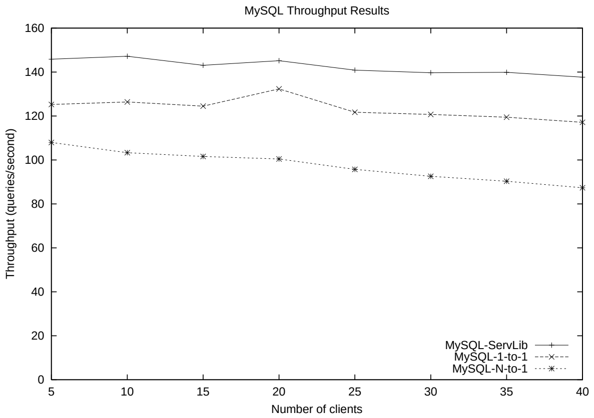

Figure 16. MySQL throughput with varying number of clients

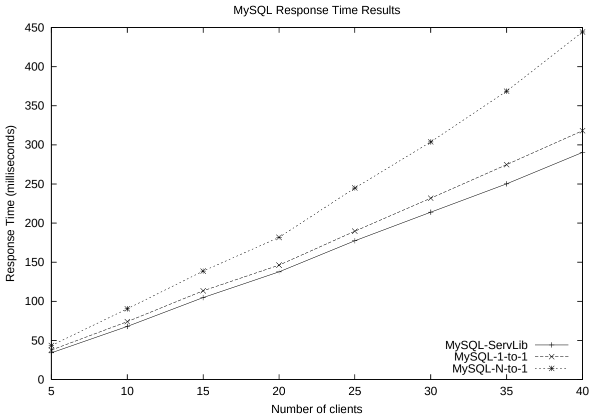

Figure 17. MySQL response times with varying number of clients

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 obtained 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 experiments, we used default configuration parameters for MySQL server.

Figure 16 shows the performance of *MySQL-ServLib*, *MySQL-1-to-1*, and *MySQL-N-to-1* on these traces with 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*. Figure 17 shows response time numbers for the three different servers for the above experiments. *MySQL-ServLib* shows a performance improvement of up to 17% for the TPC-W 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 asynchronous I/O efficiently.

V. Related Works

There has been some work on server architectures apart from multi-threading. Flash [11] is a hybrid architecture that has an event-driven core, and employs helper threads to handle blocking I/O. In SEDA [12] the server operation is broken into several stages, output and input of successive stages being connected by queues. StagedServer [7] batches the execution of similar operations arising in different server requests.

VI. Conclusion

In this paper 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 synchronous I/O vs. asynchronous I/O. We have pointed out the shortcomings of 1-to-1 threads with synchronous I/O, N-to-1 threads with asynchronous I/O, and N-to-M threads with synchronous I/O with respect to server performance. We have proposed a previously unexplored thread model of N-to-M threads with synchronous 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 10-25% 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 asynchronous I/O.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(10):53–79, February 1992.

[2] The Apache Software Foundation. Available at http://www.apache.org/.

[3] The C10K Problem. Available at http://http://www.kegel.com/c10k.html.

[4] Transaction Processing Performance Council. Available at http://www.tpc.org/.

[5] The GNU Portable Threads. Available at http://www.gnu.org/software/pth/.

[6] Multithreading in the Solaris Operating Environment. Available at http://wwws.sun.com/software/whitepapers/solaris9/multithread.pdf.

[7] James R. Larus and Michael Parkes. Using Cohort Scheduling to Enhance Server Performance. In *Proceedings of the 2002 USENIX Annual Technical Conference (Usenix 2002)*, Monterey, California, June 2002.

[8] Jonathan Lemon. Kqueue: A scalable and generic event notification facility. In *Proceedings of the 2001 USENIX Annual Technical Conference (Usenix 2001), Freenix Track*, Boston, Massachusetts, June 2001.

[9] State Threads Library. Available at http://state-threads.sourceforge.net/.

[10] The MySQL Open Source Database. Available at http://www.mysql.com/.

[11] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 USENIX Annual Technical Conference (Usenix 1999)*, pages 199–212, Monterey, California, June 1999.

[12] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP 2001)*, pages 230–243, Alberta, Canada, October 2001.

[13] Nathan J. Williams. An Implementation of Scheduler Activations on the NetBSD Operating System. In *Proceedings of the 2002 USENIX Annual Technical Conference (Usenix 2002), Freenix Track*, Monterey, California, June 2002.

[14] Hyong youb Kim, Vijay S. Pai, and Scott Rixner. Increasing Web Server Throughput with Network Interface Data Caching. In *Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, pages 239–250, San Jose, California, October 2002.