

Networking Support for Large Scale Multiprocessor Servers

David J. Yates, Erich M. Nahum, James F. Kurose, and Don Towsley*
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610, USA
{yates,nahum,kurose,towsley}@cs.umass.edu

Abstract

Over the next several years the performance demands on globally available information servers are expected to increase dramatically. These servers must be capable of sending and receiving data over hundreds or even thousands of simultaneous connections. In this paper, we show that connection-level parallel protocols (where different connections are processed in parallel) running on a shared-memory multiprocessor can deliver high network bandwidth across a large number of connections.

We experimentally evaluate connection-level parallel implementations of both TCP/IP and UDP/IP protocol stacks. We focus on three questions in our performance evaluation: how throughput scales with the number of processors, how throughput changes as the number of connections increases, and how fairly the aggregate bandwidth is distributed across connections. We show how several factors impact performance: the number of processors used, the number of threads in the system, the number of connections assigned to each thread, and the type of protocols in the stack (i.e., TCP versus UDP).

Our results show that with careful implementation connection-level parallel protocol stacks scale well with the number of processors, and deliver high throughput which is, for the most part, sustained as the number of connections increases. Maximizing the number of threads in the system yields the best overall throughput. However, the best fairness behavior is achieved by matching the number of threads to the number of processors and scheduling connections assigned to threads in a round-robin manner.

1 Introduction

Over the next several years integrated services network connectivity will continue to expand rapidly. At the same time, the information available via the Global Information Infrastructure (GII) will become more bandwidth intensive as text-only information sources are augmented with voice, video, and image data. In combination, these factors will dramatically increase the performance requirements for large scale information servers. Examples include servers for public information (e.g., digital libraries or government information sources), video-on-demand, and high-performance file systems. Servers for such applications must send or receive information at high throughput, which must be sustained (or allowed

to degrade gracefully) in the presence of large numbers of connections. In continuous-media applications, such as video-on-demand, it is also important that connections receive their “fair share” of the overall throughput in order to provide consistent quality of service to end users.

Maintaining high performance while supporting large numbers of connections is important to any information server. A natural way to accomplish this is by exploiting the inherent parallelism among connections. *Connection-level parallelism* associates the protocol processing required by connections with individual processes or threads. On a shared-memory multiprocessor, performance gains can be realized over multiple connections by executing these threads concurrently on different processors.

Previous work on connection-level parallelism can be found in [8, 25, 28, 29]. In particular, Schmidt and Suda [29] have shown good scalability of the receive-side data path in connection-level parallelism, using a thread for each connection, on a 20-processor Sun SPARCcenter 2000.

In this paper, we experimentally evaluate connection-level parallelism in a number of previously unexamined dimensions. We focus on three questions in our performance evaluation: how throughput scales with the number of processors, how throughput changes as the number of connections increases, and how fairly the aggregate bandwidth is distributed across connections. Specifically, we show the following:

- We present an implementation that allows us to vary the available concurrency in connection-level parallelism, and evaluate the impact this has on performance.
- We demonstrate that connection-level parallelism scales well with the number of processors, for both sending and receiving data.
- We show that thread per connection (where connections are assigned to threads) provides better aggregate throughput than processor per connection (where connections are assigned to processors).
- When there are more connections than available threads, we find that the best throughput is achieved by using as many threads as possible.
- We show that aggregate throughput is sustained well as the number of connections is increased by more than two orders of magnitude, thus demonstrating graceful degradation.
- We find that the distribution of aggregate throughput across connections can be unfair, and that this is directly affected by the execution time allocated to threads by the scheduler, and by the per-packet latency seen by each thread.
- Finally, we demonstrate that processor per connection provides better fairness than thread per connection.

*This research supported in part by NSF under grant NCR-9206908 and ARPA under contract number F19628-92-C-0089. David Yates is the recipient of a Motorola Codex University Partnership in Research Grant. Erich Nahum was supported by an ARPA Research Assistantship in Parallel Processing.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Our evaluation of connection-level parallelism is performed for both TCP/IP and UDP/IP protocol stacks, implemented using the *x*-kernel [11]. Our implementation runs in user space on a shared-memory Silicon Graphics (SGI) Challenge multiprocessor [7].

Several other approaches to parallelizing network protocols have also been proposed and are briefly described here; more detailed surveys can be found in [3, 10]. *Functional parallelism* decomposes functions within a protocol stack and assigns them to processing elements. Examples include [15, 16, 24]. In *layered parallelism*, protocols are assigned to specific processors, and messages are passed between layers through interprocess communication. Parallelism gains can be achieved mainly through pipelining effects, as shown in [9]. *Packet-level parallelism* associates processing with each individual packet, achieving speedup both with multiple connections and within a single connection. Examples include [3, 10, 13, 23].

The remainder of the paper is structured as follows: Section 2 provides background on connection-level parallelism. Section 3 discusses our implementation of connection-level parallelism, and describes our experiments. In section 4 we present our results. Finally, section 5 summarizes the paper.

2 Connection-Level Parallelism

Connection-level parallelism is essentially the synthesis of two ideas: the first is extending the notion of a connection through the entire protocol stack, even down to the device; the second is running these multiple connections in parallel.

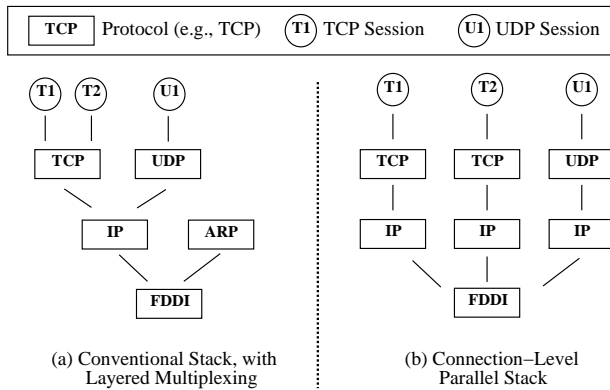


Figure 1: Protocol Stack Configurations

Figure 1(a) shows a conventional protocol stack, where protocols and connections are multiplexed on top of other protocols and connections. In this figure, a TCP connection and a UDP connection multiplex the IP protocol. On the receive path, IP demultiplexes incoming packets to the appropriate protocol.

Figure 1(b) shows a connection-level parallel protocol stack. In this example, the TCP and UDP connections extend all the way down to the device. Protocols are conceptually replicated, and multiplexing occurs at the lowest layer of the protocol stack. On the receive path, a packet is immediately demultiplexed to the appropriate connection. This demultiplexing is performed by a packet filter or classifier [1, 19, 20, 32].

Given a set of connections, threads, and processors, the assignment or mapping between them can be done in a number of different ways. Choosing a mapping defines the granularity of a connection-level parallel implementation. Previous work on connection-level parallelism [8, 25, 28, 29] has focused on relatively static assignments of connections to processes. One novel aspect of our imple-

mentation is that it allows us to vary the mapping between processors, connections, and threads. We introduce the abstraction of a *virtual processor*, which allows us to vary this assignment, and thus examine how structural choices affect performance.

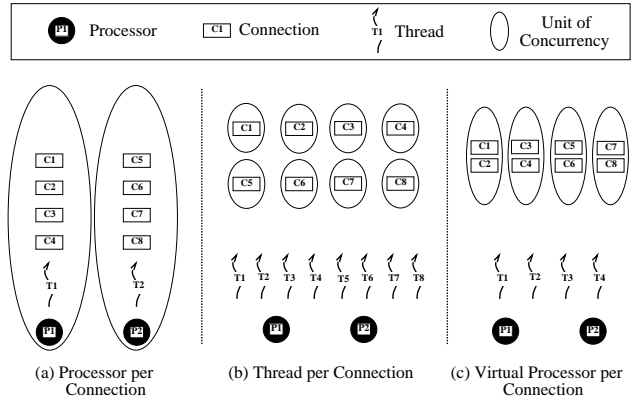


Figure 2: Approaches to Connection-Level Parallelism

The simplest and most coarse-grained approach to connection-level parallelism is *processor per connection* (PPC), which associates each connection with an individual processor. Figure 2(a) shows an example where 8 connections are mapped to 2 threads, which are each bound to their own processor. Ovals in this figure denote units of concurrency. Thus, only one thread at a time can perform protocol processing for each of the 2 sets of 4 connections in Figure 2(a). PPC has the advantage of simple implementation, but may not balance load well when some connections are more active than others.

The most fine-grained approach, *thread per connection* (TPC), makes each connection its own unit of concurrency. Figure 2(b) shows an example of TPC with 8 connections and 8 threads for sending data. These threads are free to execute on either of the two processors, and perform protocol processing on behalf of any of the connections. In practice, however, each connection is typically associated with an individual thread [8, 28]. While TPC allows easy load balancing, it may not scale well with large numbers of connections, since each thread must be allocated resources (such as a thread control block and a stack).

Our implementation facilitates varying the mapping of connections to threads and processors by using virtual processors. Connections are assigned to virtual processors, rather than physical processors or threads, in what we call *virtual processor per connection* (VPPC). Instead of a connection, the notion of a virtual processor (which corresponds to one or more connections) is extended down through the protocol stack, all the way to the device. Figure 2(c) shows an example where 8 connections are assigned to 4 virtual processors on 2 physical processors. As with thread per connection, threads are free to execute on either of the two processors, and perform protocol processing on behalf of connections assigned to any of the four virtual processors. If the number of virtual processors is the same as the number of connections, VPPC yields the same (maximal) concurrency as thread per connection. If the number of virtual processors matches the number of physical processors, (and threads are wired to physical processors), then VPPC is equivalent to PPC.

Another way of thinking of VPPC is that it extends thread per connection to support multiplexing a set of connections onto a pool of threads, where the number of threads can be varied. This becomes important if one considers a server supporting thousands of connections. Since each thread requires resources (and increases

the cost of scheduling), it is easy to imagine that TPC won't scale as well as VPPC (or even PPC) under such conditions.

At first glance, a virtual processor may seem synonymous with a thread, however, there are important differences. A virtual processor defines a unit of concurrency, not a specific thread. For example, on the send side, an application thread might be "borrowed" to run on a virtual processor in order to deliver data to the device. Conversely, on the receive side, a thread dispatched from a lower layer in the protocol stack would run on a virtual processor to deliver data to the application.

3 Implementation and Experiments

In order to study the performance of connection-level parallelism (CLP), we have implemented multiprocessor versions of the core Internet protocols (TCP, UDP and IP) over FDDI, running in a version of the *x*-kernel which was extended to support CLP. This section describes the important features of our parallelized *x*-kernel and protocols. In the last part of this section, we describe our experimental design.

3.1 Connection-Level Parallel *x*-Kernel

In our implementation of connection-level parallelism, concurrency control is accomplished by having a semaphore for each virtual processor. To execute protocol code on a virtual processor, a thread must acquire the appropriate semaphore. Once running on a virtual processor, a thread runs to completion, as in the original *x*-kernel [11].

One innovative feature of our implementation is that there are no locks on the fast path through the protocol stack, on either the send or receive side (once packets have been demultiplexed to the appropriate virtual processor). This is in contrast to earlier implementations [3, 23, 25, 29] in which data structures are locked on the fast path. We accomplish this by replicating data structures on a per virtual processor basis where possible. Thus, threads which require exclusive access to an object merely look up their current virtual processor identifier, and use it to index into an array of replicated objects. Locking is not necessary since only one thread at a time can be executing on a particular virtual processor. Examples of data structures which we are able to replicate include *x*-kernel demultiplexing hash tables, TCP send buffer free lists, and IP datagram identifier counters.

It is important to point out that not all data structures can be replicated. For example, TCP connections must be uniquely instantiated. Thus, our CLP implementation requires a packet filter mechanism to demultiplex received packets to the appropriate virtual processor for a TCP (or UDP) connection. Packet filters are becoming more popular (and efficient [1, 32]) in contemporary operating systems since early demultiplexing yields other performance gains. For example, depositing received packets directly into application buffers avoids copying data [5, 18, 30]. Our use of a packet filter, to demultiplex to the appropriate virtual processor, simply leverages this existing mechanism for an additional purpose.

3.2 Connection-Level Parallel Protocols

Our Internet protocols are all based on the uniprocessor implementations distributed with the December 1993 version of the *x*-kernel. In addition to modifying them for connection-level parallelism, we made two other important changes to these protocols. First, we updated the TCP code to be current with the Berkeley 4.4Lite implementation, excluding the RFC 1323 extensions [12]. We also replaced the Internet checksum code with the fastest available portable algorithm that we were aware of, which was from UCSD [14].

We also implemented a connection-level parallel version of UDP. Even though UDP is a connectionless protocol, we consider long term associations between a sender and receiver to be a "connection", or flow [4]. A UDP association is used to map connections to virtual processors, and provides a handle for demultiplexing, in the same fashion as TCP.

Since we did not have access to a device which matches the performance of our protocols, we replaced the lowest level drivers in the *x*-kernel with in-memory device drivers for both the TCP and UDP protocol stacks. The drivers emulate an FDDI interface capable of operating at memory speeds (i.e., much faster than 100 Mbps), and support the FDDI maximum transmission unit (MTU) of slightly over 4KB. This approach is similar to the those used by other researchers [3, 10, 17, 23, 29].

In our experiments, we assume that a connection always has data to send (i.e., each connection is an infinite data source). Therefore, the drivers act as peer senders or receivers, producing or consuming packets as fast as possible. This simulates the behavior of a collection of clients sending or receiving data over simplex connections routed through an error-free network. To minimize experimental perturbation, the receive-side driver uses preconstructed packet templates. To simulate an actual sender, this driver should compute the Internet checksum and update the template (for TCP). Instead, a dummy checksum value is left in the template and the TCP or UDP receivers compute the checksum, but ignore the result.

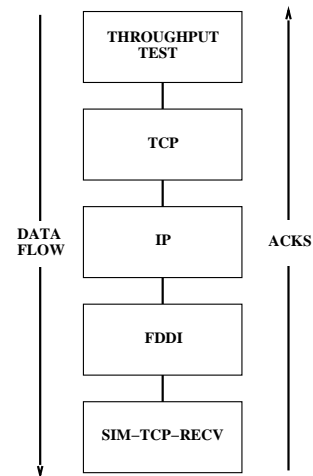


Figure 3: TCP Send-Side Configuration

Figure 3 shows an example of a test configuration. The example is of a send-side TCP throughput test, where a simulated TCP receiver sits below the FDDI layer. The simulated TCP receiver generates acknowledgements for packets sent by the actual senders. The driver acknowledges every other packet for each connection, thus mimicking the behavior of Berkeley TCP when communicating with itself as a peer. Since spawning threads is expensive in user space in IRIX, the driver "borrows" the stack of a calling thread to send an acknowledgement back up.

The TCP receive-side driver (i.e., simulated TCP sender) produces packets in-order for consumption by the actual receiver. Both simulated TCP drivers also perform their respective roles in setting up a connection.

3.3 Experimental Design

To evaluate our CLP implementation, we ran experiments for both TCP/IP and UDP/IP protocol stacks. In addition to the protocol

stack being measured, several other parameters define an experiment: the number of physical processors, virtual processors, and connections; whether the multiprocessor is the sender or receiver; and whether or not the checksum is computed.

The throughput data from each experiment (both aggregate and per-connection) are calculated from the average of 12 runs, where a run consists of measuring the steady-state throughput of packets carrying 4KB of user data for 45 seconds, after an initial warmup period of 45 seconds or more. For all data we present in figures, we show 90% confidence intervals. During experiments, we isolated our machine as much as possible by disallowing other user activity and removing all non-essential daemons. We now discuss some of the more subtle aspects of our experiments.

Our experiments use a thread for each virtual processor to produce and send or receive packets. The protocol processing associated with connections assigned to the same virtual processor is performed in a round-robin manner among connections, with the unit of work being the processing needed to bring a single packet (or a burst of packets) up or down the protocol stack. Between processing each unit of work, the virtual processor is yielded to allow other tasks (e.g., TCP timers) to run on that virtual processor. However, the scheduling of threads running on virtual processors is controlled by the IRIX operating system.

In some of our results, we present average per-packet latency (as well as throughput) over the 45-second sampling period, for each individual thread. We compute latency using Little’s law from the measured throughput for the thread, the actual wall-clock time, and the user time given to the thread by the operating system. The user time for a thread represents the time spent executing, and therefore processing packets, in user space. This is obtained by making a `getrusage()` system call at the beginning and end of the sampling period.

It is well known that memory reference behavior can be crucial in determining how a system performs when running on a multiprocessor. We control the memory reference behavior of our protocols in several ways. First, to capture the cost of loading packets with user data, each sending connection copies data from a statically allocated 4KB page to a page-aligned *x*-kernel communication buffer. For receiving connections, the copy is performed in the reverse direction. Other than this copy, we do not attempt to capture any computation or memory references associated with an application.

For send-side experiments, we always cache communication buffers in LIFO lists. These lists are maintained per virtual processor, and therefore require no locks. This technique has been shown to improve throughput in protocol stacks running on a uniprocessor [6]. This means that our send-side experiments measure either a cache-to-cache or a memory-to-cache copy depending on the fate of the “application” data buffer in the caches.

For receive-side experiments, it is unreasonable to assume that user data will be in the cache, so we wire threads to processors and force any user data referenced for the first time to be fetched from memory and not from processor caches. To implement this, threads allocate receive-side buffers (pages) from a per physical processor circular list which is twice as large as the second-level caches on our multiprocessor. Unfortunately, managing this list introduces locking overhead, which unfairly penalizes throughput experiments where there are more virtual than physical processors. We therefore only present receive-side results for processor per connection.

In experiments where we vary the number of processors, we use IRIX’s `pset` facility, which restricts the threads in a process group to running on a subset of processors.

4 Results

We report results from an evaluation of our connection-level parallel implementation along three different dimensions. First, we examine how aggregate throughput scales as the number of processors increases. Second, we examine how throughput is affected by the number of connections. Finally, we investigate how fairly the total throughput is distributed across both moderate and large numbers of connections.

To quantify fairness, we measure the throughput seen by each virtual processor, and compute percentiles between the maximum and minimum throughputs. Since our fairness experiments use a single thread per virtual processor, our results refer to per-thread throughput, or throughput per thread. In contrast, we refer to aggregate (or total) throughput as just throughput.

4.1 Throughput Scalability with Respect to Processors

To evaluate how throughput scales with the number of processors we ran experiments varying the number of physical processors from 1 to 20, and bound a single connection to each processor using our processor per connection implementation.

Figure 4 shows throughput versus number of processors (and connections) for both TCP and UDP protocol stacks. In these experiments buffers are “written” on the send side by repeatedly copying a 128 byte (the size of a second-level cache line) memory buffer into the 4KB of user data, and “read” on the receive side by repeatedly copying portions of the user data into a 128 byte buffer. The checksum is also computed on packets. Virtual processor per connection (VPPC) and thread per connection (TPC) data are omitted from Figure 4 since all three schemes yield essentially the same performance in this scenario. Note that throughput is highest for UDP send-side processing and lowest for TCP receive-side processing.

Figure 5 shows speedup corresponding to the throughput data presented in Figure 4. Here speedup is throughput normalized by the corresponding single processor throughput. Note that in all four experiments speedup is linear, and in the range of 14 to 17 at 20 processors.

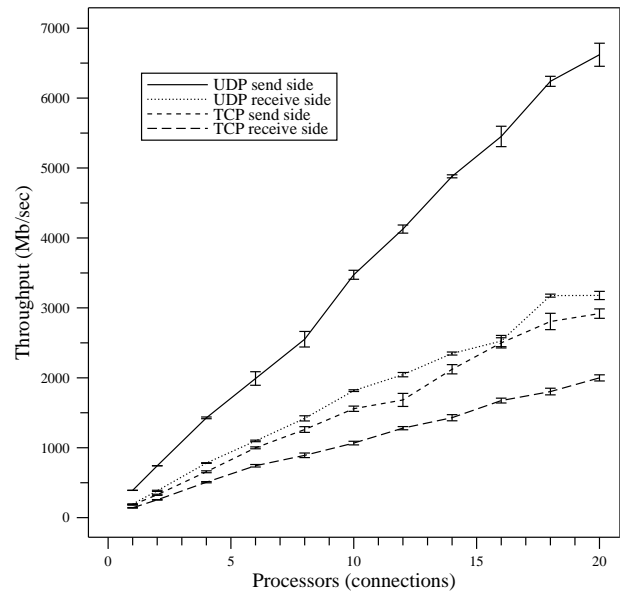


Figure 4: UDP and TCP Aggregate Throughput for Processor per Connection.

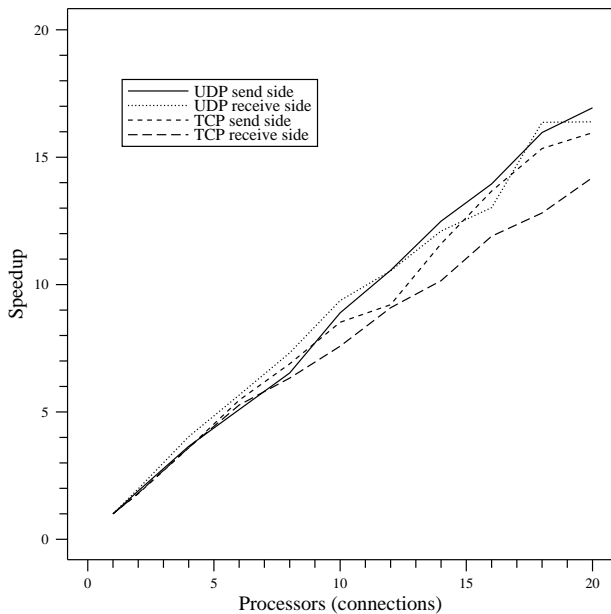


Figure 5: UDP and TCP Speedup for Processor per Connection.

Figures 4 and 5 give compelling evidence that connection-level parallel protocol stacks (even with TCP) can scale with the number of processors. These results are consistent with those reported in [29], which focus on the receive side for thread per connection parallelism. We extend their results in two significant ways. First, our UDP and send-side TCP results are new. Second, our receive-side results include the cost of fetching user data from memory, over the system bus, the first time it is touched.

The data presented in Figures 4 and 5 were all gathered on a 20-processor 150 MHz SGI Challenge. Unfortunately, we only had short-term access to this machine, so the throughput data presented here should not be directly compared with data elsewhere in the paper, which were gathered on a 12-processor 100 MHz Challenge.

4.2 Throughput Sustainability with Respect to Connections

To measure the impact of adding connections on the overall throughput, we ran experiments varying the number of connections running on a 12-processor machine from 12 to 3072. We also varied the granularity of parallelism from coarse (processor per connection) to fine (thread per connection). We wanted to determine whether increasing parallelism (by multiplexing fewer connections onto more virtual processors) is beneficial, because of increased scheduling flexibility and improved locality of memory reference, or detrimental, because of the overhead of using more virtual processors. Comparing the throughput of different implementations with the same number of connections answers this question.

In these experiments, threads are not wired to processors. Thus, the results presented here are for thread per connection (TPC) or virtual processor per connection (VPPC), even when the number of virtual processors is the same as the number of physical processors. We found that in all experiments wiring threads to processors decreased the aggregate throughput. In some configurations this decrease was as much as 50%. Furthermore, the version of IRIX we used schedules threads for cache affinity [2], which has been shown to benefit connection-level parallelism [26, 27].

Figure 6 shows the throughput of our multiprocessor sending data on 12 to 3072 TCP connections, where the checksum is computed. Figure 7 shows the corresponding results for our UDP protocol stack, without checksumming. Where VPPC data are shown,

the number of virtual processors is indicated by V in the legend. We also conducted experiments using between 12 and 384 virtual processors (not shown in these figures) and found these configurations to yield throughput values which generally lie between those shown. It is worth noting that the curves for thread per connection stop at 384 connections since IRIX only allows a maximum of 512 threads in a process group to share an address space. We were unable to use every thread for sending (or receiving) data, and therefore limited the number of protocol processing threads to 384.

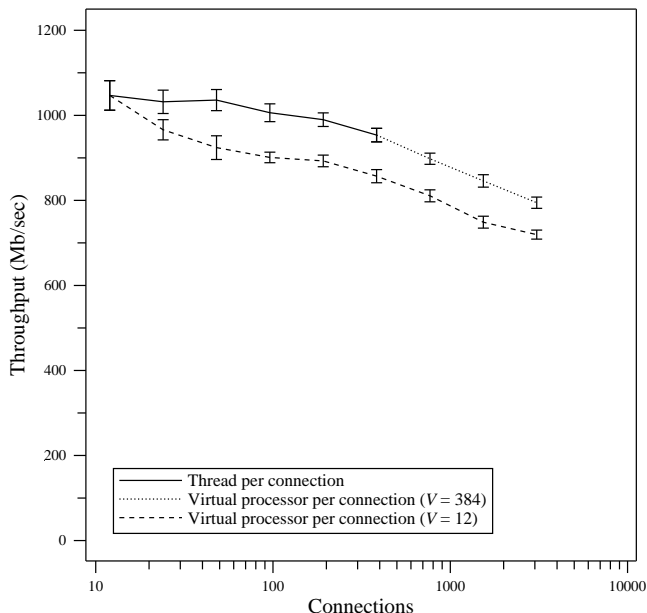


Figure 6: TCP Send-Side Aggregate Throughput from 12 to 3072 Connections.

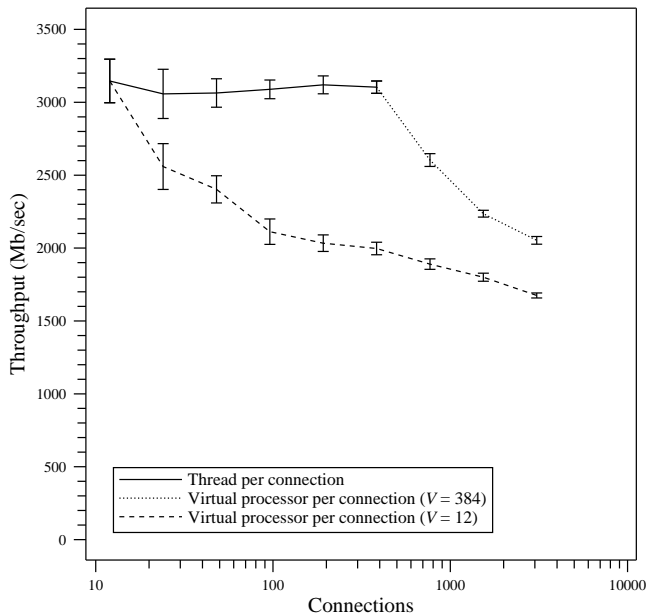


Figure 7: UDP Send-Side Throughput from 12 to 3072 Connections.

Several conclusions can be drawn from the data in Figures 6 and 7. The throughput for both TCP and UDP is sustained surprisingly

well as the number of connections increases. For example, using 384 virtual processors, the overall throughput for TCP send-side processing only degrades from 960 Mbps to 800 Mbps as the number of connections increases from 384 to 3072. Furthermore, with more than 12 connections using the maximum number of virtual processors (i.e., using TPC or VPPC with 384 virtual processors) yields the best throughput. Using more virtual processors also has the advantage that it pushes out the point at which throughput starts to degrade. Specifically, for 12 virtual processors, the throughput starts to degrade immediately (going from 12 to 24 connections), while for TPC, the throughput does not degrade until there are 96 active connections. This is encouraging since under conditions where connection sources do not have an infinite amount of data to send (i.e., are occasionally idle), the scheduling flexibility gained by increasing the number of virtual processors widens the performance improvement over that achieved by using fewer virtual processors [27, 31].

Figures 6 and 7 also show a surprising result: the throughput of TCP is less sensitive to an increase in the number of connections than that of UDP. We suspect that this is because the code path for sending packets over UDP connections is shorter than the code path for TCP connections, making the UDP results more sensitive to conflicts in the caches on the multiprocessor. This phenomenon has been observed by others on a uniprocessor [21].

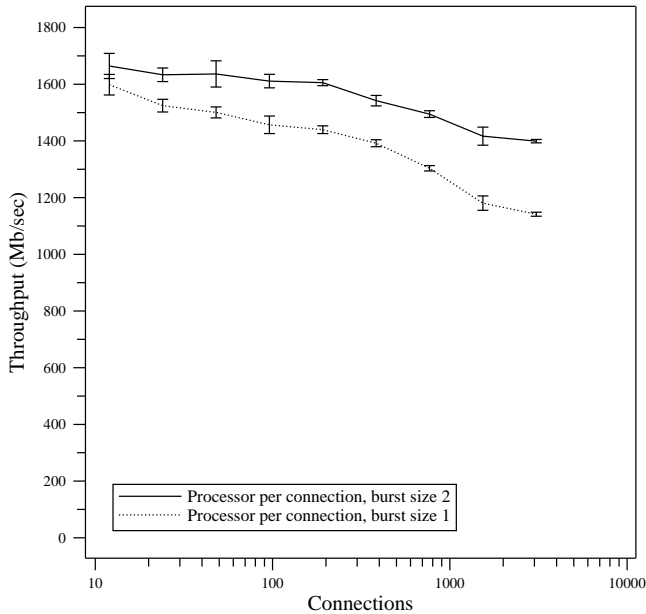


Figure 8: UDP Receive-Side Throughput from 12 to 3072 Connections.

So far we have presented results only for experiments where the multiprocessor is the sender. However, for applications such as high-performance file service (which typically run on top of UDP), it is also interesting to examine the case where the multiprocessor is the receiver. When clients write to files on the server, it is likely that consecutive packets arrive quickly enough that their processing can be amortized over a single virtual processor yield. If each receiving thread yields the virtual processor after processing two packets, the unit transferred is 8KB, a typical disk block size for networked file systems. We refer to processing multiple packets between a single virtual processor yield as processing packets in “bursts”. Thus, a single packet is processed if the burst size is one, and a pair of packets are processed if the burst size is two, etc. Amortizing what are normally per-packet costs over multiple packets is a well known technique for improving networking performance.

Figure 8 shows processor per connection (PPC) throughput for a UDP receiver, where the burst size used by each connection is either one or two, and the checksum is not computed. Recall that the mechanism to ensure that application data is fetched from memory as it is first touched adds sufficient overhead that comparison with a greater number of virtual processors is difficult. However, for up to 96 virtual processors the results are all comparable to those shown for PPC. Comparing the dotted line in Figure 8 with the dashed virtual processor per connection curve in Figure 7 (for $V = 12$) indicates that the UDP receive side throughput degrades more gracefully than the send side, again suggesting that the results in Figure 7 are in part due to the impact of cache behavior on UDP’s send-side code path. Figure 8 also shows that throughput improves up to 20% by processing received packets in pairs.

We also investigated the benefits of amortizing multiple packet transfers over a single virtual processor yield (i.e., processing packets in bursts) on the send side. This strategy is applicable in any situation where the size of data objects being transferred is large relative to the MTU for the network (e.g., image files). However, this strategy may not be applicable if the latency of sending data objects is crucial (e.g., packet voice or video). Since some simple experiments showed that this technique yields diminishing returns, we fixed the burst size at 16 packets (64KB) for our send-side experiments.

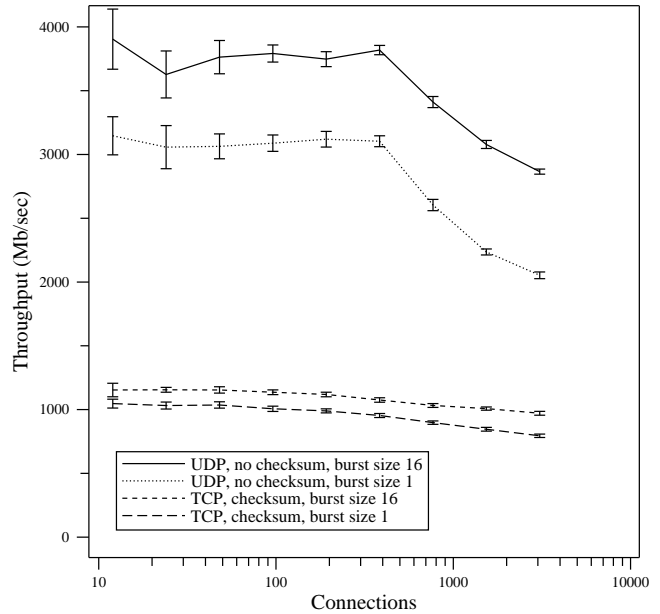


Figure 9: Impact of Processing Packets in Bursts on Send-Side Aggregate Throughput.

Figure 9 compares throughput for the maximum number of virtual processors (i.e., TPC or VPPC with 384 virtual processors) using a burst size of 16 with our previous results for a burst size of one (see Figures 6 and 7). Note that the throughput improves by 10-20% for TCP and up to 40% for UDP. For a burst size of 16, the gap between the throughput curves for 12 and 384 virtual processors in Figure 6 is narrowed. In fact, it becomes insignificant at 384 or more connections for TCP. This suggests that using bursts when there are a large number of active connections decreases the performance impact of the number of virtual processors in the system.

4.3 Fairness

In the results we have discussed so far, the throughput measure of interest has been the overall, *aggregate* throughput of all of the

connections. Another important measure of performance, however, is the throughput seen by *individual connections*. Recall that our experiments use a single thread for each virtual processor to send or receive packets. Furthermore, processing of packets for connections assigned to the same virtual processor is performed in a round-robin manner. Thus, any difference in throughput for connections assigned to the same virtual processor is at most one packet's worth of data over the sampling interval (i.e., 45 seconds). We exploit this fact to present fairness results for per-thread throughput rather than per-connection throughput. In describing these results, we will use the term "thread" to mean the thread performing protocol processing on a virtual processor.

Clearly, the extent to which a *thread* receives its fair share of the total throughput is influenced by the manner in which threads are scheduled by IRIX. With N threads, one might be tempted to assume that each thread receives approximately $1/N$ (i.e., roughly its "fair share") of this aggregate throughput, particularly when the per-thread throughput is measured over a sufficiently long period of time. As we will see, however, our results show there can be significant differences in the per-thread throughput.

We quantify whether our protocol implementations are distributing aggregate throughput fairly by measuring the throughput seen by each thread and computing percentiles of the per-thread throughputs, between the maximum and minimum values. We report here on results from two configurations, consisting of UDP send-side processing without checksumming, with both a moderate and large number of connections. However, the send-side results for TCP also show the same trends.

Figure 10 shows the difference in throughput that threads see when sending data for 384 connections. The curves plot percentiles of per-thread throughput as a function of the number of threads. The throughput shown is normalized by the average throughput seen by all threads. The dotted line in Figure 10, for example, shows the normalized throughput such that 75% of the threads receive a throughput of less than or equal to the y-axis value. The spread between the curves is thus a rough indication of the distribution of per-thread throughput seen by the number of threads indicated by the x-axis.

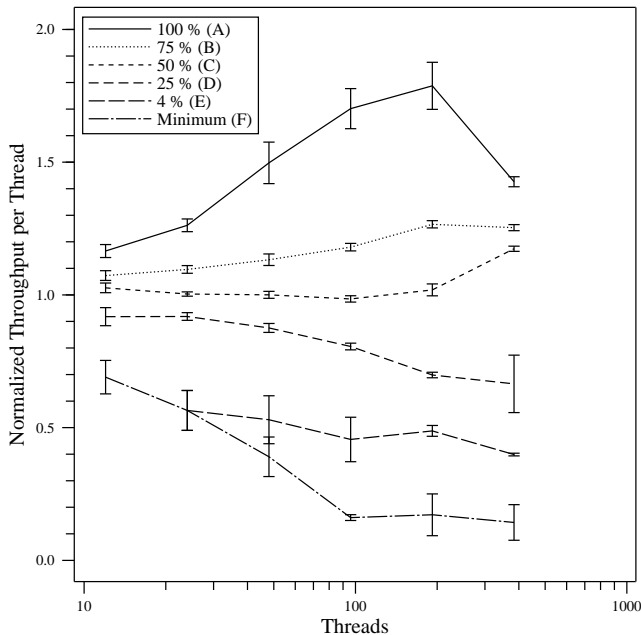


Figure 10: UDP Send-Side Throughput per Thread for 384 Connections.

Figure 10 illustrates several interesting aspects of the server's behavior. First, we note that there can be a significant difference in the throughput seen by individual threads. For example, with 96 threads, the thread receiving the highest throughput receives approximately 10 times the throughput received by the thread with the lowest throughput. We found this quite surprising since with 96 threads, 1500 scheduling time quanta (in a 45-second interval), and 12 processors, each thread should have received approximately 190 time quanta – long enough to have averaged out the performance differences one might expect in the throughputs during the individual 30 ms time quanta.

Figure 10 also illustrates that the differences in throughput per thread are minimized when there are only 12 threads. Thus, from a fairness standpoint, a smaller number of threads is to be preferred. Recall from our discussion of Figure 7, however, that a smaller number of threads results in lower throughput. For example, with 384 connections, the aggregate throughput (as indicated in Figure 7) is 2.0 Gbps and 3.1 Gbps for 12 and 384 threads, respectively. Thus, while 12 threads provide a fairer allocation of throughput among threads, the aggregate throughput is lower.

We were surprised by the magnitude of the unfairness shown in Figure 10, and wanted to understand its cause. As we will see, the user time given to threads, and the latency processing packets, both appear to play a role in the unfair distribution of aggregate throughput to individual threads.

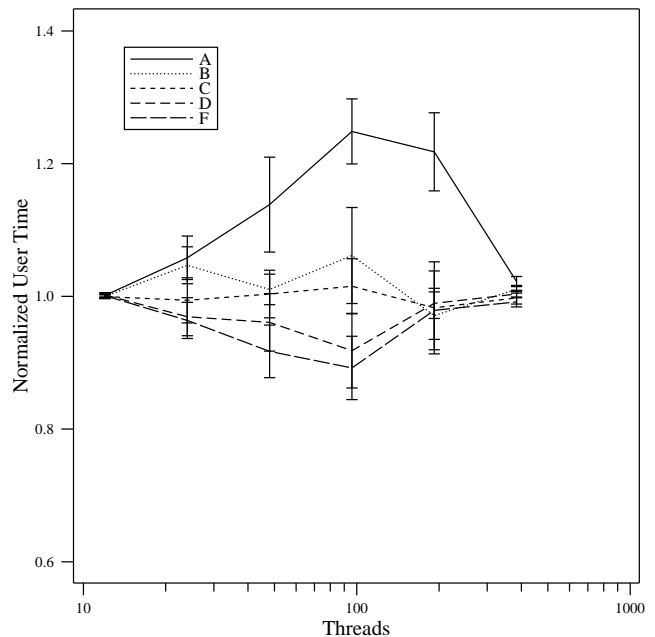


Figure 11: UDP Send-Side User Time for 384 Connections.

Figure 11 shows the user time given to the threads corresponding to the quartiles in Figure 10 (labelled A through D, and F). The user time shown is normalized by the average user time given to all threads. We stress that Figure 11 is *not* a quartile plot. Hence, curves in this figure can cross, as seen for threads B and D at 192 threads. The similarity in shape and slope of the curves in Figures 10 and 11, at least up to 96 threads, suggests that the scheduler has some determining effect on fairness in this region. However, this is no longer the case with more than 96 threads, since the user time for threads B, C, D and F are all similar in this region, yet deliver dramatically different throughput (as shown in Figure 10).

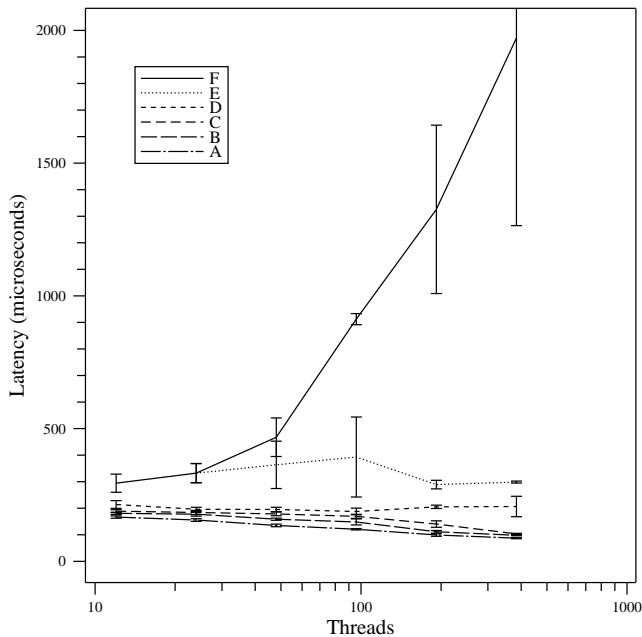


Figure 12: UDP Send-Side Latency for 384 Connections.

Figure 12 shows the average per-packet latency seen by the threads in Figure 10 (labelled A through F). Again, note that Figure 12 is not a percentile plot. The most noticeable feature of Figure 12 is the dramatic difference in latency between threads, when there are a moderate or large number of them. For example, thread E (which corresponds to the 4th percentile in throughput) and F (which corresponds to the minimum throughput), show differences of a factor of 2.3 to 6.3, at 96 threads and above. This suggests that the distribution of per-packet latency seen by the threads has a heavy tail in this region. One possible explanation for this is the memory reference behavior during protocol processing. These differences in latency have a dramatic affect on the per-thread throughput, especially when there are more than 96 threads.

We were also interested in examining the fairness behavior of our protocol stack as the number of connections increased from a moderate to large size. To compare our results with those described above, we used the same protocol stack (i.e., UDP send-side, without checksumming), the same range of threads (i.e., 12 to 384), and again measured values over a 45-second interval. Figures 13 through 15 show the fairness behavior of our server sending data over 3072 UDP connections. We discuss these figures in order, as before.

Figure 13 shows percentiles of normalized per-thread throughput as a function of the number of threads, for 3072 connections. Note that the differences in per-thread throughput in this figure are still significant (i.e., are up to a factor of 5), but are less than those in Figure 10, which exceed a factor of 10. In common with our results for fewer connections, Figure 13 also illustrates that the differences in per-thread throughput are minimized when there are only 12 threads. However, this improvement in fairness behavior again comes at the cost of aggregate throughput. For example, when using 384 threads (at the extreme right of Figure 13) the aggregate throughput from Figure 7 is 2.1 Gbps. However, for 12 threads (at the far left of Figure 13) the aggregate throughput is 1.7 Gbps.

Figures 14 and 15 again show the effect of user time given to threads, and average per-packet latency, on the per-thread throughput. Figure 14 shows the normalized user time given to the threads corresponding to the quartiles in Figure 13 (labelled A through D, and F). The similarity in shape and slope of the curves in Figures 13 and 14, at least up to 96 threads, again suggests that the scheduler

has some determining effect on fairness in this region. However, for more than 96 threads, the scheduling behavior again has less impact on fairness. Figure 15 shows the average per-packet latency seen by the threads in Figure 13 (labelled A through F). The most interesting feature of Figure 15 is the drop in average per-packet latency for the thread which delivers the lowest throughput (F), when compared with the corresponding curve in Figure 12. For example, with 384 connections and the same number of threads, the average per-packet latency for thread F is roughly 2000 microseconds (see Figure 12). For 3072 connections and 384 threads, the per-packet latency for thread F is only 625 microseconds. In fact, over the range of threads we examined, the gap between the latencies of the thread delivering the 25th percentile throughput (D) and minimum throughput (F), is narrowed considerably as the number of connections is increased. Thus, while increasing the number of connections assigned to the same number of threads reduces aggregate throughput, it can improve the worst-case latency behavior.

5 Conclusion

In this paper, we have shown that connection-level parallel protocol stacks scale well with the number of processors, and deliver high throughput which is, for the most part, sustained as the number of connections increases. For moderate to large numbers of connections, the number of threads in the system and the number of connections assigned to each thread are the key factors in determining performance on a multiprocessor. Choosing which connection-level parallel implementation to use should depend on whether total throughput or fairness is more important.

Our results show that thread per connection parallelism yields the best aggregate throughput. However, thread per connection may not be feasible if the number of connections exceeds the number of threads that can be reasonably supported by a particular multiprocessor. In this case, the best throughput is obtained by using virtual processor per connection and maximizing the number of threads in the system.

We also examined the fairness behavior of our implementation by considering both the range and value of per-thread throughputs and latencies. Our results show that matching the number of threads to the number of physical processors, while scheduling connections assigned to each thread (or virtual processor) in a round-robin manner, yields the best fairness behavior. In this implementation, the range of throughputs seen by different threads for a UDP-based protocol stack is decreased by a factor of five or more. However, an improvement in fairness may come at the cost of aggregate throughput.

If an application needs maximal throughput from a multiprocessor protocol stack, amortizing per-packet costs over multiple packets is worthwhile. In our implementation, we showed that yielding the virtual processor after a burst of a few packets rather than after every packet improved throughput by up to 40%. Our experiments to date show that processing packets in bursts does not significantly impact fairness over long intervals of time (45 seconds or more). However, one would expect that this would not be true for shorter intervals of time (e.g., tens of milliseconds). We plan on investigating this in future research.

Acknowledgments

We are indebted to Larry Peterson and the *x*-kernel group at the University of Arizona for extending their help and hospitality. Ed Menze and Hilarie Orman answered countless questions. Franklin Reynolds and Franco Travostino of OSF; and Bill Fisher, Neal Nuckolls, Greg Chesson, and Bill Nowicki of SGI gave us excellent suggestions for improving this work. We are grateful to Jim Salehi and Eric Brown for many hours of lively discussion about connec-

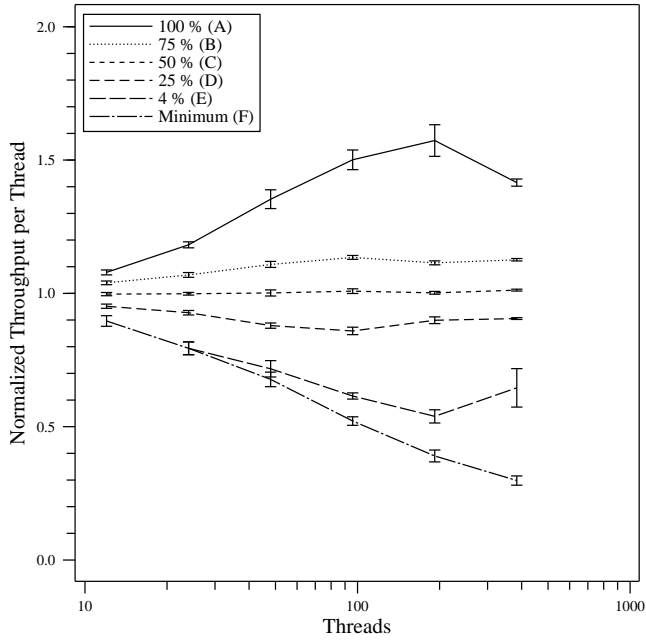


Figure 13: UDP Send-Side Throughput per Thread for 3072 Connections.

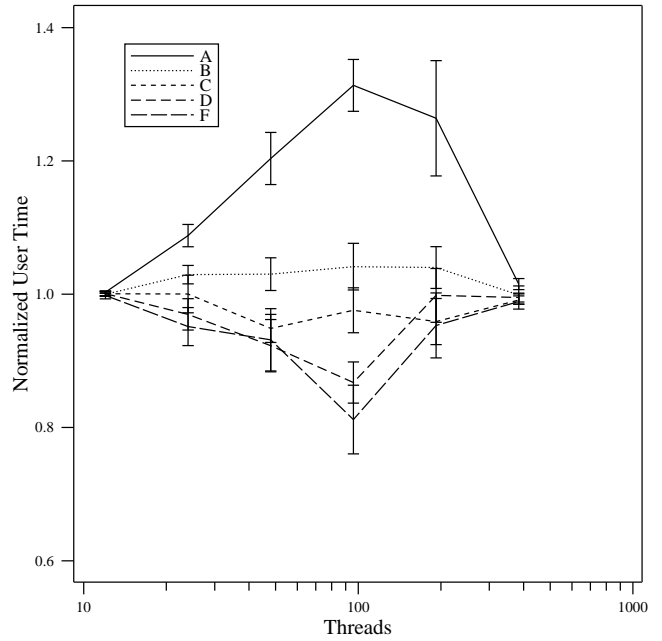


Figure 14: UDP Send-Side User Time for 3072 Connections.

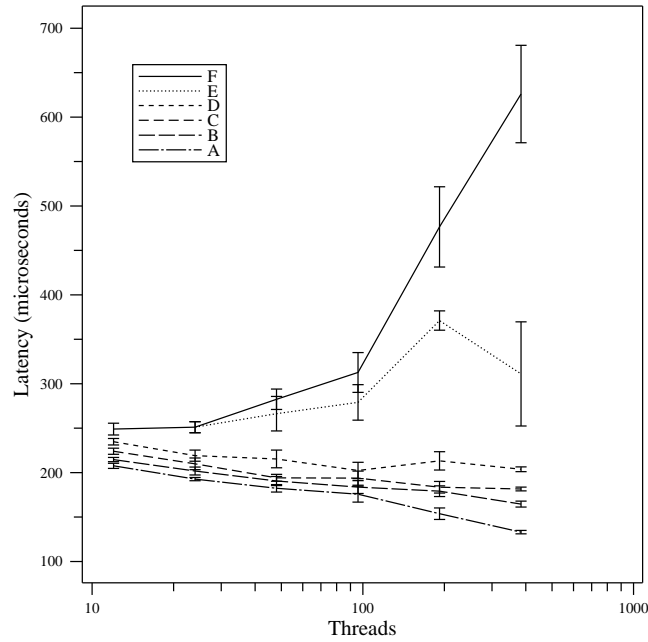


Figure 15: UDP Send-Side Latency for 3072 Connections.

tion-level parallelism. The anonymous reviewers greatly improved the content and presentation of this paper. Thanks also to Whitney Harris and Paul Sorenson for arranging access to a 20-processor machine.

References

- [1] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: a pattern-based packet classifier. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 115–123, Monterey, CA, Nov. 1994.
- [2] J. M. Barton and N. Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 24–40, Santa Barbara, CA, Apr. 1995.
- [3] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 74–83, San Francisco, CA, Sept. 1993. ACM.
- [4] K. C. Claffy, H.-W. Braun, and G. C. Polyzos. Internet traffic flow profiling. Technical Report UCSD Report CS93-328, SDSC Report GA-A21526, University of California at San Diego, March 1994.
- [5] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 2–13, London, England, Aug. 1994. ACM.
- [6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville, NC, Dec. 1993.
- [7] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, Silicon Graphics Inc., Mt. View, CA, May 1994.
- [8] A. Garg. Parallel STREAMS: a multi-processor implementation. In *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., Jan. 1990.
- [9] D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. C. Williamson. High-speed parallel protocol implementation. *First IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, pages 165–180, May 1989.
- [10] M. W. Goldberg, G. W. Neufeld, and M. R. Ito. A parallel approach to OSI connection-oriented protocols. *Third IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, pages 219–232, May 1992.
- [11] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.
- [12] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. In *Network Information Center RFC 1323*, pages 1–37, Menlo Park, CA, May 1992. SRI International.
- [13] N. Jain, M. Schwartz, and T. R. Bashkow. Transport protocol processing at Gbps rates. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 188–199, Philadelphia, PA, Sept. 1990. ACM.
- [14] J. Kay and J. Pasquale. Measurement, analysis, and improvement of UDP/IP throughput for the DECStation 5000. In *Proceedings of the Winter 1993 USENIX Conference*, pages 249–258, San Diego, CA, 1993.
- [15] O. G. Koufopavlou and M. Zitterbart. Parallel TCP for high performance communication subsystems. In *Proceedings of the Global Telecommunications Conference (GLOBECOM)*, pages 1395–1399, 1992.
- [16] T. F. La Porta and M. Schwartz. A high-speed protocol parallel implementation: Design and analysis. *Fourth IFIP TC6.1/WG6.4 International Conference on High Performance Networking*, pages 135–150, Dec. 1992.
- [17] B. Lindgren, B. Krupczak, M. Ammar, and K. Schwan. An architecture and toolkit for parallel and configurable protocols. In *Proceedings of the International Conference on Network Protocols*, pages 234–242, San Francisco, CA, Mar. 1993.
- [18] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, Dec. 1993.
- [19] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, Jan. 1993.
- [20] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings 11th Symposium on Operating System Principles*, pages 39–51, Austin, TX, November 1987.
- [21] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994.
- [22] E. Nahum, D. J. Yates, S. O'Malley, H. Orman, and R. Schroepfel. Parallelized network security protocols. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, CA, Feb. 1996.
- [23] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 125–137, Monterey, CA, Nov. 1994.
- [24] A. N. Netravali, W. D. Roome, and K. Sabnani. Design and implementation of a high-speed transport protocol. *IEEE Transactions on Communications*, 38(11):2010–2024, Nov. 1990.
- [25] D. Presotto. Multiprocessor streams for Plan 9. In *Proceedings of the United Kingdom UNIX Users Group*, Jan. 1993.
- [26] J. D. Salehi, J. F. Kurose, and D. Towsley. The performance impact of scheduling for cache affinity in parallel network processing. In *International Symposium on High Performance Distributed Computing (HPDC-4)*, Pentagon City, VA, Aug. 1995.
- [27] J. D. Salehi, J. F. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor networking. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, page 2C.2, San Francisco, CA, Mar. 1996.
- [28] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan. Pitfalls in multithreading SVR4 STREAMS and other weightless processes. In *Proceedings of the Winter 1993 USENIX Conference*, pages 85–96, San Diego, CA, Jan. 1993.
- [29] D. C. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, pages 624–633, Boston, MA, Apr. 1995.
- [30] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 64–73, San Francisco, CA, Sept. 1993. ACM.
- [31] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. Technical Report CMPSCI 95-83 (in preparation), URL = ftp://gaia.cs.umass.edu/pub/Yate95:Networking.ps.Z, University of Massachusetts, Amherst, MA, Dec. 1995.
- [32] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, Jan. 1994.