Appears in "Proceedings of the First Symposium on Operating Systems Design and Implementation," Usenix Association, November 1994.

Performance Issues in Parallelized Network Protocols

Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley[*]

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

Parallel processing has been proposed as a means of improving network protocol throughput. Several different strategies have been taken towards parallelizing protocols. A relatively popular approach is *packet-level parallelism*, where packets are distributed across processors.

This paper provides an experimental performance study of packet-level parallelism on a contemporary shared-memory multiprocessor. We examine several unexplored areas in packet-level parallelism and investigate how various protocol structuring and implementation techniques can affect performance. We study TCP/IP and UDP/IP protocol stacks, implemented with a parallel version of the *x*-kernel running in user space on Silicon Graphics multiprocessors.

Our results show that only limited packet-level parallelism can be achieved within a single connection under TCP, but that using multiple connections can improve available parallelism. We also demonstrate that packet ordering plays a key role in determining single-connection TCP performance, that careful use of locks is a necessity, and that selective exploitation of caching can improve throughput. We also describe experiments that compare parallel protocol performance on two generations of a parallel machine and show how computer architectural trends can influence performance.

1 Introduction

Parallel processing has been proposed as a means of improving network protocol throughput. Two trends motivate the use of parallelism in network processing. First, network bandwidths are increasing by orders of magnitude, with the advent of technologies such as ATM. Second, shared-memory multiprocessors are becoming more common, as shown by recent vendor introductions [1, 8, 9]. There is thus an opportunity to exploit the potential of parallelism in network protocol processing, and this has become a growing area of research.

The approach we study here is that of *packet-level parallelism*, sometimes referred to as thread-per-packet or processor-per-message parallelism. Originally proposed by Hutchinson and Peterson in the *x*-kernel [14], this approach distributes packets across processors, achieving speedup both with multiple connections and within a single connection. Packets can be processed on any processor, maximizing flexibility and utilization. Other systems using this approach include [5, 11].

Several other approaches to parallelism have also been proposed and are briefly described here; more detailed surveys can be found in [5, 11]. In *layered parallelism*, protocols are assigned to specific processors, and messages passed between layers through interprocess communication. Parallelism gains can be achieved mainly through pipelining effects. An example is found in [10]. *Connection-level parallelism* associates connections with a single processor or thread, achieving speedup with multiple connections. Multiprocessor STREAMS most closely matches this model [26, 27]. *Functional parallelism* decomposes functions within a single protocol and assigns them to processing elements. Examples include [19, 23, 25]. The relative merits of one approach over the others depends on many factors, including the host architecture, the number of connections, whether the implementation is in hardware or software, the thread scheduling policies employed , and the cost of primitives such as locking and context switching. Schmidt and Suda [28] show that packet-level parallelism and connection-level parallelism generally perform better than layer parallelism on a shared-memory multiprocessor, due to the context-switching overhead when crossing layers using layer parallelism.

This paper provides an experimental performance study of packet-level parallelism using TCP/IP and UDP/IP protocol stacks. We have conducted this study in the context of a multiprocessor implementation of the *x*-kernel, which

[*]This research supported in part by NSF under grant NCR-9206908 and ARPA under contract number F19628-92-C-0089. Erich Nahum was supported by an ARPA Research Assistantship in Parallel Processing. David Yates is the recipient of a Motorola Codex University Partnership in Research Grant. The authors can be reached at {nahum, yates, kurose, towsley}@cs.umass.edu.

runs in user space on Silicon Graphics shared-memory multiprocessors using the IRIX operating system.

Our results show that only limited packet-level parallelism can be achieved under TCP within a single connection, but that using multiple connections improves available parallelism. The effects of checksumming and packet size on speedup are also examined. We also find that ordering plays a key role in determining single-connection TCP performance, that careful use of locks is a necessity, and that selective exploitation of caching can improve throughput. Finally, we examine packet-level parallelism on three platforms: the current Challenge series using both 100 MHz and 150 MHz MIPS R4400 processors, and the older Power Series with 33 MHz MIPS R3000's.

The remainder of the paper is organized as follows: In Section 2, we describe our experimental environment, including the parallelized *x*-kernel and protocols. Section 3 gives our baseline results. Section 4 describes ordering issues and shows how ordering can impact performance. Section 5 examines locking strategies and techniques. Section 6 illustrates the effects of caching. Section 7 examines parallelized protocol performance on several different architectures. In Section 8 we summarize our results and conclude.

2 Experimental Environment

As stated earlier, our environment is based on a parallelized *x*-kernel, and as such, it is similar in several respects to the platform described by Bjorkman and Gunningberg at the Swedish Institute of Computer Science (SICS) [4, 5]. Our platform was, for the most part, developed independently, and for a different type of machine. The exception is the SICS MP TCP code, which we used to guide the design of our parallel TCP, as described in Section 5.1. The SICS platform, however, was based on the February 1992 release of the *x*-kernel, and ran on the Sequent Symmetry. Our environment is based on the December 1993 *x*-kernel release, and runs on the SGI Challenge. Given the differences in hardware, host operating systems, versions of the *x*-kernel infrastructure and protocols, a direct comparison is thus not possible. Where applicable, however, we describe differences between the systems.

2.1 Parallelized *x*-kernel

Our parallelized *x*-kernel was developed by adding locks into appropriate places in the *x*-kernel infrastructure. Like the SICS system, we placed locks protecting *x*-kernel infrastructure within the *x*-kernel, and placed locks concerning protocols within the protocols. Unlike the SICS system, which used a finite set of static, global locks, we instantiate locks on a finer-grained, per data-structure basis.

The *x*-kernel's *message tool* is a facility for managing packet data, analogous to Berkeley mbuf's. Messages are per-thread data structures, and thus required no locks. They point to allocated data structures called *MNodes* which are reference counted; these reference counts must be incremented and decremented atomically.

The *x*-kernel's *map manager* provides a mapping from an external identifier (e.g., a TCP port number) to an internal identifier (e.g., a TCP protocol control block), using chained-bucket hash tables with a 1-behind cache. Maps have many uses, but are primarily used for demultiplexing. They must be locked to insert, lookup, or remove entries. In addition, since the map manager provides an *iterator* function `mapForEach()`, the map manager can call itself recursively. To handle this recursion, counting locks are used, so that if a thread already owns the lock, it simply increments a count and proceeds. Similarly, an unlock decrements the count, and the lock is released when the count reaches zero.

The *event manager* uses a timing wheel [31] to manage events which are to occur in the future. The wheel is essentially another chained-bucket hash table, where the hashing function is based on the time that the event is scheduled to run. To protect this structure, we added per-chain locks, so that concurrent updates to the table were less likely to conflict with one another.

Other components of the *x*-kernel require locks for various reasons, most frequently for atomic addition and subtraction for object reference counts.

2.2 Parallelized Protocols

In order to experimentally study various performance-related issues in parallel protocols, we implemented multiprocessor versions of FDDI, IP, UDP, and TCP. This section briefly describes our parallel implementations of these protocols.

The FDDI protocol in the *x*-kernel is very simple; it essentially prepends headers to outgoing packets and removes headers from incoming packets. Locking is only necessary in two instances: during session creation and on packet demultiplexing (to determine the upper-layer protocol to which a message should be dispatched to). No locking is required for outgoing packets during data transfer.

The Internet Protocol is structured similarly to FDDI but has a slightly larger amount of state, which must be locked. On the send side, IP has a datagram identifier used for fragmenting packets larger than the network interface MTU. The identifier must be atomically incremented, per-datagram. On the receive side, if a packet is a fragment, a fragment table must be locked to serialize lookups and updates.

UDP is a connectionless transport protocol that provides little beyond simple multiplexing and demultiplexing. Like FDDI, locking is only required for session creation and packet demultiplexing.

TCP is a much more complex protocol than UDP. It provides reliable, in-order data delivery with no loss, error, or duplication, and has built in flow control and congestion control mechanisms. Our TCP is based upon the *x*-kernel's adaptation of the Berkeley Tahoe release, but was updated to be compliant with the BSD Net/2 release. In addition to adding header prediction, this involved updating the congestion control and timer mechanisms, as well as reordering code in the send side to test for the most frequent scenarios first [15]. The one change we made to the base Net/2 structure was to use 32-bit flow-control windows, rather than the 16-bit windows defined by the TCP specification. This turns out to be important for the high bandwidths generated by our experiments, and we note that 32-bit flow control information is used in both 4.4 BSD with large windows [16] and in the next-generation TCP proposals [6, 30].

Due to the semantics of TCP, the protocol consequently has a great deal of per-connection state, which must be locked to provide consistency and semantic correctness. For example, each connection has a retransmission queue, a reassembly queue, and various windows for both the send and receive sides. Given this large amount of state, several state locking strategies are possible. We thus implemented three Net/2-based versions of TCP, where each version uses a different locking granularity. These are described in more detail in section 5.1.

Checksumming has been identified as a potential performance issue in TCP/UDP implementations. We thus wished to examine the extent to which checksumming made a difference in protocol speedup and throughput. The checksum code used in our studies was the fastest available portable algorithm that we were aware of, which was from UCSD [18].

2.3 In-Memory Drivers

Since our platform runs in user space, accessing the FDDI adaptor involves crossing the IRIX socket layer, which is prohibitively expensive. Normally, in a user-space implementation of the *x*-kernel, a simulated device driver is configured below the media access control layer (in this case, FDDI). The simulated driver uses the socket interface to emulate a network device. To avoid this socket-crossing cost, we replaced the simulated driver with in-memory device drivers for both the TCP and UDP protocol stacks. The drivers emulate a high-speed FDDI interface, and support the FDDI maximum transmission unit (MTU) of slightly over 4K bytes. This is similar to the approaches taken in [5, 11, 21, 28].

The drivers act as senders or receivers, producing or consuming packets as fast as possible, to simulate the behavior of a simplex data transfer over an error-free network. To minimize execution time and experimental perturbation, the receive-side driver uses preconstructed packet templates, and does not calculate TCP and UDP checksums. Instead, in experiments that examine checksumming using a simulated sender, the actual TCP and UDP receivers calculate the checksum, but ignore the result.

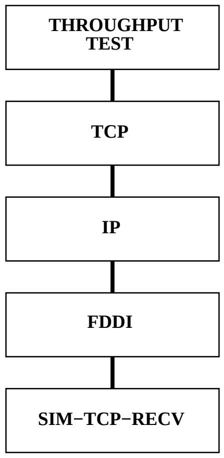

Figure 1: TCP Send-Side Configuration

Figure 1 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 acknowledgement packets for packets sent by the actual TCP sender. The driver acknowledges every other packet, thus mimicking the behavior of Net/2 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 TCP receiver, and flow-controls itself appropriately using the acknowledgements and window information returned by the TCP receiver. Both simulated TCP drivers also perform their respective roles in setting up a connection.

3 Baseline Results

In this section we present a set of baseline results on our 8-processor 100 MHz Challenge machine. Our goal here is to illustrate the differences between the send and receive paths, and the impact of checksumming and packet size on scalability. The baseline protocol implementations which generated these results include message caching, atomic increment/decrement, and (in the case of TCP) a single lock on the TCP state. The locks used are the SGI supplied mutex locks. In sections 5 and 6 we describe these protocol structuring and implementation choices, and examine how they and various other alternative approaches effect and determine performance.

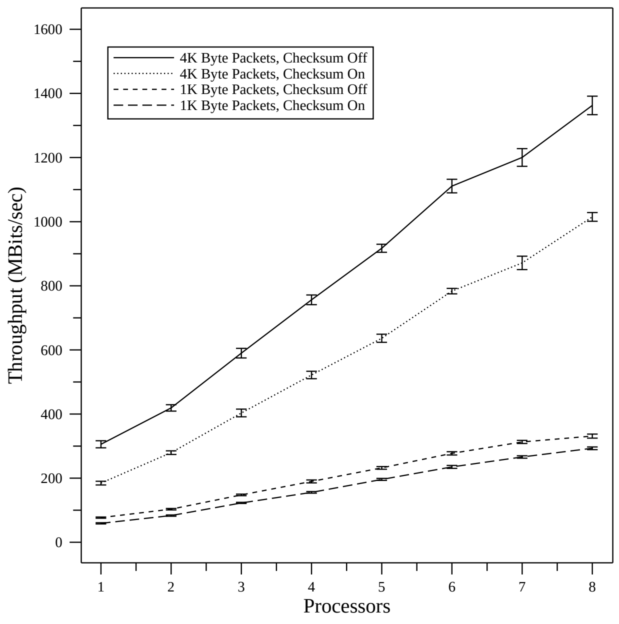

Figure 2: UDP Send Side Throughputs

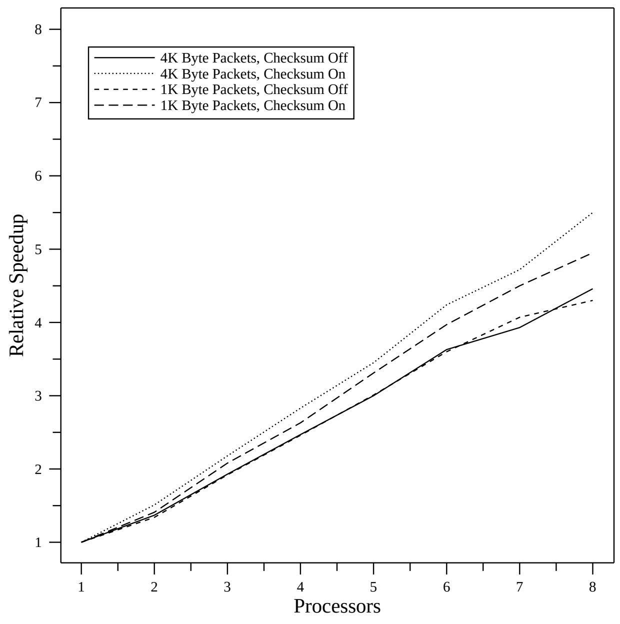

Figure 3: UDP Send Side Speedup

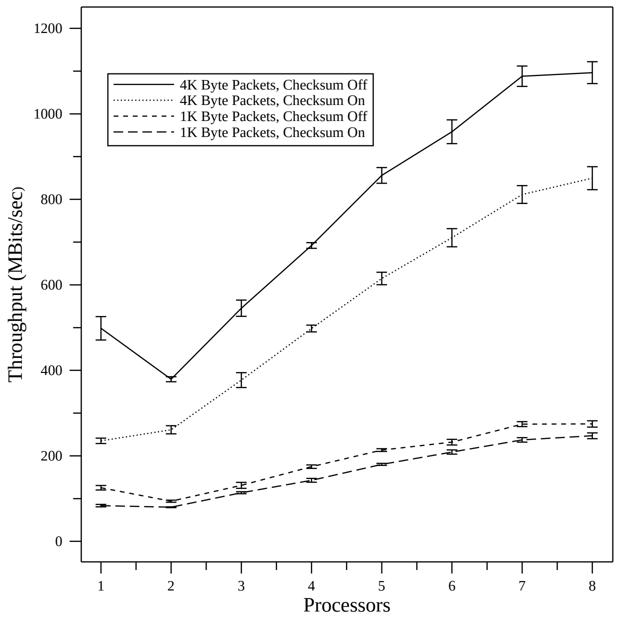

Figure 4: UDP Receive Side Throughputs

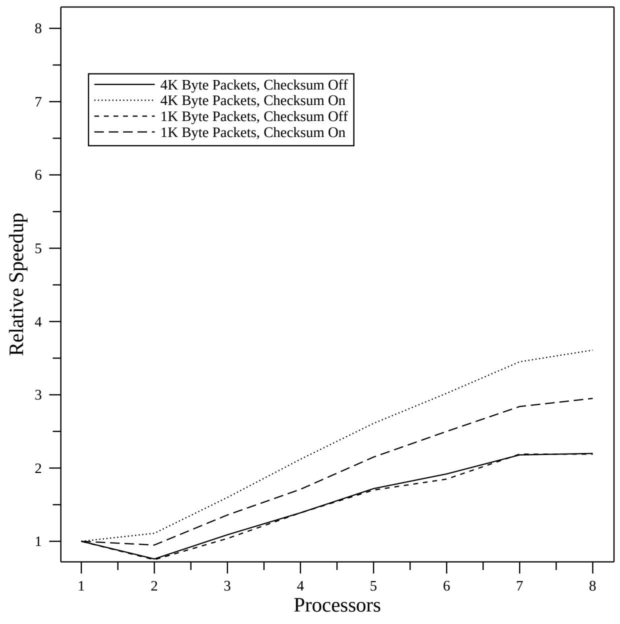

Figure 5: UDP Receive Side Speedup

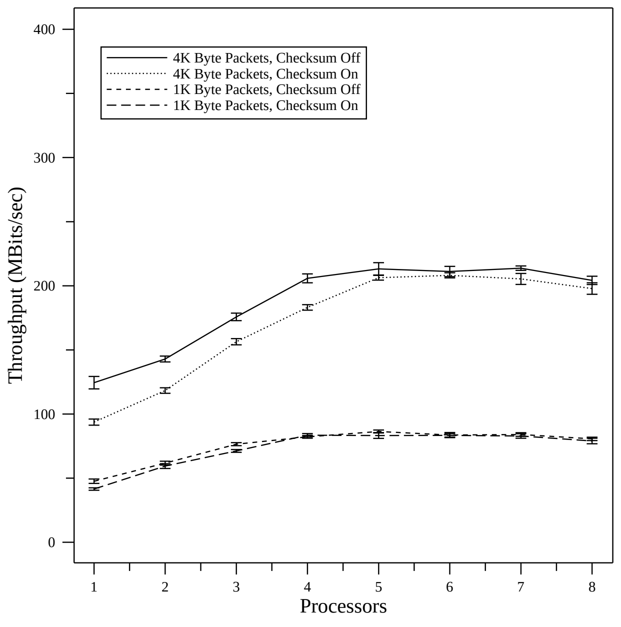

Figure 6: TCP Send Side Throughputs

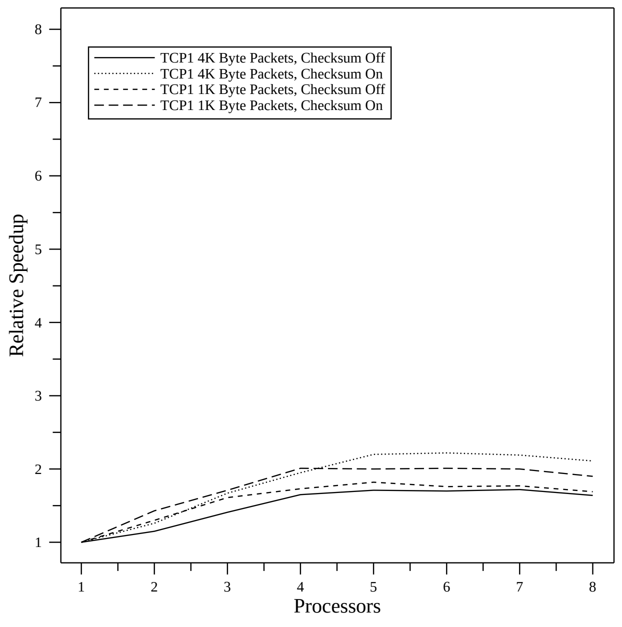

Figure 7: TCP Send Side Speedup

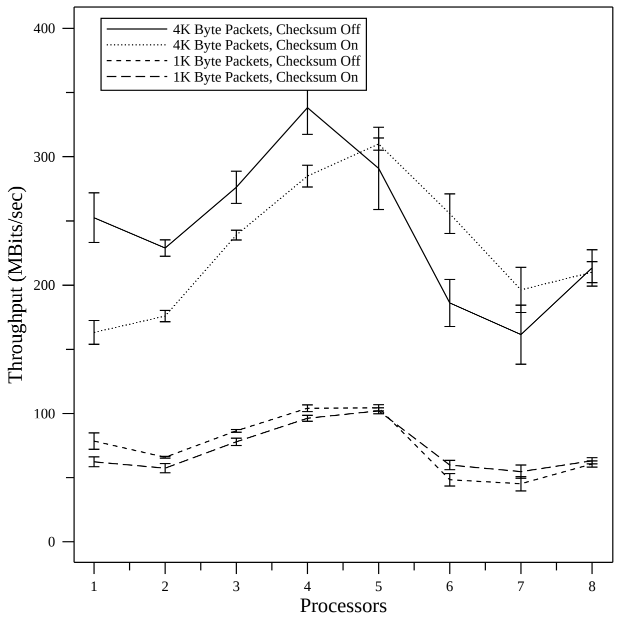

Figure 8: TCP Receive Side Throughputs

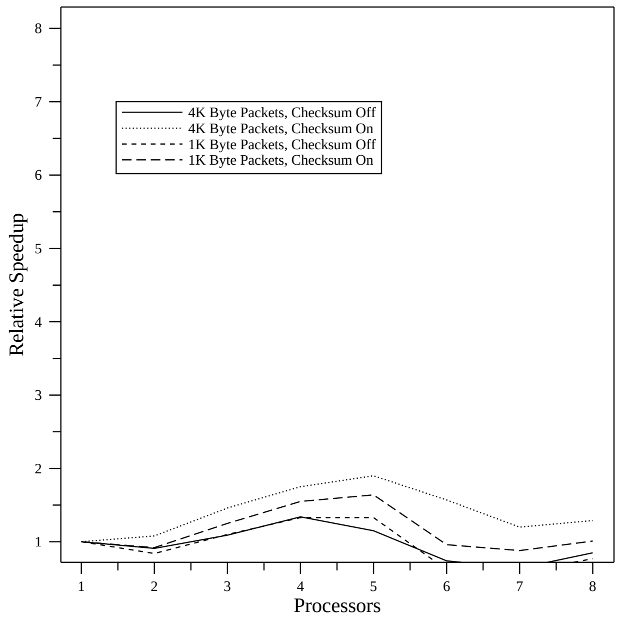

Figure 9: TCP Receive Side Speedup

In our experiments each processor has a single thread which is wired to that processor, similar to the method used by Bjorkman and Gunningberg. To see if wiring impacted our results, we ran several experiments without wiring threads to processors with TCP and UDP, send and receive side, with and without checksumming. The only change we observed was a small (approximately ten percent) difference on the send side for UDP above 4 processors. IRIX 5.2 schedules for cache affinity, and so we conclude that wiring has little perturbation of our experiments.

3.1 Send and Receive Side Processing

Figure 2 shows UDP send-side throughput, for a single UDP connection, in Megabits per second, measured on our 8-processor Challenge machine. Figure 3 shows relative speedup for the send side in UDP, where speedup is normalized relative to the uniprocessor throughput for that particular packet size. Figures 4 and 5 show UDP receive-side throughput and speedup, respectively. For these and all subsequent graphs, each data point is the average of 10 runs, where a run consists of measuring the steady-state throughput for 30 seconds, after an initial 30 second warmup period. In addition, we isolated our Challenge multiprocessor as much as possible by running experiments with no other user activity. All non-essential daemons were removed, and the machine did not mount or export any remote file systems. To check variance, we ran one 8-processor test 400 times, and observed that the data fit a normal bell-curve distribution. Throughput graphs include 90 percent confidence intervals.

The figures show that, as Bjorkman and Gunningberg discovered [5], UDP send-side performance scales well with larger numbers of processors. In our discussion, *scalability* means the first derivative of speedup as the last processor is added to the experiment. Note that a test can demonstrate high speedup to a point but exhibit poor scalability. We observe that send and receive side processing scale differently, but we do not wish to claim any inherent difference between their relative scalability. This is because our send-side experiments explicitly yield the processor on every packet, but the the receive side relies on the operating system to preempt the thread. This is partially a historical artifact of our implementation, and we plan a more detailed comparison between the two in the future.

One major difference between the send and receive paths is that a protocol's receive processing must demultiplex incoming packets to the appropriate upper layer protocol. At first, we thought that the locks used in the map manager for demultiplexing might be creating a bottleneck, but running the test without locking the maps yielded a small (approximately 10 percent) improvement in throughput.

The TCP throughput and speedup results, again for a single connection, are given in Figures 6-9 respectively. The TCP numbers here are from our baseline TCP, TCP-1, further described in section 5.1. Our results show that TCP does not scale nearly as well as UDP, in either the send or receive case. Locking state is the culprit here. For example, profiling with Pixie [29] shows that in an 8-processor receive-side test, 90 percent of the time is spent waiting to acquire the TCP connection state lock; on the send side, the amount is 85 percent.

Several unusual points warrant mentioning. Figure 6 shows that send-side throughput appears to level off at around 215 megabits/sec. Figure 8 shows that receive-side throughput levels off above 350 megabits/sec, but then drops off suddenly afterwards. This dip is caused by the combination of TCP packets being misordered when threads contend for the connection state lock, and the difference in processing times for in-order versus out-of-order packets in TCP. Section 4.1 discusses how this problem was discovered, as well as the solution.

3.2 Checksumming and Packet Size

As mentioned earlier, we were interested in how checksumming and packet size influence the performance of parallel protocols. Our expectations were that relative speedup would be greater when processing larger packets with checksumming, since checksumming occurs outside of locked regions and thus constant per-packet costs would constitute a smaller fraction of the processing time [17]. Figures 3, 5, 7, and 9 show that, in general, tests with larger packets have better speedup than those with smaller ones, and experiments for a particular packet size with checksumming have better speedup than those without, although the differences are not as pronounced as we had expected. The trends agree somewhat with those shown in [11], which showed better speedup with larger data units. However, their tests included presentation-layer conversion, which is much more compute-bound and data-intensive than checksumming.

Although the SGI documentation gives the aggregate bus bandwidth as 1.2 gigabytes/sec, we wished to see the read bandwidth limitations imposed by checksumming. To this end, we ran a micro-benchmark that checksummed over a large amount of data, to force cache misses. We observed that each processor could checksum at a rate of 32 MB/sec, or 256 megabits/sec, at least up to 8 processors. Assuming the bandwidth does not degrade as processors are added, this implies that the bus could support up to 38 processors doing nothing but checksumming.

4 Ordering Issues

4.1 Ordering Issues in TCP

Recall that in Figure 8, receive-side TCP throughput falls drastically beyond 4 or 5 processors. Further investigation showed large numbers of out-of-order arrivals at the TCP layer, a surprising result since data was being generated

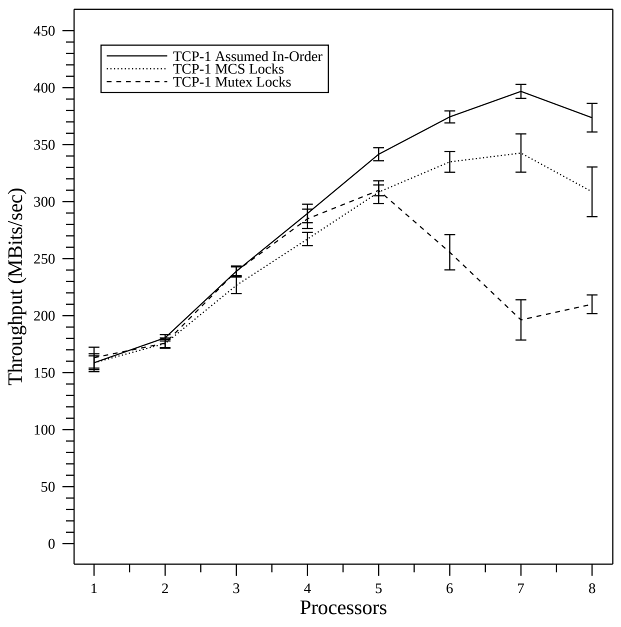

Figure 10: Ordering Effects in TCP

in-order by the simulated TCP sender. As the TCP header prediction algorithm is dependent on the arrival of in-order packets, we hypothesized that out-of-order arrivals were reducing performance. To test this hypothesis, we ran a test using a version of TCP modified to treat *every* packet as if it were in-order. The result was the disappearance of the anomaly. The question then became how to bridge the gap between the observed behavior and the forced in-order experiment.

The Pixie results showed high contention for the connection state lock, and since the raw mutex locks provided by IRIX are not FIFO, this suggested that lock contention was causing threads, and thus packets, to be reordered. To preserve the original ordering, we implemented FIFO queueing using the MCS locks by Mellor-Crummey and Scott [22]. Their locking algorithm requires atomic swap and compare-and-swap functions, which we implemented using short R4000 assembler routines.

Figure 10 illustrates the effects, using 4 KB packets with checksumming on. The top curve in the figure is from the modified TCP where packets are *assumed* to be in order, a potential upper bound. The bottom curve is the baseline TCP-1 implementation using regular mutex locks for the connection state. The middle curve is TCP-1 using MCS locks. We see that using these locks bridges the majority of the gap between the baseline case and the "upper bound." In the case with checksumming off (not shown), there is no statistically discernible difference between the performance of the "upper bound" TCP and TCP with MCS locks. Closing the remainder of the gap with checksumming is not trivial. For example, we tried a receive-side test where map lookups were serialized by MCS locks, and observed a slight reduction in throughput. Since MCS locks have a greater fixed-overhead cost without contention than the straight mutex locks (1.5 usec vs. 0.7 usec), we did not wish to simply replace all mutex locks in the system with MCS locks. However, as observed above, in the right scenario they can create an enormous performance win.

Processors	1	2	3	4	5	6	7	8
Mutex Locks	00	02	04	05	11	25	42	54
MCS Locks	00	02	04	06	09	11	14	18

Table 1. Percentage of packets out-of-order.

Table 1 also shows the impact of using FIFO locks. The table gives the percentage of packets received out-of-order in TCP with mutex locks and MCS locks, for a receive-side test using 4KB packets with checksumming. The table shows a large difference in the number of out-of-order packets between the two locking schemes as the number of processors increases.

An interesting side issue is the misordering that can occur on the send side when threads pass each other below TCP but before reaching the FDDI driver. This would cause packets to be placed out-of-order on the wire, and probably arrive out-of-order at the receiver. To quantify this potential problem, we measured the percentage of out-of-order packets in the send-side driver, and observed that fewer than one percent were misordered with up to eight processors.

4.2 Ordering and Correctness

We note that preserving order is a *semantic correctness* issue. If an application uses TCP and cannot cope with out-of-order delivery, packet order must also be preserved *above* TCP. When parallelism is introduced, an arriving packet cannot simply release the TCP connection state lock and continue; the moment the lock is relinquished, guarantees of ordered data above TCP are lost. Similarly, on the send side, the order of the data the application passes to TCP must be preserved, lest the order TCP preserves is different from the one the application observed. For some applications, this is not a problem. For example, NFS does not assume ordered packets, and can be configured to use TCP. In most cases, however, the application requires order to be preserved.

To examine this issue, we implemented a ticketing scheme similar to a bakery algorithm. Before releasing the TCP connection state lock, a receiving thread acquires an *up-ticket* for the next higher layer. The thread then releases the connection state lock, and continues up the stack. In the test application above TCP, at the point where the application requires order, the thread can then wait for its ticket to be called. The amount of mechanism required to implement this feature is not large, but restricts order, further limiting performance. Figure 11 shows a receive side TCP throughput test using 4KB packets, comparing an application that requires order preservation versus one that

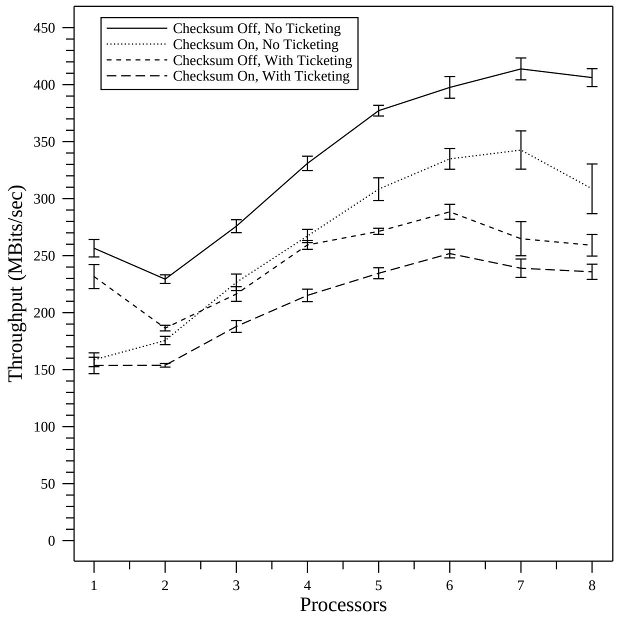

Figure 11: Ticketing Effects in TCP

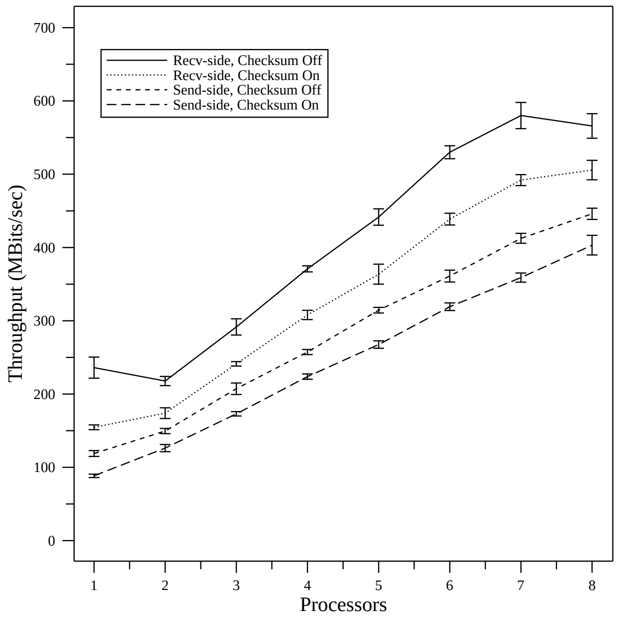

Figure 12: TCP with Multiple Connections

does not. In this example, the application is our test code, which simply counts packets that arrive. The application's critical section itself is small, a lock-increment-unlock sequence; the performance is lost preserving the order.

We are not the first to observe this problem [11, 13], but to our knowledge, previous work has not provided adequate solutions. For example, in [11], Goldberg et. al. use a ticketing scheme similar to ours, but assign tickets to packets at the driver for use in re-ordering at the application. However, this assumes a one-to-one correspondence between arriving packets and application data units. It does not address issues such as corrupted packets that are dropped, fragmented packets that are reassembled, or packets that are not data at all, such as acknowledgements.

The more general problem is to provide a mechanism that is correct in a general fashion, across several protocol layers. The solution we describe above only solves the problem when there is a one-to-one correspondence between a TCP connection and the application's notion of a connection. This is the case in the example of TCP and BSD sockets. However, if a TCP connection was multiplexed by several other higher-layer protocols, each message must be "re-ticketed" at each multiplexing or demultiplexing point[1]. A general solution that meshes with the *x*-kernel's infrastructure is an issue still under study.

4.3 Multiple Connections

Given the performance penalty exacted for maintaining order, and the single-connection performance limits in TCP, we argue that if parallel applications are to reap the benefits of parallelized networking, they should perform their own ordering. Using either a connectionless protocol such as UDP or a connection-oriented protocol such as TCP with *multiple connections*, an application *must* be able to handle out-of order delivery. Lindgren et. al. [21] make a related argument that the parallel application must be tied closely to the parallel communication system.

To illustrate the benefits of using multiple connections, we ran send-side and receive-side experiments of TCP-1 with MCS locks, without ticketing, using 4KB packets with and without checksumming. In these tests, each processor was responsible for a separate connection. For example, the eight processor experiment examines throughput for eight connections. The simulated drivers were modified slightly to support multiple connections for these tests. The results are shown in Figure 12. The graph shows steadily increasing throughput as connections (and their associated processors) are added. This test is somewhat "idealized" in that the distribution of traffic across connections is relatively uniform. However, the point of the experiment is to show that the connection state lock is the major bottleneck for a single connection, and that it may be overcome by using multiple connections.

5 Locking Issues

5.1 Locking Granularity in TCP

Recall that TCP maintains a relatively large amount of state per connection. A question we wished to address was how

[1]Thanks to Mats Bjorkman for pointing this out.

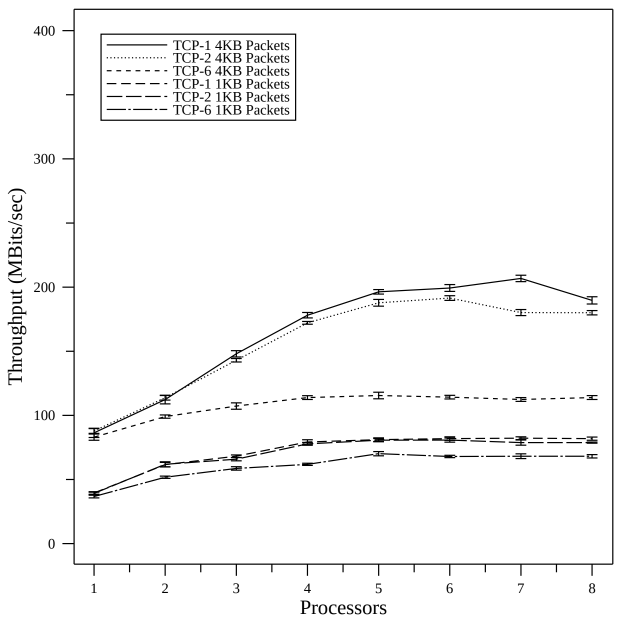

Figure 13: TCP Send-Side Locking Comparison

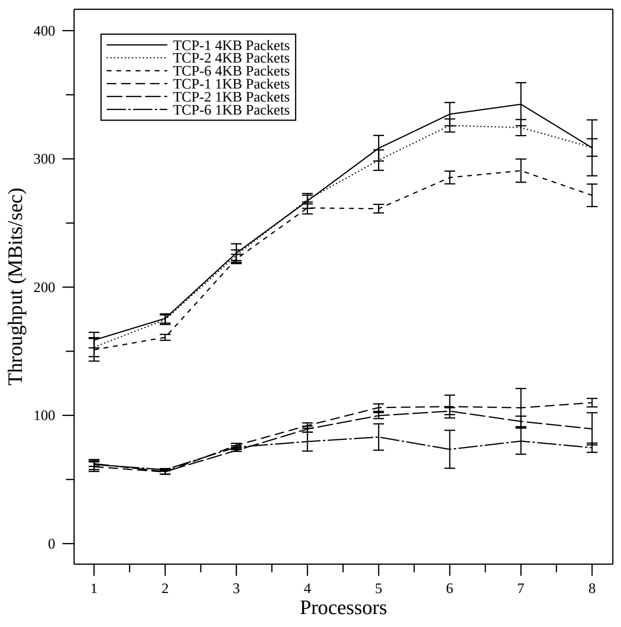

Figure 14: TCP Receive-Side Locking Comparison

that state should be locked in order to maximize performance and speedup. Towards this end, we produced three versions of our TCP, each with a different number of locks. For illustrative purposes, we call them TCP-N, where N indicates the number of locks involving connection state. The first version, the baseline given in Section 3, is TCP-1, which uses only a single lock to protect all connection information. The second version, TCP-2, uses two locks per connection: one to protect send-side state, and the other to protect receive-side state. The last version, TCP-6, uses the locking style from the SICS MP TCP, with six locks serializing access to various components of the connection state.

More specifically, TCP-6 has separate locks to protect the receive-side reassembly queue, the send-side retransmission buffer, the header prepend operation, header remove operation, send side window state, and receive side window state. In most cases, this locking is either redundant or unnecessary. For example, header manipulation occurs solely on the stack of the calling thread; thus, no locking is necessary. Similarly, the send and receive queues need to be locked at the same time as the send and receive window state, which is redundant.

Another concern we had with the SICS TCP implementation was that locks were being held where checksum calculation would have been done, on both incoming and outgoing packets.[2] In the *x*-kernel, this occurs where headers are prepended or removed, respectively, and the TCP-6 code is consistent with their implementation. However, we saw that locking was not necessary here, and our two other TCP implementations reflect this. The key realization is that checksumming a packet is orthogonal to manipulating connection state. The only change needed was, in the case of the outbound processing in `tcp_output`, the checksum calculation had to be moved so that it was done outside the scope of the send window lock. This did not affect correctness, however.

The results for the three TCP implementations are given in Figures 13 and 14, which plot send and receive side throughput respectively with checksumming. The three TCP's measured here are based on the baseline version described in Section 3 with the addition of MCS locks. The goal here is simply to compare locking strategies. TCP-1 and TCP-2 both outperform TCP-6, particularly when checksumming is enabled. With checksumming off, the gaps are smaller, but the relative ordering between the three TCP's is the same. In all cases, send and receive side, with and without checksumming, the code with the simplest locking, TCP-1, performed the best. We also observed this behavior when the three TCP's did *not* include MCS locks. In retrospect, we can see how the single-lock version would perform the best, since the Net/2 TCP implementation manipulates send-side state on the receive path, and receive-side state on the send path. For example, in the TCP header prediction code (intended to be common-case processing), both the send and receive state locks must be acquired.

Another attractive feature of using a single lock is its simplicity. Implementation is easier, deadlock is easier to avoid, and atomicity of changes to protocol state is easier

[2] We note that Bjorkman and Gunningberg reported results for TCP without checksumming.

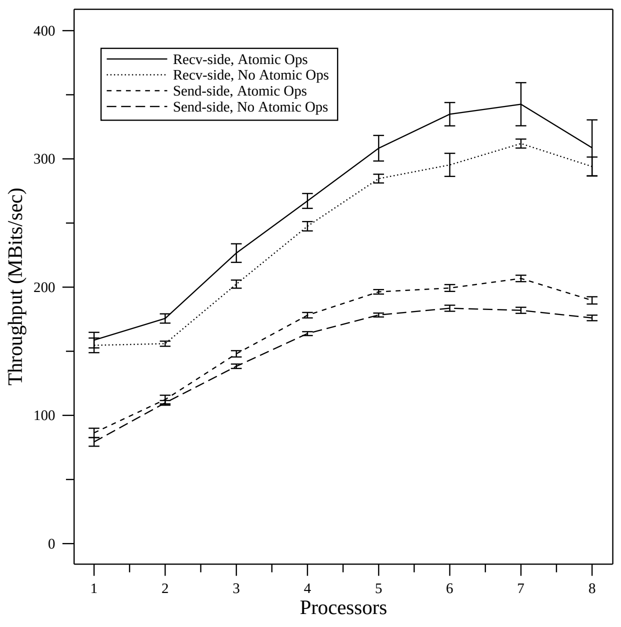

Figure 15: TCP Atomic Operations Impact

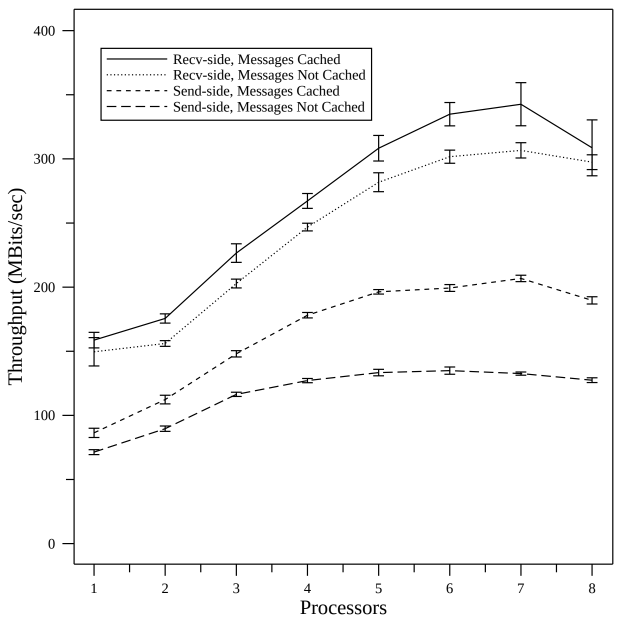

Figure 16: TCP Message Caching Impact

to guarantee.

We note that this result is specific to the BSD implementation, and that a TCP implementation designed around separating send and receive side processing may well yield better speedup with multiple locks. However, due to the widespread use of the BSD code, using the Net/2 example is applicable to many operating systems.

5.2 Atomic Increment and Decrement

Another locking issue we examined was using atomic increment and decrement functions that exploited the R4000's load-linked (LL) and store-conditional (SC) instructions. LL and SC allow programmers to produce *lock free* primitives [12]. A simple example of this is atomic increment, which replaces a lock-increment-unlock sequence.

We tried this for two reasons. First, the *x*-kernel's message tool relies on the notion that reference counts are atomically manipulated, and so the primitives map perfectly with the existing code. Thus, the primitives benefit the message tool, and subsequently all protocols that use it. Second, the *x*-kernel uses reference counts on session and protocol state in order to know when objects can be freed. When a packet is demultiplexed, these reference counts are incremented on the way up the stack and then decremented on the way down. This means that two locks are acquired and released *per-layer*, on the fast path of data transfer. Thus, atomic primitives again potentially benefit the entire protocol stack.

Replacing a lock-increment-unlock sequence with atomic increment pays off in two ways. First, a layer of procedure call is removed, which can affect performance on the fast path. Second, in the best case, it reduces memory traffic by replacing three writes with a single one. We implemented these primitives with short R4000 assembler routines. Sample results are given in Figure 15, which shows the effects of atomic primitives on TCP throughputs with 4KB packets and checksumming on. Both TCP and UDP see improvements with the atomic primitives. The UDP receive-side obtains a larger benefit than the send side from atomic increments, due to the reference count manipulation that happens during demultiplexing. The benefits to the TCP send and receive sides were approximately equal, as the majority of the improvement is due to a more efficient message tool.

6 Per-Processor Resource Caching

As mentioned earlier, *x*-kernel protocols make heavy use of the message tool to manipulate packets. Since caching has been shown to be effective in data structure manipulations [2, 7], we decided to evaluate the use of simple per-thread resource caches in the message tool. Whenever a thread requires a new MNode (the message tool's internal data representation), it first checks a local cache, which can be done without locking. The cache is managed last-in first-out (LIFO) to maximize cache affinity. This avoids contention in two ways: first, the lock in `malloc` serializing memory allocation is avoided, reducing locking contention, and possible system calls (e.g., `sbrk`). Second, memory freed by a processor is re-used by that processor, avoiding memory contention. Figure 16 gives a sample of the results, displaying TCP throughputs with 4 KB packets with check-

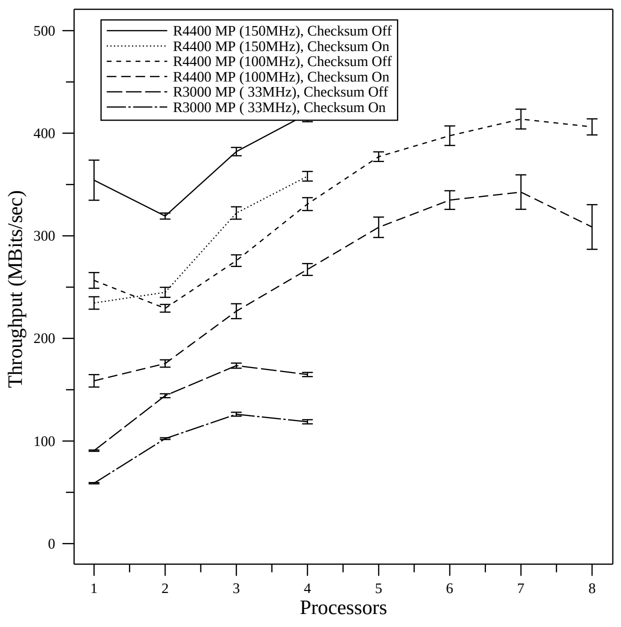

Figure 17: TCP Throughputs across Architectures

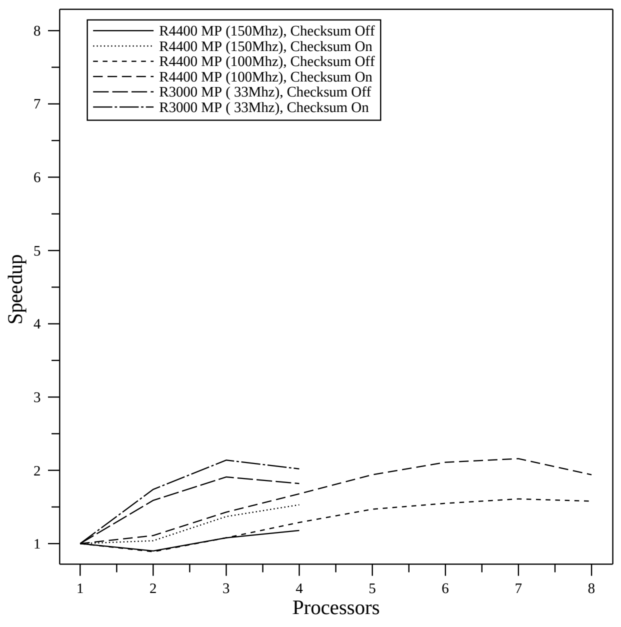

Figure 18: TCP Speedups across Architectures

summing. The improvement in TCP is significant, due to its heavy use of the message tool. The results are also positive for UDP send and receive side.

7 Architectural Trends

Of the experiments given in Section 3 that coincide with those given by Bjorkman and Gunningberg on the Sequent, we observed similar trends but relatively lower speedups. Drawing conclusions based on comparing speedups between two largely different architectures would most likely be inappropriate. Still, we were curious as to the differences that hardware made, since the Sequent used in their experiments was an older machine. Although we could not compare our results with theirs directly, we thought it would be illustrative to run our code on older hardware. To this end, we ran the same experiments on a Power Series, the previous generation Silicon Graphics multiprocessor, with four 33 MHz MIPS R3000's. We also ran our experiments on a faster version of our machine, a four-processor Challenge using 150 MHz R4400's. In all cases, the machines ran version 5.2 of the IRIX operating system. In these additional tests, we did not have exclusive access to the other machines, and so were not able to isolate them as carefully as with our Challenge experiments. However, we did run all our tests with minimal other activity on the systems, and the width of the confidence intervals on these graphs show that variance is low.

Examples of the architectural comparisons are given in Figures 17 and 18, which show receive-side throughput and speedup respectively for TCP on the three platforms. Space constraints prevent us from showing all of our data, but in general, our findings were consistent across platforms. We do not wish to draw broad conclusions, especially from machines with only four processors, but we can summarize our observations:

- On all platforms, TCP-1 outperformed TCP-2 and TCP-6.
- On all platforms, UDP send-side scaled well, and TCP scaled poorly.
- On all platforms, the fastest machine had the highest throughput for a particular test.
- Speedup was consistently best on the Power Series (the oldest machine) and about the same on the two Challenge platforms.
- The two Challenge machines exhibited the receive-side drop in throughput at 2 processors, but the Power Series did not. In particular, UDP receive-side performance scales on the Power Series as far as could be observed, namely up to four processors.

The last item is perhaps the most interesting. Without more detailed information, we cannot assert any explanations for the behavior. We do note though, that the Power Series performs locking using a separate dedicated synchronization bus, similar to the Sequent. The Challenge, however, uses memory to synchronize, relying on the coherency protocol and the load-linked/store-conditional instructions [9]. Given that Bjorkman and Gunningberg did not observe the receive-side drop for their UDP receive side tests on the Sequent, we suspect that the difference in synchronization may be the cause of the anomaly. We are pursuing further studies along these dimensions.

Finally, the 100 MHz Challenge uniprocessor throughputs are roughly 25 to 50 percent better than those of the 33MHz Power Series. This is surprising, given that the former has a three times faster clock cycle, on-chip caches, and larger secondary caches. This is only one architectural comparison, with different generations of both the MIPS architecture and multiprocessor interconnects. Still, it suggests that network protocol processing speed may not be improving as fast as application performance, which agrees with the operating system trends shown in [3, 24]. We plan to investigate this further.

8 Conclusions and Future Work

We briefly summarize our findings as follows:

- *Preserving order pays.* We showed that, in cases where contention for locks perturbs order, simple FIFO queueing locks preserve this order, which improves performance.

- *Single-connection TCP parallelism is limited,* both on the receive side, and on the send side, even more so than shown by Bjorkman and Gunningberg.

- *Multiple-connection TCP parallelism can scale,* since contention for the connection state lock is avoided. However, the application must manage order across connections.

- *Exploiting cache affinity, and avoiding contention, is crucial.* This is demonstrated by the effectiveness of per-processor resource caching. Contemporary machines are memory-bound, due to the disparity between CPU and memory speeds, and the gap is only expected to grow.

- *Simpler locking is better.* We showed that, on a modern machine, locking structure impacts performance, and that a complex protocol with large connection state yields better speedup with a single lock than with multiple locks.

- *Atomic primitives can make a big difference.* Replacing sequences of lock-increment-unlock with an atomic increment improved receive-side TCP and UDP performance by about 20 percent on average, and send-side between 5 and 10 percent.

- *Checksumming has some influence on speedup.* This was demonstrated by the differences in relative speedup between experiments with and without checksumming.

These results indicate that packet-level parallelism is especially beneficial for connectionless protocols, but that connection-oriented protocols will have limited benefits in speedup within a single connection. Applications will need to use multiple connections to obtain parallel performance with connection-oriented protocols, which means they must manage order between connections. Due to time and space constraints, we have only been able to briefly address multiple connections in this paper. We plan to examine issues involving multiple connections more in-depth, and to examine another strategy of parallelizing protocols involving connection-level parallelism.

Acknowledgements

We are indebted to Larry Peterson and the *x*-kernel group at the University of Arizona for extending their help and hospitality, and allowing us to use their Power Series machine. Ed Menze and Hilarie Orman answered countless questions. Lawrence Brakmo assisted our understanding of TCP in general. Mats Bjorkman, Franklin Reynolds, and Franco Travostino discussed ordering issues with us. Special thanks to Mats Bjorkman for providing us with his original parallel TCP code, helping us with understanding it, and engaging in lively discussion. Thanks to David Oliver at the Center for Geometry Analysis Numerics and Graphics at the University of Massachusetts for letting us use their 150 MHz Challenge machine.

Many people contributed comments that improved the form and content of this paper, including Mats Bjorkman, Neal Nuckolls, and the anonymous reviewers.

References

[1] Brian Allison. DEC 7000/10000 Model 600 AXP multiprocessor server. In *Proceedings IEEE COMPCON*, pages 456–464, San Francisco CA, February 1993.

[2] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[4] Mats Björkman. The xx-Kernel: an execution environment for parallel execution of communication protocols. Dept. of Computer Science, Uppsala University, June 1993.

[5] Mats Björkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 74–83, San Francisco, CA, September 1993.

[6] Dave Borman. NTCP: A proposal for the next generation of TCP and UDP. In *Submission to the End2End-Interest mailing list*, pages 1–37, Eagan, MN, 1993. Cray Research. End2End archives available via FTP at ftp.isi.edu.

[7] Peter Druschel and Larry 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.

[8] Michel Cekleov et. al. SPARCCenter 2000:Multiprocessing for the 90's! In *Proceedings IEEE COMPCON*, pages 345–353, San Francisco CA, February 1993.

[9] Mile Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, Silicon Graphics Inc., Mt. View, CA, May 1994.

[10] Dario Giarrizzo, Matthias Kaiserswerth, Thomas Wicki, and Robin 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.

[11] Murray W. Goldberg, Gerald W. Neufeld, and Mabo 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 1993.

[12] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):6–16, November 1993.

[13] Norman C. Hutchinson. Protocols versus parallelism. In *Proceedings from the x-Kernel Workshop*, Tucson, AZ, November 1992. University of Arizona.

[14] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[15] Van Jacobson. Efficient protocol implementation. In *ACM SIGCOMM 1990 Tutorial Notes*, Philadelphia, PA, September 1990.

[16] Van Jacobson, Robert Braden, and Dave Borman. TCP extensions for high performance. In *Network Information Center RFC 1323*, pages 1–37, Menlo Park, CA, May 1992. SRI International.

[17] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 259–269, San Francisco, CA, September 1993. ACM.

[18] Jonathan Kay and Joseph Pasquale. Measurement, analysis, and improvement of UDP/IP throughput for the DECStation 5000. In *USENIX Winter 1993 Technical Conference*, pages 249–258, San Diego, CA, 1993.

[19] Odysseas G. Koufopavlou and Martina Zitterbart. Parallel TCP for high performance communication subsystems. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1395–1399, 1992.

[20] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, Reading, Massachusetts, 1989.

[21] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. An architecture and toolkit for parallel and configurable protocols. In *Proceedings of the International Conference on Network Protocols*, San Francisco, CA, October 1993.

[22] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[23] Arun 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, November 1990.

[24] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Conference*, pages 247–256, June 1990.

[25] Tom F. La Porta and Mischa 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, December 1992.

[26] David Presotto. Multiprocessor streams for Plan 9. In *UKUUG*, January 1993.

[27] Sunil Saxena, J. Kent Peacock, Fred Yang, Vijaya Verma, and Mohan Krishnan. Pitfalls in multithreading SVR4 STREAMS and other weightless processes. In *Winter 1993 USENIX Technical Conference*, pages 85–96, San Diego, CA, January 1993.

[28] Douglas C. Schmidt and Tatsuya Suda. Measuring the impact of alternative parallel process architectures on communication subsystem performance. *Fourth IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, August 1994.

[29] Michael D. Smith. Tracing with Pixie. Technical report, Center for Integrated Systems, Stanford University, Stanford, CA, April 1991.

[30] Robert Ullman. TP/IX: The next internet. In *Network Information Center RFC 1475*, Menlo Park, CA, June 1993. SRI International.

[31] George Varghese and Tony Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.