

# Networking Support for Large Scale Multiprocessor Servers (Extended Abstract)

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

## 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 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 applications where customers pay for access to information, 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 a large number 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* (CLP) 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.

In this paper, we experimentally evaluate both TCP/IP and UDP/IP protocol stacks, implemented using the *x*-kernel [8], which we extended to support connection-level parallelism. Our implementation runs in user space on a shared-memory Silicon Graphics (SGI) Challenge multiprocessor. 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.

Our results show that with careful implementation both TCP/IP and UDP/IP protocol stacks scale well with respect to number of processors. For example, when sending and receiving data using either protocol stack we observe linear speedup as processors (and connections) are added. Our results also show that both TCP and UDP stacks deliver high throughput which is, for the most part, sustained as the number of connections increases. We also find that for moderate to large numbers of connections, maximizing the number of threads in the system yields the best overall throughput. We also examine the fairness behavior of our implementation by considering both the range and value of per-connection throughputs. We find that the best fairness behavior is achieved by matching the number of threads to the number of processors while scheduling connections assigned to threads in a round-robin manner.

\*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.

Previous work on connection-level parallelism can be found in [5, 18, 20, 21]. The most complete performance study to date is [21], in which the authors show that by using a thread for each connection, throughput scales well for 20 connections, all receiving data, on a 20-processor Sun SPARCCenter 2000.

## 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.

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 [5, 18, 20, 21] has focused on relatively static assignments of connections to processes. One novel aspect of our implementation 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.

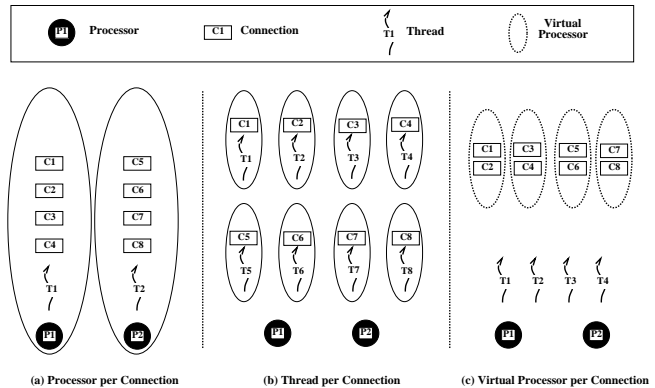


Figure 1: Approaches to Connection-Level Parallelism

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 1(a) shows an example, where 8 connections are mapped to 2 threads which are each bound to their own processor (ovals surrounding objects of different types in this figure denote a binding). 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), associates each connection with an individual process or thread.

Threads may be wired to processors, or may be allowed to migrate among processors. Figure 1(b) shows an example of TPC, with 8 connections, each with its own thread for sending data, which can run on either of the two processors. While TPC allows easy load balancing, it may not scale well with large numbers of connections, since each thread must be allocated resources.

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 1(c) shows an example where 8 connections are assigned to 4 virtual processors on 2 physical processors. In Figure 1(c), 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 which run on virtual processors are wired to physical processors), then VPPC is equivalent to PPC.

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 (by a packet filter or classifier [1, 14, 15, 24]) 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<sup>1</sup> and IP) over FDDI, running in a version of the *x*-kernel which was extended to support CLP.

#### 3.1 Connection-Level Parallel *x*-Kernel and Protocols

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 [8].

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 [2, 16, 18, 21], 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.

<sup>1</sup>Even though UDP is a connectionless protocol, we consider long term associations between a sender and receiver to be a “connection”, or flow. A UDP association is used to map connections to virtual processors, and provides a handle for demultiplexing, in the same fashion as TCP.

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, 24]) in contemporary operating systems since early demultiplexing yields other performance gains. For example, depositing received packets directly into application buffers avoids copying data [4, 13, 22]. Our use of a packet filter, to demultiplex to the appropriate virtual processor, leverages this existing mechanism for an additional purpose.

Since we do not have access to a device which matches the performance of our protocols, we replace the lowest level drivers in the *x*-kernel 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 4KB. This is similar to the approaches taken in [2, 7, 12, 16, 21].

These drivers simulate the behavior of a collection of clients sending or receiving data over simplex connections routed through an error-free network. Since they ensure that the protocol processing on the multiprocessor is that of either a data source or sink, our experiments require only one thread per virtual processor. 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.

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 TCP when communicating with itself as a peer. Since spawning threads is expensive in user space in SGI’s IRIX operating system, 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.

### 4 Results

To measure the performance of our connection-level parallel 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.

We evaluate our CLP 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 a large number of connections.

To quantify fairness, we measure the throughput seen by each connection, and compute quartiles between the maximum and minimum throughputs. In discussing these results, we refer to per-connection throughput or throughput per connection, whereas 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

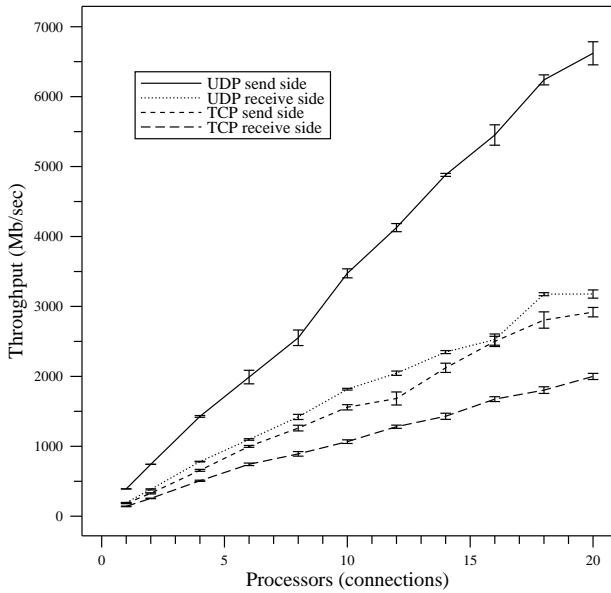


Figure 2: UDP and TCP Aggregate Throughput for Processor per Connection.

from 1 to 20, and bound a single connection to each processor using our processor per connection implementation.

Figure 2 shows throughput versus number of processors (and connections) for both TCP and UDP protocol stacks, where the checksum is computed on packets. The data points shown in this figure (and in all other figures) include 90% confidence intervals. VPPC and TPC data are omitted from Figure 2 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. As shown in [23], speedup computed from these data is linear for all four experiments, and is in the range of 14 to 17 at 20 processors.

Figure 2 gives 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 [21], which focus on the receive side for thread per connection parallelism.

The data presented in Figure 2 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 was 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.

To avoid constraining the IRIX scheduler, threads are not wired to processors in these experiments. Thus, the results presented here are for virtual processor per connection (VPPC) or thread per

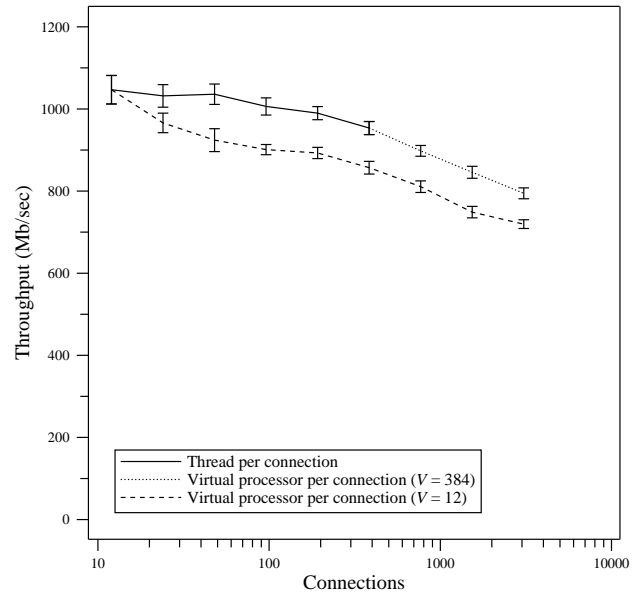


Figure 3: TCP Send-Side Throughput from 12 to 3072 Connections.

connection (TPC), even when the number of virtual processors is the same as the number of physical processors.

Figure 3 shows the overall throughput of our multiprocessor sending data on 12 to 3072 TCP connections, where the checksum is computed. Corresponding results for our UDP protocol stack are given in [23]. Where VPPC data are shown, the number of virtual processors is indicated by  $V$  in the legend. 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.

Several conclusions can be drawn from the data in Figure 3. The throughput for TCP 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 doesn't degrade significantly until there are 96 active connections. This is encouraging since presumably under conditions where connection sources are occasionally idle, the scheduling flexibility gained by increasing the number of virtual processors (within the limits of system resources) would only widen the performance improvement over that achieved by using fewer virtual processors.

So far we have only presented results 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

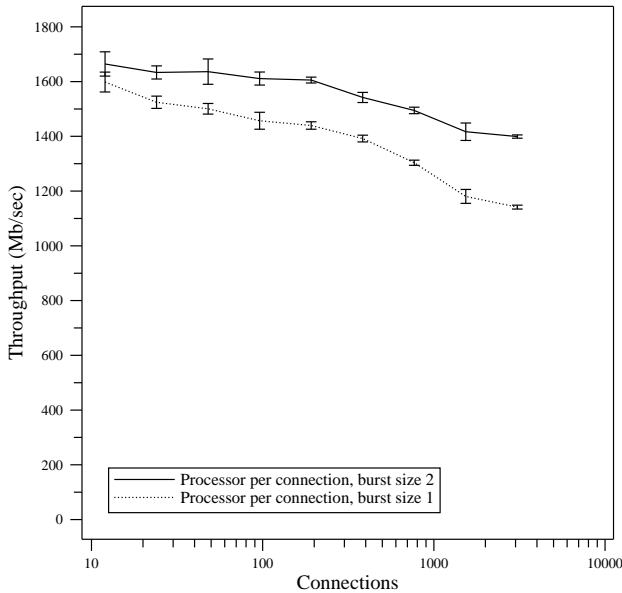


Figure 4: UDP Receive-Side Throughput from 12 to 3072 Connections.

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 (see, for example, [4]). Figure 4 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.

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 is not applicable if the latency of sending data objects is crucial (e.g., packet voice or video). Results presented in [23] show that throughput improves by 10-20% for TCP and up to 30% for UDP, using a burst size of 16.

### 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. With  $N$  connections, one might be tempted to assume that each connection receives approximately  $1/N$  (i.e., roughly its “fair share”) of this aggregate throughput, particularly when the per-connection 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-connection throughput received by different connections.

Clearly, the extent to which a connection receives its fair share of the total throughput is influenced by the manner in which connection protocol processing is scheduled. The processing of packets for connections assigned to the same virtual processor is performed in a round-robin manner. However, the scheduling of the threads running on virtual processors is controlled by the IRIX operating system. Since the IRIX time quantum is 30 ms, this means that each thread executing the processing associated with a virtual processor

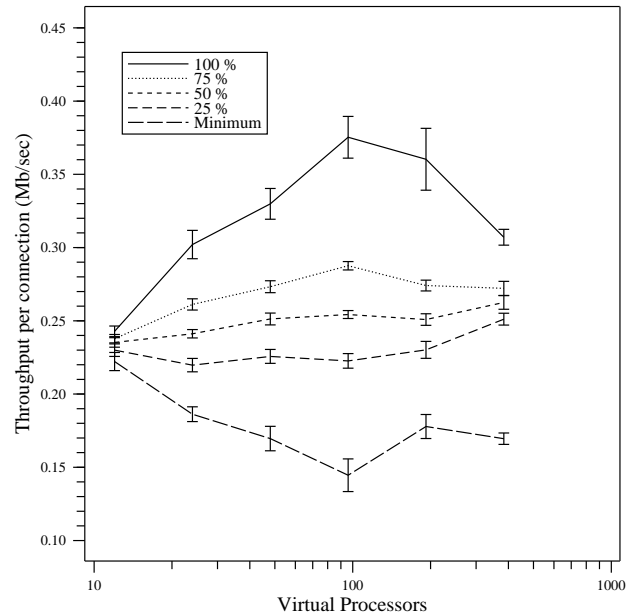


Figure 5: TCP Send-Side Throughput per Connection for 3072 Connections.

usually runs uninterrupted for 30 ms before being preempted. Our experiments were all run for 45 seconds, which is equivalent to 1500 IRIX time quanta.

Since all of the send-side experiments that we ran show the same trends, we report here on the results from one configuration consisting of TCP send-side processing with checksumming, and with 3072 connections.

Figure 5 plots the quartiles of per-connection throughput as a function of the number of virtual processors. The dotted line in Figure 5, for example, shows the throughput value such that 75% of the connections receive a throughput of less than or equal to the y-axis value. The spread between the five curves is thus a rough indication of the distribution of per-connection throughput seen by the 3072 connections.

Figure 5 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 connections. For example, with 96 virtual processors, the connection receiving the highest per-connection throughput receives approximately 2.5 times the throughput received by the connection with the lowest per-connection throughput. We found this quite surprising since with 96 virtual processors, 1500 scheduling time quanta (in a 45-second interval), and 12 processors, each virtual processor should have run for approximately 190 time quanta – long enough to have averaged out the performance differences one might expect in the throughputs received in the individual time quanta.

Figure 5 also illustrates that the differences in per-connection throughput are minimized when there are only 12 virtual processors. Thus, from a fairness standpoint, a smaller number of virtual processors is to be preferred. Recall from our discussion of Figure 3, however, that a smaller number of virtual processors results in lower throughput. For example, with 3072 connections, the aggregate throughput (as indicated in Figure 3) is 720 Mbps and 800 Mbps for 12 and 384 virtual processors, respectively. Thus, while 12 virtual processors provide a fairer allocation of throughput among connections, the aggregate throughput is lower. It is worth noting, however, that even with the overall increased throughput with 384 virtual processors, Figure 5 indicates some connections would still receive less throughput with 384 virtual processors than

with 12 virtual processors.

The fact that using a smaller number of virtual processors provides for a more equitable allocation of per-connection throughput is not surprising. Indeed, one would expect that 3072 connections would be serviced more fairly if 256 connections were scheduled round-robin on each of 12 virtual processors than if 8 connections were scheduled round-robin on each of 384 virtual processors.

## 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 throughput or fairness is more important. If throughput is more important, then thread per connection parallelism is best. 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 virtual processors in the system. If fairness is more important, then matching the number of virtual processors to the number of physical processors, while scheduling connections assigned to each virtual processor in a round-robin manner, yields the best fairness behavior. However, an improvement in fairness may come at the cost of aggregate throughput.

## Acknowledgements

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. The authors are grateful to Jim Salehi and Eric Brown for many hours of lively discussion about connection-level parallelism. 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] 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.
- [3] T. Braun and M. Zitterbart. Parallel transport system design. *Fourth IFIP WG 6.4 Conference on High Performance Networking*, pages 397–412, Dec. 1992.
- [4] 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.
- [5] A. Garg. Parallel STREAMS: a multi-processor implementation. In *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., Jan. 1990.
- [6] 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.
- [7] 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 1993.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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*, San Francisco, CA, Oct. 1993.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] D. Presotto. Multiprocessor streams for Plan 9. In *UKUUG*, Jan. 1993.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. Technical Report in preparation, URL = <ftp://gaia.cs.umass.edu/pub/Yate95:Networking.ps>, July 1995.
- [24] 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.