FURTHER RESULTS IN AFFINITY-BASED SCHEDULING OF PARALLEL NETWORKING

J.SALEHI, J. KUROSE, and D. TOWSLEY

CMPSCI Technical Report 95-46

July, 1995

Further Results in Affinity-Based Scheduling of Parallel Networking*

James D. Salehi James F. Kurose Don Towsley
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

July 24, 1995

Abstract

In this paper, we present further results in processor-cache affinity scheduling of parallel network protocol processing, in a setting in which protocol processing executes on the multiprocessor host concurrently with a general workload of non-protocol activity. In earlier work [31, 32] we evaluated affinity-based scheduling of receive-side protocol processing under two parallelization approaches: *Locking* and *Independent Protocol Stacks* (IPS). In this work, we *i)* evaluate affinity-based scheduling of send-side UDP/IP/FDDI processing, *ii)* examine the performance of affinity-based scheduling as a function of stream burstiness and source locality, *iii)* explore under IPS the impact of varying the number of independent protocol stacks, and *iv)* incorporate into our results the overhead of copying uncached packet data. We obtain our results following the research methodology developed in our earlier work, extending the developed infrastructure as necessary.

Our results show that affinity-based scheduling performs well for send-side processing, and is resilient to stream burstiness and source locality—two well-known properties of network traffic. We find that the performance of IPS improves dramatically when the number of stacks increases with the number of admitted streams, due to lower packet processing times and improved packet queueing behavior. Finally, we show that the benefit of affinity-based scheduling remains significant when processing includes the overhead of copying uncached packet data, in spite of the increased packet delay.

University of Massachusetts Technical Report UM-CS-1995-046, May 1995

*This work was supported by NSF under grant NCR-9206908 and by ARPA under ESD/AVS contract F-19628-92-C-0089. The authors can be contacted at [salehi,kurose,towsley]@cs.umass.edu.

1 Introduction

Processor-cache affinity scheduling is of growing interest as processor speeds continue to increase faster than memory speeds [7, 10, 22, 32, 38, 40]. On modern shared-memory machines, the time to access an uncached memory location is typically much larger than when accessing one cached locally. For example, on the SGI Challenge XL multiprocessor (the experimental platform used in our research) a memory reference can be serviced by the first-level cache in one processor cycle, whereas the fastest main memory access over the shared bus requires 130-180 cycles[1]. Such vastly different memory access times have given rise to "affinity-based" scheduling—choosing a processor to run a computation so that the generated memory references are likely to be found in that processor's cache, thus avoiding accesses to the slower main memory and resulting in faster execution times.

We study affinity-based scheduling of *parallel network protocol processing*, which has recently become an area of active research [3, 8, 12, 14, 17, 20, 24, 34, 35, 36, 41] and significant applied/commercial interest [8, 12, 29, 34]. The use of parallelism in protocol processing is motivated by the development of high-speed networks (such as ATM) capable of delivering gigabit-range bandwidth to individual machines. Emerging large-scale server applications, such as digital multimedia information repositories, require *application-level* access to such high bandwidth. Distributed applications, on the other hand, require very low latency network communication. Network parallelism in the host operating system—both within and among connections—can both increase the bandwidth and decrease the latency of multiprocessor communication.

While previous studies have explored the benefits of affinity-based scheduling of non-network-related application processing [7, 10, 22, 40], our work [31, 32, 33] is the first to apply the technique to operating system network processing. In our experimental environment (consisting of a parallelized *x*-kernel [13, 26] running in user-space on an 8-processor MIPS R4400-based SGI Challenge XL), packet execution times can vary by as much as a factor of *four*, depending on the state of the processor cache. This suggests that affinity-based scheduling presents a promising research opportunity in parallel networking. However, the analysis of the scheduling technique in this domain is potentially complex. There is a diversity of approaches to parallelizing protocol code, and a given implementation may combine several approaches[2]. To evaluate the performance of the scheduling technique in the presence of a general workload of non-protocol activity executing concurrently on the multiprocessor host, some mechanism is required to capture its impact on the cached protocol state. Finally, in contrast to previous work in affinity scheduling, the scheduling problem in parallel networking involves not only the concurrent management of protocol threads and available processors, but also of packets and the memory resources accessed during packet processing.

[1]Depending on the coherence state of the referenced cache line.

[2]In functional parallelism, an individual packet concurrently visits multiple processors [4, 17, 20]. In layer parallelism, packets visit multiple processors in a pipelined fashion [9, 30, 35]. Packet-level [3, 8, 12, 14, 15, 24, 29, 34, 35, 36] and connection-level [8, 12, 29, 34, 36] parallelisms enable concurrency at higher levels of granularity.

We present the following results.

- We study affinity-based scheduling of *send-side* UDP/IP/FDDI protocol processing (Section 6). In our earlier work, we identified two approaches to parallelizing receive-side protocol code—*Locking*, which enables protocol parallelism by protecting access to non-read-only data structures with software locks, and *Independent Protocols Stacks* (IPS), which implements multiple, independently-functioning stacks none of which individually support protocol concurrency. We explore the parallelization of send-side UDP/IP/FDDI processing, and show that affinity-based scheduling of send-side processing performs well.

- We evaluate the performance of affinity-based scheduling as a function of stream burstiness and source locality. In the earlier work, we assumed a workload of a number of Poisson streams. In this work we improve the source model, varying the number of streams (Section 4) and modeling individual streams with the Packet-Train model developed in [16] (Section 7). We find the impact of varying the number of streams to be significant under IPS, and that affinity scheduling performs well in the presence of stream burstiness and source locality.

- We explore under IPS the benefit of increasing the number of independent stacks with the number of admitted streams (Section 8), instead of matching the number of stacks to the number of processsors (as was done in our earlier work). We find that the performance of IPS improves dramatically, reflecting lower packet processing times and improved packet queueing behavior.

- We consider the impact of data-touching operations on our performance results (Section 5). Most network implementations perform a copy of packet data, an expensive operation which perturbs cached protocol state. We experimentally measure the impact of an uncached copy of packet data on packet execution time, and incorporate (using a published UDP/IP/FDDI packet-size distribution) the overhead into our multiprocessor simulation model. We find that the benefit of affinity-based scheduling remains significant in spite of the increased packet delay.

These results complement and strengthen our earlier work, further underscoring the importance of affinity-based scheduling in parallel networking.

To establish our results, we follow the research methodology developed in our earlier work, extending the developed infrastructure as necessary. For the send-side results, we conduct a set of multiprocessor experiments with the x-kernel's UDP/IP/FDDI protocol stack which are designed to measure packet execution times under specific conditions of cache state. These measurements are used to parameterize the analytic component of a simulation model of multiprocessor protocol processing, under various affinity-based scheduling policies. (The other results are obtained via appropriate modification to this simulation model). The analytic model accounts for the impact of general non-protocol activity on packet execution time, in terms of its displacement of the cached protocol

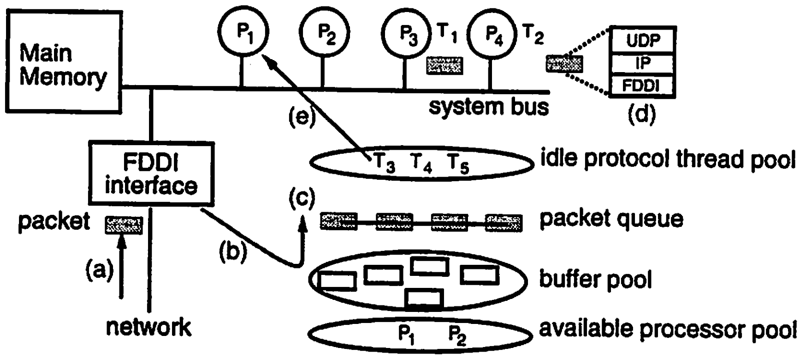


Figure 1: Conceptual model of multiprocessor protocol processing

footprint (i.e., set of cache lines referenced during protocol processing). Thus, the experimentally-parameterized analytic/simulation approach enables us to explore the benefits of affinity-based scheduling in a carefully-controlled experimental setting.

This paper is organized as follows. We begin by presenting the problem formulation in Section 2. In Section 3 we discuss the research methodology, summarizing the relevant aspects of the simulation and analytic models. Performance results are presented in Sections 4-8. We summarize the results in Section 9. For a discussion of related work in affinity-based scheduling, see [31, 32].

2 Problem formulation

2.1 Affinity-based scheduling of protocol processing

Consider Figure 1, which depicts a simplified view of in-kernel, receive-side protocol processing on a multiprocessor with FDDI network attachment. In the figure, processors P_3 and P_4 are running kernel-level protocol threads (i.e., threads executing a communication protocol or protocol stack), while the other processors are running non-protocol threads. For simplicity, we will assume that processors are kept busy executing non-protocol processing when not executing protocol code, that protocol processing receives priority over non-protocol processing, and that when unblocked a protocol thread is scheduled immediately.

Packets arriving from the network (a) are DMA'd to an in-kernel buffer (b) and added to the queue of packets (c) awaiting subsequent processing by some protocol thread T_i (d). In the context of this activity, there are several affinity-based scheduling opportunites.

- ***Code affinity*** (i.e., avoidance of cache misses on code and read-only data references[3]) is gained by scheduling

[3]Not all read-only references are eligible. For example, initial references to packet data (e.g., during a copy operation) necessarily result in cache misses, since the packet arrived in main memory via device DMA.

protocol processing where it more recently executed. The reason is that the hardware cache coherence mechanism allows read-only references to be replicated among processor caches, and the intervening non-protocol processing displaces this cached protocol state as it executes. Thus we consider (as code affinity scheduling) MRU management of the "available processor pool" of processors not currently executing protocol code, in selecting a processor when an additional protocol thread is scheduled (e). We also consider LRU management, to illustrate the performance lower bound.

- ***Thread stack affinity*** (i.e., avoidance of coherence-based misses on initial writes to the protocol thread stack area) is gained by organizing the "idle protocol thread pool" into per-processor lists, and selecting a thread from the appropriate pool when one is scheduled[4]. The key idea is that, for each write reference, the cache coherence protocol requires the processor to first gain exclusive ownership of the underlying cache line, resulting in a cache miss when the line was last written by some other processor. The protocol thread writes into its stack area for each packet; by organizing threads into per-processor pools, misses on the initial references are avoided. To illustrate the performance lower bound, we also consider a global organization managed MRU/LRU, under which thread stacks tend to migrate among processors.

- ***Stream affinity*** (i.e., avoidance of coherence-based misses on initial writes to per-stream data structures) is gained by "wiring" streams to processors. Under "Wired-Streams" processor scheduling, the processor selected when an additional protocol thread is scheduled is determined via a packet-filter operation based on the stream ID of the next packet to be processed. Note that stream affinity scheduling also influences what packet is selected when an executing thread dequeues its next packet.

- ***Free memory affinity*** (i.e., avoidance of coherence-based misses on initial writes to the memory resources allocated dynamically during packet processing) is gained by maintaining per-processor free-memory pools[5]. To illustrate the performance lower bound, we also consider a global free-memory pool, managed LRU; the overhead of migrating the underlying cache lines is incurred when some other processor last accessed the pool.

Table 1 summarizes the affinity-based scheduling objectives, the resources managed, and the scheduling policies we consider.

[4]Note that while the load-balancing and synchronization benefits of per-processor thread pools has been explored [2], the cache affinity benefits of such organization have not previously been evaluated.

[5]For simplicity, we assume that each free-memory pool is implemented as an array of pointers to segments of available memory. To acquire memory, the protocol thread increments the array index and returns a pointer to the allocated memory, which is subsequently written. To release memory, the thread decrements the index and stores the pointer to the memory being released.

Scheduling objective	*Resource managed*	*Policy*
Schedule for code affinity Schedule for stream affinity Performance lower bound (code affinity)	processors	MRU Wired-Streams LRU
Schedule for stack affinity Performance lower bound	threads	per-processor pools global pool
Schedule for free-memory affinity Performance lower bound	free-memory	per-processor pools global pool

Table 1: Scheduling protocol processing for cache affinity

2.2 Parallelization alternatives

To explore the performance of these affinity-based scheduling policies, we parallelized the receive-side fast path of the x-kernel's UDP/IP/FDDI protocol stack on our SGI Challenge XL multiprocessor. We began with an unparallelized x-kernel (version 3.2) running in user-space above the native IRIX 5.2 operating system; a simulated device driver emulates the protocol functionality associated with managing an FDDI network interface attachment. We developed "in-memory drivers" (a technique also used in [24, 36]), since the Challenge's eight 100MHz R4400 processors are together much faster than the single FDDI network attachment on our machine. Data is not actually received from (or sent on) the actual FDDI network.

To parallelize this x-kernel implementation, we first identified the non-read only data structures referenced during protocol processing.

- During packet processing, the protocol thread visits protocol modules (i.e., accesses and executes protocol code) and references stream-specific data structures known as *session objects*. At each protocol layer, a lookup in the protocol's *active map* (a data structure which records its active sessions) enables demultiplexing of the packet to the correct session. There, a session reference counter is incremented to register outstanding packet processing on the session.

- In the device driver, the protocol thread first builds an x-kernel "message structure" around the packet buffer written by the device interface. A portion of this structure is dynamically allocated, and subsequently released when the packet's processing is complete. We assume the protocol subsystem manages its own free-memory pools.

To support concurrency, accesses to the protocol active maps during demultiplexing operations and to session reference counters during increment or decrement operations must be protected. Consider the following two parallelization approaches.

- ***Locking*** enables parallelism by protecting each active map and session object with a software lock. While this

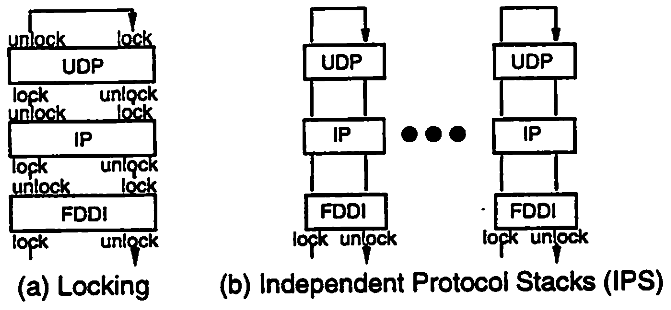


Figure 2: Abstract view of parallelization approaches

enables very flexible concurrency, 75% of the $2.5\mu s$ overhead incurred acquiring and releasing an uncached lock on our platform is due to migration of the underlying cache lines. This suggests that a parallelization approach which avoids software locks may yield higher performance.

- *Independent Protocol Stacks (IPS)* implements multiple, independently-functioning stacks none of which individually support protocol concurrency—and therefore do not require software locks[6]. The key idea is to identify the non-read-only *protocol* data structures in the unparallelized stack, and to create N copies of each. In the x-kernel, each protocol layer would be expanded to contain N active maps. Taken together, the data structures at index i ($1 \leq i \leq N$) constitute a complete protocol stack. Under IPS, session objects need not be duplicated: an individual stream is associated with one of the N independent stacks. A packet filter operation in the network device interrupt handler routes an incoming packet to the correct IPS.

  IPS avoids the cache-related overheads associated with software locks, yet does not allow concurrent processing of packets from streams mapped to the same stack—raising the possibility of higher packet delay.

Memory pool access must also be protected; we assume access to each pool is serialized via a dedicated software lock.

Figure 2 provides an abstract view of the parallelization alternatives. We have implemented both Locking and IPS through modifications to the original multi-threaded, uniprocessor x-kernel. These implementations form the basis for the experimental component of the work, discussed in the next section.

3 Research method

We use a combination of simulation, analytic, and experimental techniques to evaluate the performance of affinity-based scheduling in the context of Locking and IPS. In this section, we summarize the salient aspects of our approach; extensive details are provided in [31, 32].

[6] A single per-stack lock ensures serialization of its processing.

We begin by developing a multiprocessor simulation model that closely follows the behavior of Figure 1. To capture the displacement of the cached protocol footprint by the non-protocol workload, we develop an analytic model of packet execution time (combining established analytic results from other researchers [37, 39]) that reflects the specific cache architecture and organization of our SGI Challenge. We then conduct a set of multiprocessor experiments, designed to measure packet execution times under specific conditions of cache state, and parameterize the analytic model with the experimentally-measured values.

The benefit of this approach is that it enables us to explore, *in a controlled manner*, the performance of affinity scheduling in a setting in which protocol processing executes concurrently with a general workload of non-protocol activity. In addition, the approach—which is based on timing measurements—does not require identifying the footprint of the task being affinity scheduled (as is assumed, e.g., in [38]), and thus avoids the difficulties inherent in capturing memory traces from large parallel applications.

3.1 Simulation model of multiprocessor networking

Consider first the simulation of Locking, under which there are N processors and N protocol threads. Arriving packets are queued in a global (per-processor) queue under code (stream) affinity scheduling, and served FIFO.

A protocol thread is scheduled immediately if possible. Under code affinity scheduling, a processor is selected if any is free (i.e., not executing protocol code); under stream affinity scheduling, the destination processor is determined via a packet-filter operation, and selected if free. A thread is selected from the per-processor (global) pool when threads are (are not) affinity scheduled.

The analytic model is used to compute the protocol processing time, and the packet enters service. Upon completion, the protocol thread continues with the next packet from the global (per-processor) queue under code (stream) affinity scheduling, if any; otherwise, the processor is released to non-protocol processing. Thus protocol processing is work conserving under code affinity scheduling, but non-work conserving under stream affinity scheduling.

For IPS, arriving packets are queued in per-IPS queues. Protocol processing is non-work conserving under all scheduling policies.

3.2 Analytic model of packet execution time

Consider Figure 3, which presents a simplified view of the memory architecture of our SGI Challenge. Each R4400 processor has separate on-chip 16KB first-level instruction and data caches (L1), and a private 1MB unified second-level cache (L2). When a memory reference is issued, L1 is first checked (a); on a miss, L2 is checked (b). A miss in L2 generates a request sent over the system bus (c), which is satisfied by the owner of the cache line—either another

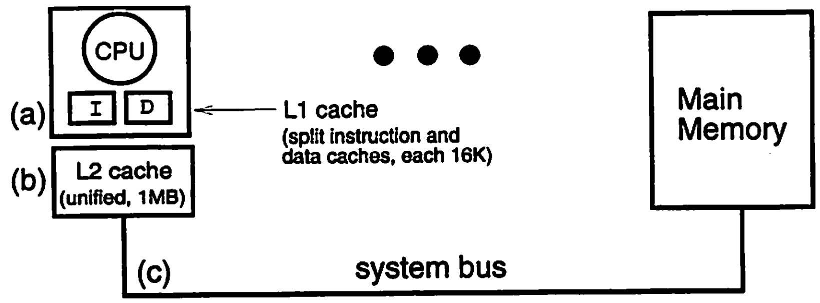


Figure 3: A simplified view of the memory hierarchy of the SGI Challenge

processor's cache, or main memory.

The analytic model requires three timing measurements.

- $t_{hot,\ hot}$, the time to process a packet immediately following the completion of the previous packet (i.e., in the absence of intervening processing, when both levels of the cache hierarchy are "hot");
- $t_{cold,\ hot}$, the time to process a packet when L1 is "cold" (i.e., entirely flushed of the protocol footprint), but L2 is hot;
- $t_{cold,\ cold}$, the time to process a packet when the protocol footprint is entirely flushed from both levels of the cache hierarchy.

Each timing represents a bound on protocol execution time with respect to the overhead migrating the protocol footprint from the given layer in the cache hierarchy. $t_{hot,\ hot}$ corresponds to case of minimal overhead migrating the protocol footprint; $t_{cold,\ hot}$ and $t_{cold,\ cold}$ measure the maximum overhead of migrating the footprint from L2 and main memory, respectively. To reflect the parallelization alternative (IPS or Locking) and the overhead of coherence-based cache misses (i.e., migration of the thread stack, protocol-specific, and session-specific cache lines), a set of timing measurements is obtained for each possible combination of these variables. The appropriate set of timing measurements is selected before the packet execution time is computed[7].

The non-protocol workload displaces the cached protocol footprint at each layer of the cache hierarchy. To capture this displacement, the model first computes (using analytic results from [37, 39]) the fractions $F_1(x_i)$ and $F_2(x_i)$ of the footprint that have been flushed from L1 and L2, respectively, given that non-protocol processing has executed for time x_i since P_i last executed protocol code. Details are provided in [31, 32]. $F_1(x_i)$ and $F_2(x_i)$ are formulated to reflect the cache architecture and organization of our SGI Challenge.

[7]The migration overhead incurred by P_i when accessing a global free-memory pool is incorporated in a slightly different manner. The simulation keeps track of the last processor P_{last} to access the pool; whenever $P_i \neq P_{last}$, the access overhead is added to the computed packet execution time.

Finally, the packet execution time t_i is computed by scaling the protocol execution time bounds by the fraction of the footprint which remains at each layer in the cache hierarchy:

$$t_i(x_i) = (1 - F_1(x_i))t_{hot,\,hot} + F_1(x_i)\left[(1 - F_2(x_i))t_{cold,\,hot} + F_2(x_i)t_{cold,\,cold}\right]. \tag{1}$$

To summarize, the parallelization approach and coherence-based migration overheads are incorporated into t_i by selecting the appropriate set of timing bounds. The impact of the non-protocol workload is captured by scaling these bounds by the fraction of the protocol footprint found at each corresponding layer in the cache hierarchy[8].

To acquire the timing bounds, we measured the time to process a packet in a set of experiments on our multiprocessor in which we varied the scheduling of protocol threads and explicitly manipulated the processor caches. The receive-side experimental design and resulting measurements are presented in extensive detail in [31, 32]. The send-side experimental analysis, which is similar but simpler, appears in Section 6 of this paper.

We now turn to the first set of set of performance results.

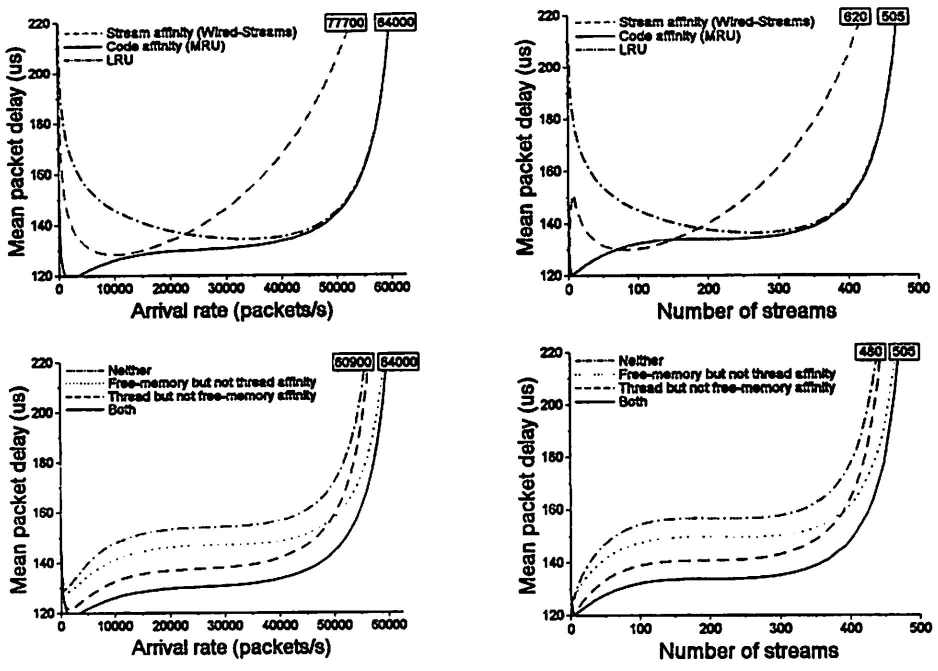
4 Stream scaling

We begin by examining how affinity-based scheduling performs as the number of streams in the protocol stack varies. In [32], we fixed the number of streams at eight, and examined affinity-based scheduling under *packet scaling*—by varying the mean packet rate of the eight individual streams. In this work, we relax the assumption of a fixed number of streams and examine affinity-based scheduling under *stream scaling*—as a function of the number of admitted streams, while, for simplicity, holding the per-stream packet rate constant. Stream scaling is a more realistic paradigm in the sense that most multiprocessor server applications (e.g., file servers, web servers) must support a varying and potentially large number of concurrent streams.

In the following two subsections, we evaluate stream scaling, compare it to the packet-scaled results of [32], and discuss the reasons behind the differences. Under stream scaling, we set the individual stream packet rate to 125 packets/s. The inter-packet delay distribution is taken to be exponential (as in the packet-scaled results of [32]) to enable a meaningful comparison.

As background for understanding the impact of stream scaling, consider the following. Each protocol stack contains (at each protocol layer) fast-path demultiplexing support for the anticipated common case of network source locality. That is, the FDDI, IP, and UDP active maps each contain a "table cache" holding the key and resolution pair of the most recent active map lookup (i.e., demultiplexing operation at that layer). In the common case that

[8]Task execution time as the linear interpolation of the maximum "reload transient" is also the approach taken in [38]. In the authors' formulation, the inherent computing demand of a task is denoted D, the average time to reload the entire footprint is C, and the fraction of the footprint displaced is R. The task execution time is modeled as $D + RC$.



(a) Packet scaling

(b) Stream scaling

Figure 4: Impact of stream scaling under Locking

subsequent packets are from the same connection [16, 23], the packet is demultiplexed based on the table cache value, and the active map lookup is avoided. But when the stream ID changes from packet to packet, the active map lookup must be performed and a new table cache value written—incurring a coherence-based cache miss when the table cache value was last written by some other processor.

4.1 Locking

Under Locking, there is a single, global protocol stack. As the number of streams becomes large under stream scaling, the fraction of packets which benefit from the fast-path optimization becomes small. Similarly, under packet scaling with N Poisson steams, the fast-path optimization benefits on average only $1/N$ of the packets. Thus, we can expect that under Locking the impact of stream scaling should be relatively small, since $N = 8$ in the earlier study—and 1/8 of the overhead avoided by the fast-path optimization (about $25\mu s$ [32]) is only $3\mu s$.

Figure 4 compares affinity scheduling under packet scaling (4a) and stream scaling (4b). The upper graphs show code and stream affinity scheduling; the lower graphs show thread and free memory affinity scheduling[9]. The axes have been aligned to allow direct comparison of the graphs. In the upper right of each graph, the maximum supportable throughput for each policy is indicated (e.g., 64,000 packets/s under code affinity scheduling in Figure

[9]The graphs in Figures 4a and 5a are borrowed from [32].

4a).

It is evident that, in general, the impact of the varying number of streams is small under Locking. Code affinity scheduling is impacted the most (rising above stream affinity scheduling in range of 70-140 streams) since the fast-path optimization is experienced most frequently under that scheduling policy.

A word on the asymptotic behavior of the affinity scheduling policies under Locking.

- Code affinity is achieved by default at high arrival rate, since processors execute protocol processing nearly all of the time. Thus, the MRU and LRU scheduling policies in Figure 4 converge. Stream affinity scheduling further ensures cached per-stream data structures, which reduces packet latency and increases the maximum number of supportable streams. In Figure 4b, stream affinity scheduling establishes the maximum number of streams at 620, whereas code affinity scheduling supports at most 505 streams.

- Thread affinity scheduling does not increase maximum throughput, since threads are not rescheduled at maximum load. Thus, the corresponding curves converge at high arrival rate. Since free-memory affinity scheduling reduces packet latency at high arrival rate, it increases the maximum number of supportable streams (in Figure 4b by 5%, from 480 to 505).

4.2 IPS

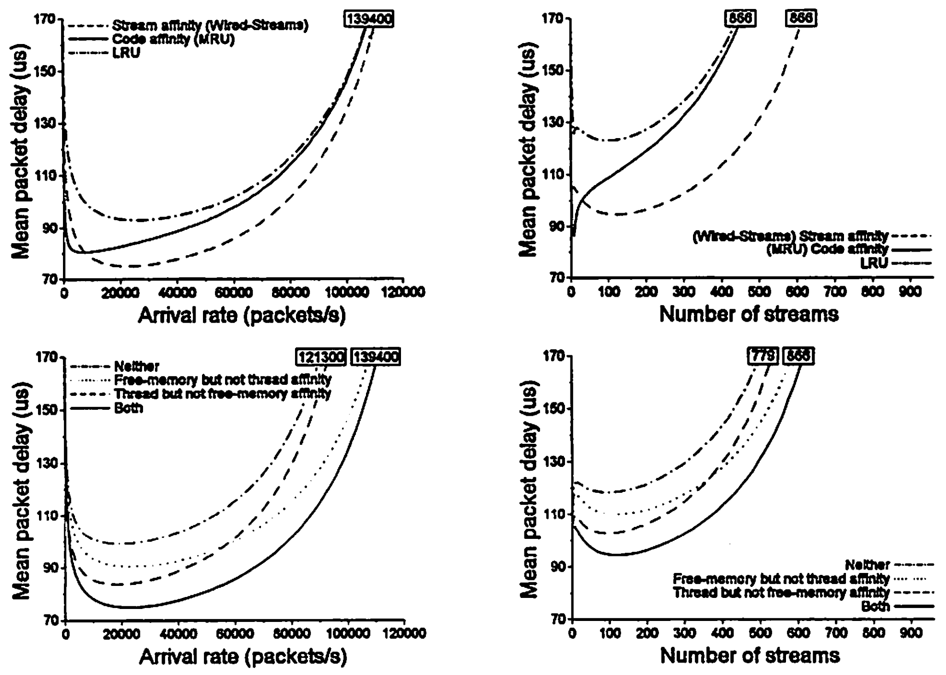
In [32], we matched the number of independent stacks with the number of processors (eight in that study) to enable all processors to work concurrently under IPS. For simplicity, we also set the number of streams to be eight. Since this resulted in one stream per stack, the fast-path optimization was realized with every packet. Yet under stream scaling (with eight processors and eight stacks), the optimization would be realized with diminishing frequency as the number of streams per stack becomes large. This reasoning suggests that stream scaling under IPS should look very different than the previous packet-scaled results.

Figure 5, which compares the performance of affinity-based scheduling under packet and stream scaling for IPS, confirms this hypothesis. The per-packet delay is about $20\mu s$ larger under stream scaling across all policies, a significant (i.e., 20-30%) fraction.

A word on the asymptotic behavior of the scheduling policies under IPS, when the number of stacks matches the number of processors.

- Unlike Locking, code and stream affinity scheduling converge at high arrival rate, as the eight independent stacks are rescheduled less and less frequently.

- As under Locking, thread scheduling does not increase maximum throughput, but free-memory affinity



(a) Packet scaling (b) Stream scaling

Figure 5: Impact of stream scaling under IPS

scheduling does. In Figure 5b, free-memory affinity scheduling increases the maximum number of supportable streams by 14%, from 779 to 886.

The large increase in packet latency under stream scaling raises the question of whether there is some way it can be avoided. One solution is to increase the number of independent stacks with the number of streams, instead matching it to number of processors (as we have done so far). We consider this alternative in Section 8.

5 Data-touching operations

So far, we have presented results without data-touching operations (e.g., copying, checksumming), motivated by the fact that packet execution time in many real environments is dominated by "non-data touching" operations with generally fixed per-packet overheads. For example, Kay and Pasquale [18] show that 84% of the processing time for TCP packets and 60% for UDP packets in their FDDI LAN environment is attributed to the non-data touching operations of protocol stack operations, buffer management, and OS overheads. Although copying and checksumming (which scale with packet size) are expensive, the reason the non-data touching overheads predominate is that typically in real environments most packets are small [6, 11, 18, 23][10].

[10]Some common applications do not touch the data *at all*. For example, the SGI implementation of the NFS server takes advantage of the fact that most SGI network interfaces support checksumming in firmware [25]. By DMA'ing data directly to the network interface, the server

There are good reasons, however, for explicitly extending our results by incorporating data copying overheads. Most implementations copy packet data between user and kernel space. The copy incurs cache misses in the common case[11], and is therefore expensive. Further, it perturbs cache state—impacting the caching behavior and execution time of protocol code.

In this section we extend our earlier work by incorporating copying of uncached packet data. In order to capture the impact of the copy on cached protocol state (and therefore on protocol execution time), we assess the overhead experimentally in the context of our developed infrastructure. We also discuss the checksumming operation, and why we have chosen to not incorporate it into the measurements.

5.1 Copying

We modified our *x*-kernel implementation to include a copy of B bytes of uncached packet data, with B ranging between 1 and 4KB. To ensure the data is uncached, the referenced memory locations are written by another processor prior to protocol processing for the packet. We repeated the timing experiments, varying B from 1 to 4KB, and found that the receive-side timings presented in [32] (as well as the send-side timings presented below in Table 3) scale up nearly linearly at a rate of 0.04μs per byte.

Incorporating this per-byte overhead into our simulation model requires a packet-size distribution. We chose the empirical UDP/IP/FDDI distribution published by Kay and Pasquale in [18]. The authors gathered this distribution from a traffic trace taken from their departmental FDDI LAN, which supports mostly workstations and file servers. Ninety percent of the packets in this FDDI trace are UDP packets, most generated by NFS; thus, the empirical data is well-matched to the protocols involved in our study. Kay and Pasquale establish that the trace is representative by comparing the empirical distribution with that of LAN traffic in another departmental environment, finding very similar results. In addition, they note that observed packet size distributions match those of Ethernet-based packet traces reported in [6, 11, 23].

The packet size distribution, taken from Figure 5b of [18], is shown in Table 2. In formulating this distribution, we have assumed that each 8KB UDP packet is sent as two 4KB FDDI frames. The simulation was modified to sample the distribution independently for each packet sent; the copy overhead of 0.04μs/byte was then added to the packet execution time. The average uncached copy overhead, based on the mean packet size in Table 2 of 696 bytes, is 27.84μs.

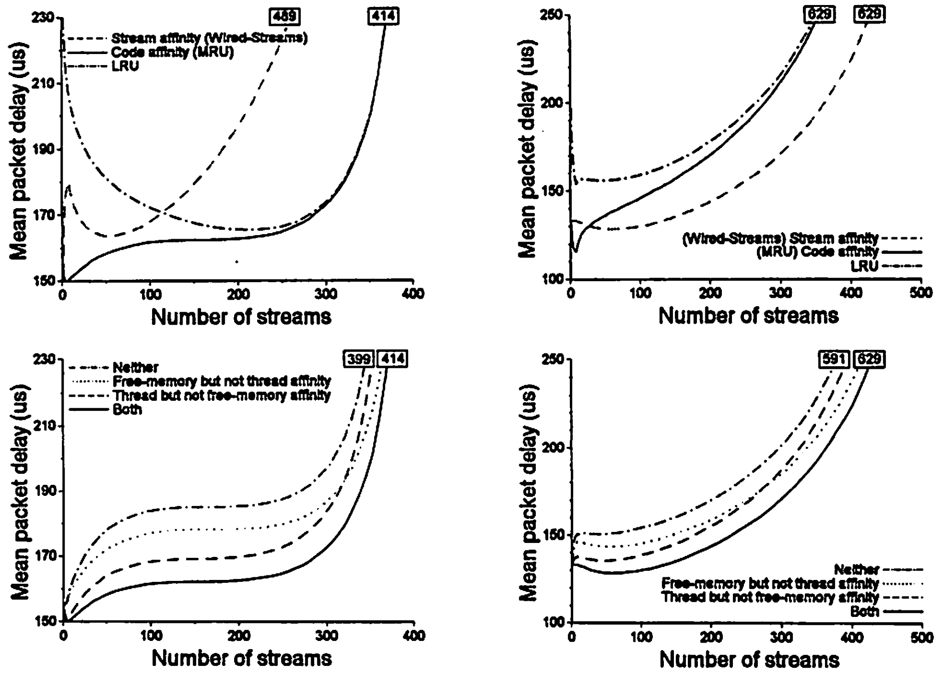
Figure 6 shows the simulation results. (For Locking, compare Figures 6a and 4b; for IPS, 6b and 5b). Although the copy increases the mean per-packet delay by about 30μs, the benefit of affinity scheduling remains significant.

avoids bringing the data into the CPU cache.

[11]The copy is always uncached on the receive side, since the packet arrived in memory via device DMA. Whether it is cached or uncached on the send-side depends on the application caching behavior.

Frame size (bytes)	% Total frames	Frame size (bytes)	% Total frames
28	2.8	98	0.9
40	4.7	128	43.4
48	4.7	320	1.9
56	3.8	470	0.9
60	0.9	2400	0.9
92	20.8	4096	14.2

Table 2: FDDI frame size distribution



(a) Locking (b) IPS

Figure 6: Affinity scheduling of receive-side processing with uncached copy of packet data

It is interesting to note that, under Locking, code now outperforms stream affinity scheduling across the broad range of arrival rates (contrast with Figure 4b), due to increased packet queueing.

The overhead of copying uncached data is incorporated into all results in the remainder of this paper.

5.2 Checksumming

The cost of checksumming cached data on our platform is $0.031\mu s$/byte [24]. Although UDP supports checksumming we have not incorporated this overhead into our results. In fact, there are several techniques for reducing or avoiding the overhead of checksumming in UDP (and TCP) [18]:

- Checksumming can be combined with data copying in *integrated layer processing* (ILP) [5, 1]. On our platform (given the relative overheads), it is likely that ILP would completely hide the checksumming overhead in the memory access latency of the uncached copy[12].
- The hardware interface can support TCP or UDP checksumming, as is generally done in Silicon Graphics interfaces [19, 25].
- Checksumming can be disabled when redundant with the network interface's cyclic redundancy check (CRC).

This last point is developed in detail by Kay and Pasquale in [19]. In that study, the authors argue that the UDP/TCP checksum operation is usually redundant: since most network interfaces (e.g., Ethernet, FDDI) support hardware CRC, and since most traffic is local to the LAN[13], in the common case UDP and TCP packets are fully protected by the interface CRC. Kay and Pasquale present an algorithm for disabling UDP/TCP checksum under these conditions. In the UDP implementation, a flag is added per network interface indicating whether hardware CRC is supported. During UDP processing, the route structure is consulted; the checksum is skipped when the next hop is the destination host and the interface supports a hardware CRC.

6 Send-side processing

In [32], we explored affinity-based scheduling of *receive-side* UDP/IP/FDDI protocol processing. How does the scheduling technique apply to send-side processing, and what is the comparative performance? To explore this, we revisited the unparallelized x-kernel implementation. We examined the send-side execution path from the top of the stack down through the UDP, IP, VNET[14] and FDDI protocols and into the SIMFDDI simulated device driver. The

[12]Thanks to David Yates for pointing this out.
[13]Substantiated in [19] by LAN traffic measurements.
[14]VNET is a "virtual" protocol for managing multiple link-layer protocols, such as FDDI.

goal was to identify the protocol and stream-specific data structures written during a send operation, as a first step toward developing send-side IPS and Locking implementations.

To perform a send, the protocol thread sequentially references a linked list of stream-specific application, UDP, IP, VNET, and FDDI session objects. Each contains a pointer to protocol code (e.g., `udp_send()`). After completing SIMFDDI processing, the protocol thread unwinds its runtime stack, thus completing the send operation.

On this execution path, the protocol thread references *only* read-only protocol-specific and session-specific data structures[15]. Thus, there is no need to support concurrency through software locks (à la Locking) or through data structure replication (à la IPS), and the "unparallelized" implementation can support concurrency immediately[16]. Thus on the send-side, there are no Locking or IPS analogs.

We consider the same affinity-based scheduling policies on the send-side as on the receive-side, with the following caveat. On the receive-side, stream affinity scheduling is a reasonable alternative, since it lowers per-packet latency by ensuring affinity for the stream-specific lines written during protocol processing. However, on the send-side (since no stream-specific state is written) it offers no benefit. In the figures which follow, we plot the stream affinity curve strictly for informal comparison with the receive-side results.

We instrumented the x-kernel implementation to deliver the timing measurements needed by the analytic component of the simulation model (Section 3). Specifically, we measured two $t_{hot,\ hot}$ and $t_{cold,\ hot}$ values (both with and without thread stack migration), one $t_{cold,\ cold}$ value, and the overheads of acquiring and releasing memory resources from a global pool when some other processor last accessed the pool. To acquire these measurements, we performed a set of experiments on our multiprocessor in which a hardware timer with microsecond accuracy is used to time send-side processing in the x-kernel's UDP/IP/FDDI stack. The experiments each involve one *control thread* and one *send thread*, and have the general structure depicted in Figure 7. The two threads operate in a lockstep manner, synchronizing via an IRIX semaphore.

The control thread runs on processor P_0. For experiments which require the sending thread's stack and/or free memory data structures to be uncached when the packet is sent, the control thread first writes the data structures, ensuring they are uncached at the sending processor. It then signals the sending thread, and waits on the semaphore. The sending thread is wired to P_1. Upon a signal from the control thread, it starts its timer, sends the packet, unwinds its stack, stops the timer, signals the control thread, and waits on the semaphore. For $t_{cold,\ hot}$ timings, the sending thread flushes its L1 cache before starting the timer[17]. Finally, $t_{cold,\ cold}$ is measured as the time for the first packet

[15]While this is in stark contrast to the receive-side path, consider the non-read-only data structures accessed during a receive operation. The protocol-specific data structures are the active maps, used for demultiplexing in the FDDI, IP, and UDP protocol layers; yet no demultiplexing occurs on the send-side. Similarly, on the receive-side the stream-specific data structures are the session reference counters, which facilitate garbage collection of open sessions; however, reference counters are unneeded on the send-side because the sending application controls session deallocation.

[16]Of course, each free memory pool must be protected by a software lock.

[17]The sending thread flushes the separate, virtually-indexed 16KB instruction and data caches at L1 by sequentially referencing 16KB of

Control thread (on processor P_0)

```
forever {
   write global memory pool (optional)
   write sending thread stack (optional)
   signal sending thread
   await signal from sending thread
}
```

Sending thread (on P_1)

```
forever {
   await signal from control thread
   flush L1 (optional)
   start timer
   send stack (UDP/IP/FDDI)
   unwind stack
   stop timer
   signal control thread
}
```

Figure 7: Thread behavior in the experimental environment

sent.

In an individual run, the control thread wires itself immediately to P_0. The sending thread sends a packet from each of P_1, P_2, ..., P_7, delivering seven $t_{cold, cold}$ timings. The sending thread then wires itself to P_1, and the run yields the mean time over 1000 sent packets for each of the remaining measured values. A sufficiently large number of independent runs (100 in our experiments) are performed to ensure that the 95% confidence interval half-widths do not exceed 1% of the overall mean, for all values.

The obtained timing measurements are shown in Table 3. (The corresponding receive-side numbers from [32] are presented for comparison purposes). The timings suggest the importance of code affinity scheduling of send-side processing. Migrating the protocol footprint from the second-level cache increases the execution time by an average of 38%, and migrating from main memory by 300%[18] (compared to 24% and 160%, respectively, on the receive-side). The timings also suggest the importance of thread affinity scheduling: migrating the thread stack on the send-side increases $t_{hot, hot}$ and $t_{cold, hot}$ by an average of 21% (compared to 18% for the receive-side). Finally, note that the overhead of acquiring free memory from the global pool when the access and acquired memory are uncached is much larger on the send-side than the receive-side, reflecting the fact that multiple buffer structures are allocated in the *x*-kernel when a packet is sent.

Figure 8 shows the simulation results. In general, the graphs confirm the effectiveness of affinity-based scheduling of send-side processing: at low arrival rates, code affinity scheduling can reduce per-packet latency by about 10%, whereas thread and free memory affinity scheduling can reduce latency by about 10% and 20% respectively. Free memory affinity scheduling increases the maximum number of supportable streams by about 25%.

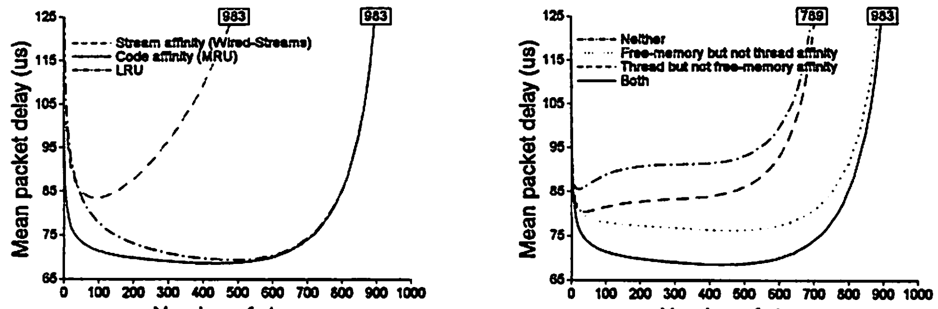
Although Figure 8 in comparison to Figures 4 and 5 indicates that send-side and receive-side affinity scheduling

code, then cycling through a 16KB data array.

[18]That is, the mean $t_{cold, hot}$ is 38% higher than the mean $t_{hot, hot}$, and $t_{cold, cold}$ is 300% higher than the mean $t_{hot, hot}$.

Timing	Thread stack	Send-side (μs)	Receive-side (μs) (overall mean)
$t_{hot,\,hot}$	cached	37.3	90.6
	migrates	47.2	110.8
$t_{cold,\,hot}$	cached	53.7	116.8
	migrates	62.0	133.8
$t_{cold,\,cold}$		168.3	258.3
global pool *acquire* overhead		15.6	4.1
global pool *release* overhead		4.5	4.5

Table 3: x-kernel UDP/IP/FDDI send times, for a single packet without data-touching operations.



(a) Code and stream affinity scheduling

(b) Thread and free-memory affinity scheduling

Figure 8: Affinity scheduling of send-side protocol processing, with copying of uncached packet data

are qualitatively similar, there are two notable distinctions.

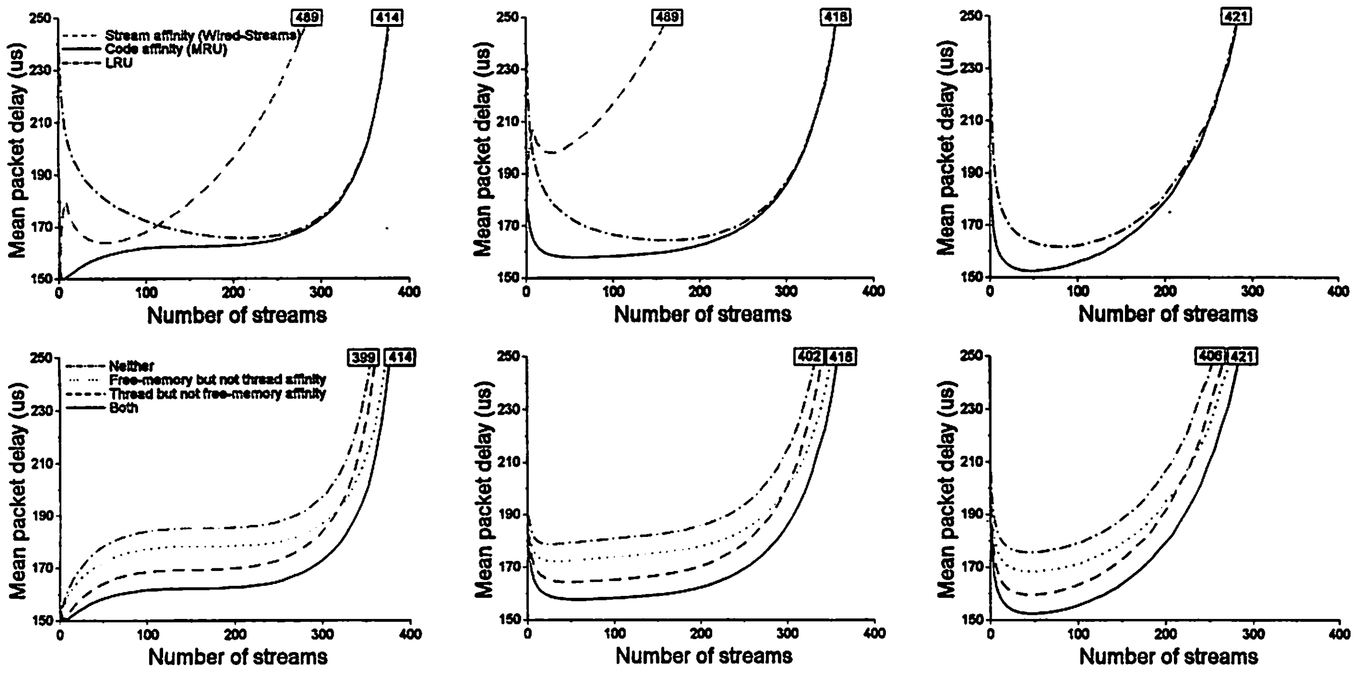
- On the send-side, because stream affinity scheduling does not lower packet latency, it converges with code affinity scheduling at high arrival rate. On the receive side, stream affinity scheduling increases maximum throughput by about 19% (Figure 6).
- Free-memory scheduling increases maximum throughput by a larger fraction on the send-side.

7 Stream burstiness and source locality

We now turn our attention to refining the stream model. While the Poisson model in [32] was a reasonable starting point, it is well-known that network traffic is generally not Poisson [6, 11, 16, 21, 27, 28]. The Poisson process is not very bursty (its coefficient of variation, an informal measure of burstiness, is just 1), nor does the independent Poisson model capture the related property of *source locality*, a well-known aspect of network traffic [16, 23]. How do burstiness and source locality impact the performance of our affinity scheduling results? In answering this question, our goal is not to evaluate affinity-based scheduling for a *specific* workload, but rather to refine the per-stream model and vary its burstiness parameters—toward showing that the demonstrated performance of the affinity-based scheduling policies is relatively insensitive to these arrival process characteristics.

To capture stream burstiness and source locality, we refine the per-stream arrival process to the "Packet-Train" model developed by Jain and Routhier [16]. Each stream is modeled as sending bursts of packets, where the burst size B (in packets) has distribution $\mathcal{B}$ with mean $\bar{B}$. The time I between packets in a burst has distribution $\mathcal{I}$ with mean $\bar{I}$; the time T between bursts has distribution $\mathcal{T}$ with mean $\bar{T}$. We assume that $\bar{T}$ is several orders of magnitude larger than $\bar{I}$, in line with Jain and Routhier's observations.

Below, we compare the performance of affinity-based scheduling across mean burst sizes $\bar{B}$=1,2,8 under constant burst size distribution $\mathcal{B}$. All streams have a mean inter-packet delay of $\bar{I} = 80\mu s$, which is approximately the protocol processing time for a single packet. We assume $\bar{T} = 8000$ when $\bar{B} = 1$, so the mean inter-burst time is two orders of magnitude larger than the mean inter-packet time. To enable a meaningful comparison across values of $\bar{B}$, when varying $\bar{B}$ we maintain the mean per-stream arrival rate by setting $\bar{T} = 8000\bar{B} - (\bar{B} - 1)\bar{I}$. Thus, for $\bar{B} = 8$, $\bar{T} = 63440$ is about three orders of magnitude larger than $\bar{I}$. Finally, we assume exponential inter-packet and inter-burst delay distributions for $\mathcal{B}$ and $\mathcal{I}$.



(a) $\bar{B} = 1$ (b) $\bar{B} = 2$ (c) $\bar{B} = 8$

Figure 9: Impact of stream burstiness on affinity scheduling under Locking

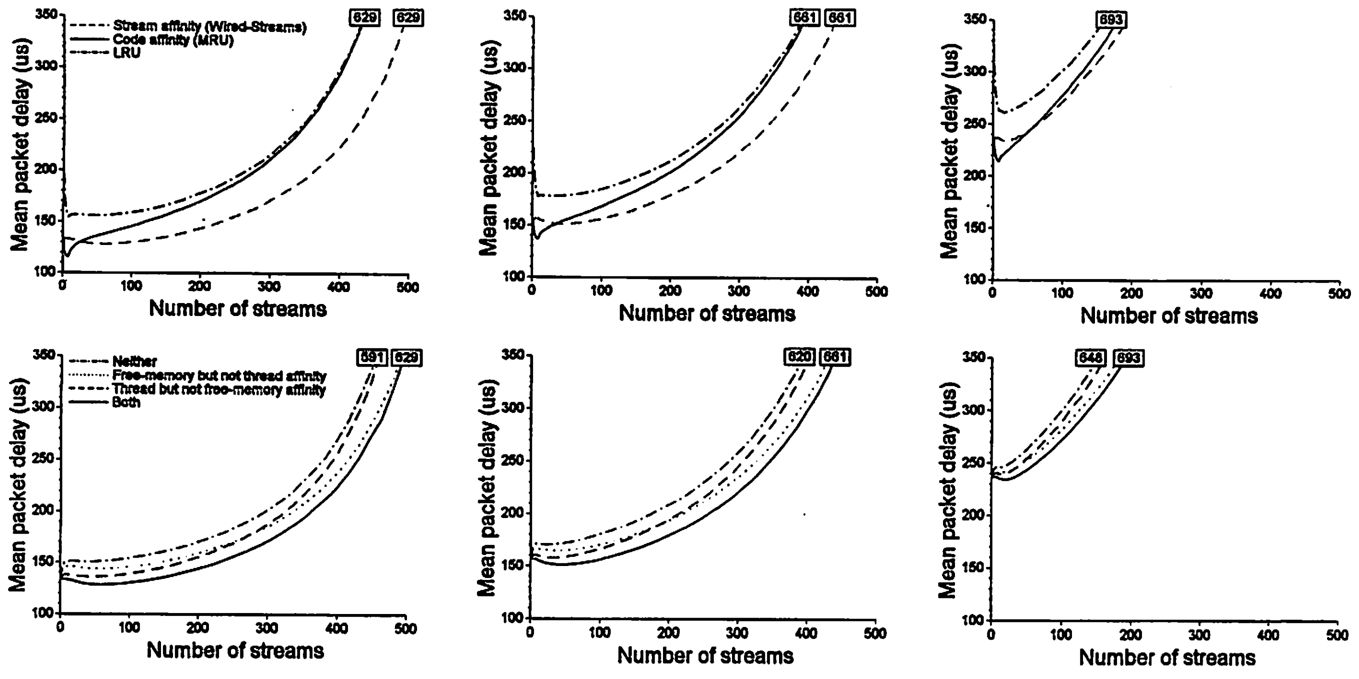
7.1 Receive-side processing

7.1.1 Locking

In Figure 9, we see that the general behavior of affinity scheduling under Locking is maintained across increases in stream burstiness. While the benefit of code affinity scheduling remains fairly consistent as $\bar{B}$ increases, the performance of stream affinity scheduling degrades rapidly due to increased packet queueing[19]. The stream affinity curve for $\bar{B} = 8$ has risen so high as to be off the scale in Figure 9c; its behavior is studied later in the discussion of Figure 11.

Note that the code affinity asymptote increases with $\bar{B}$ since the (uncached) fast path is hit more frequently as packets from the same stream are processed concurrently across processors. The free-memory affinity asymptote also shifts with $\bar{B}$, since those curves are computed under code affinity scheduling. By contrast, the stream affinity asymptote remains fixed as $\bar{B}$ varies, since under stream affinity scheduling the fast path hit rate at high load is zero.

[19]Actually, as $\bar{B}$ increases from 1 to 2, the packet execution time under stream affinity scheduling at low load drops from 130 to 110 μs. But the packet queueing time rises about 40μs, more than offsetting the execution-time decrease.



(a) $\bar{B} = 1$ (b) $\bar{B} = 2$ (c) $\bar{B} = 8$

Figure 10: Impact of stream burstiness on affinity scheduling under IPS

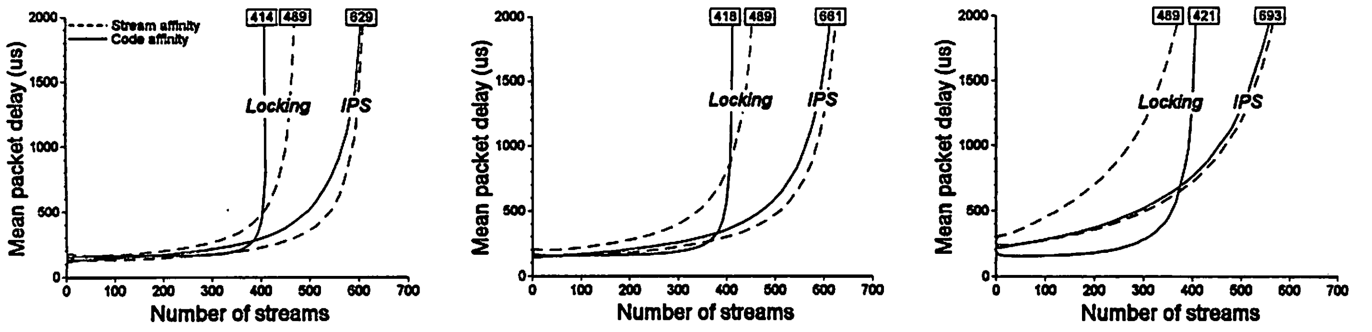
7.1.2 IPS

Figure 10 shows the behavior of affinity-based scheduling under IPS across increases in stream burstiness. While the qualitative behavior of the scheduling policies is maintained, all curves rise rapidly (as did the stream affinity curve under Locking) since IPS imposes serialized processing on each of the eight independent stacks.

The code and stream affinity scheduling asymptotes shift with increasing $\bar{B}$, since the per-stack fast path optimization is hit more frequently under burstier traffic. The asymptotic behaviors of thread and free memory affinity scheduling also display this behavior.

7.1.3 Comparing Locking with IPS

How do burstiness and source locality impact the relative performance of Locking and IPS? In Figure 11, we compare code and stream affinity scheduling (with threads and free-memory affinity scheduled) across values of $\bar{B}$. The most prominent aspect is that IPS supports a much higher maximum number of streams in all cases. However, with $\bar{B}$ sufficiently large, code affinity scheduling under Locking offers lower per-packet latency than under IPS. If the mean inter-packet delay $\bar{I}$ were smaller, the latency under IPS would rise above Locking even sooner (i.e., with respect to increasing $\bar{B}$). In [32], we considered Compound Poisson arrivals, which represent the $\bar{I} = 0$ case: Figure 13



(a) $\bar{B} = 1$ (b) $\bar{B} = 2$ (c) $\bar{B} = 8$

Figure 11: Comparing Locking with IPS

in that paper shows the latency to be lower under Locking for all values of $\bar{B}$ greater than 1. Note however that back-to-back arrivals (which the Compound Poisson essentially models) occur rarely in real networks [11, 16].

7.2 Send-side processing

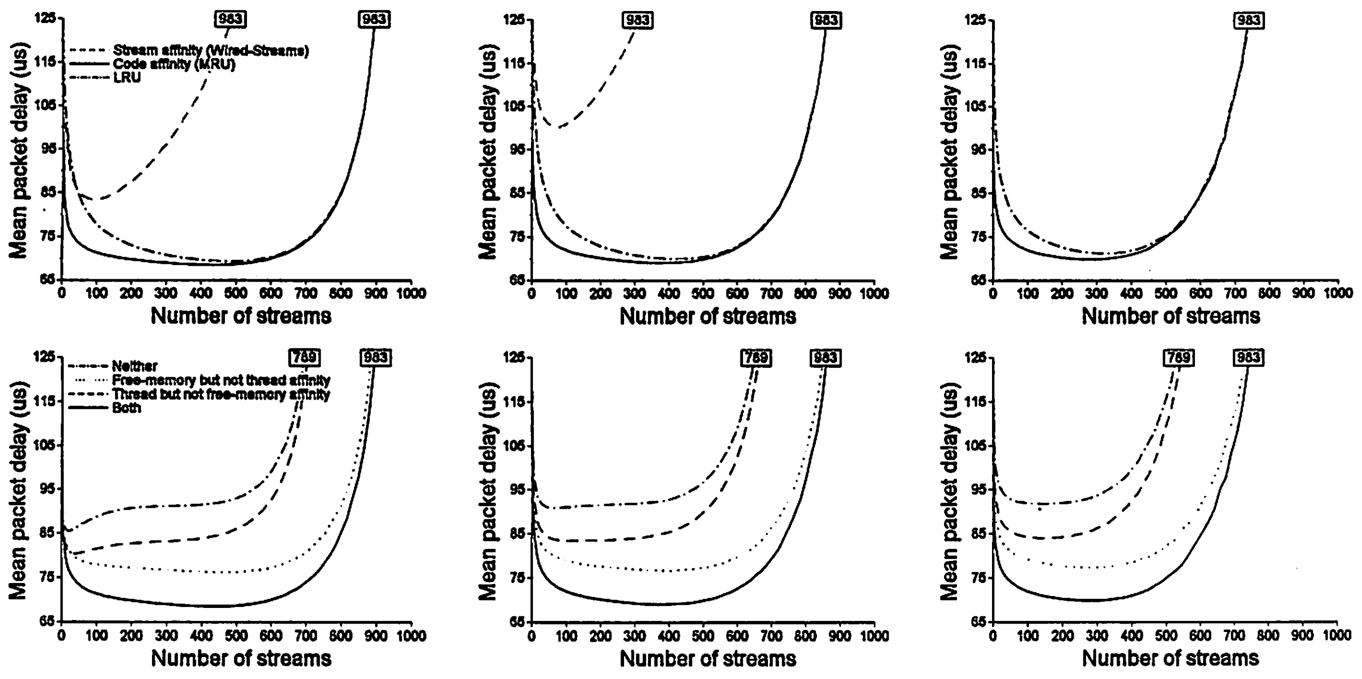
Figure 12 demonstrates that the benefit of affinity-based scheduling of send-side processing is maintained across increases in stream burstiness. Note that although the delay under stream affinity scheduling rises dramatically—in fact, the curve is off the scale in Figure 12c—it is of interest merely for informal comparison with the receive-side results. Because no protocol-specific or session-specific state is written on the send-side, none of the affinity asymptotes change as $\bar{B}$ varies.

On the send-side, it is reasonable to consider a smaller value of the mean inter-packet delay $\bar{I}$, in the sense that packets may be offered relatively quickly by an application. To address this consideration, we examined the performance for $\bar{I} = 8$ across values of $\bar{B}$. The results were as expected: delay under stream affinity scheduling rises dramatically (since arriving packets incur increased queueing), but there was little change in the performance of the other scheduling policies.

8 IPS Scaling

Thus far we have essentially considered *processor-scaled* IPS, in which the number of independent stacks is set according to the number of processors. What is the performance of *connection-scaled* IPS, in which the number of independent stacks scales (i.e., increases) with the number of connections[20]? In this section, we explore the pros and cons of connection-scaled IPS.

[20] Here we switch from "streams" to "connections" as denoting a logical association between communicating endpoints, to avoid confusion with the "stream scaling" terminology introduced in Section 4.



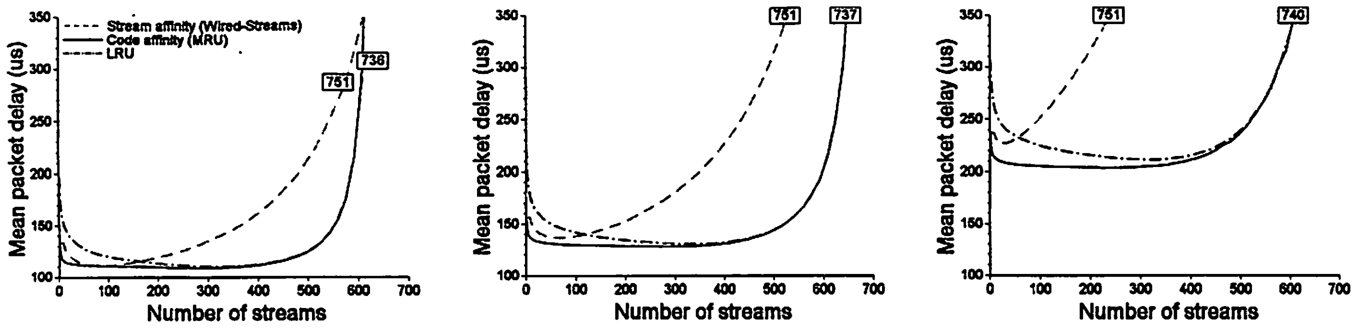
(a) $\bar{B} = 1$ (b) $\bar{B} = 2$ (c) $\bar{B} = 8$

Figure 12: Impact of stream burstiness on affinity scheduling for send-side processing

Recall from Section 4 that stream scaling increases packet latency by about 20% under IPS, since a high number of streams per independent stack results in infrequent realization of the fast-path demultiplexing optimizations. By maintaining a low number of streams per independent stack, connection-scaled IPS should lower per-packet latency. In addition, connection-scaled IPS should increase the enabled concurrency in the system. The fundamental concurrency limitation under IPS is sequential processing on individual stacks; increasing the number of stacks thus raises the level of concurrency, which may dramatically decrease packet queueing delay.

One problem is that scaling the number of IPS's beyond the number of processors complicates the packet queueing mechanism. Consider FIFO packet queueing as the baseline implementation. When a packet arrives, its IPS is determined and stored with the packet, which is then queued; packets are dequeued in FIFO order. The per-IPS concurrency constraint, however, complicates the determination of which packets are actually available for processing at any given time. To select a packet, a processor must search down the queue until it finds a packet from an IPS which is not currently executing. This potentially lengthy search complicates the otherwise simple packet dequeueing operation.

We can avoid the search (and approximate FIFO packet scheduling) by first queueing packets in per-IPS queues, and then queueing IPS's and serving *them* in FIFO order. Consider code affinity scheduling, with a global queue of IPS's which have waiting packets.



(a) $\bar{B} = 1$ (b) $\bar{B} = 2$ (c) $\bar{B} = 8$

Figure 13: Affinity scheduling under connection-scaled IPS

- When a packet arrives, it is added to the appropriate IPS packet queue. If this IPS was previously idle (i.e., had no packets waiting or executing), it is moved to the global IPS queue. If any processor is available, a thread is scheduled; else, the IPS waits.

- When a protocol thread finishes with a packet, it selects the next packet (if any) from the current IPS's packet queue; otherwise, the thread dequeues the next IPS (if any) from the global queue; otherwise, the processor is released.

For stream affinity scheduling, there are per-processor IPS queues. When an IPS becomes busy, it is added to the appropriate per-processor queue, and a thread is scheduled if that processor is not already executing protocol code. When dequeueing a new IPS, the thread consults only the per-processor IPS queue before releasing the processor.

Of course, the protocol thread need not completely empty an IPS's packet queue before moving on to the next IPS. To avoid fairness problems (e.g., a very bursty set of streams might starve streams mapped to other queued IPS's), a limit can be imposed on the number of packets which may be processed sequentially from a single IPS.

The following points bear emphasis.

- The algorithm is "semi-work conserving" in the sense that it allows work-conserving execution of IPS's, but not of packets on an individual IPS. As the number of independent stacks increases, the algorithm becomes less non-work-conserving, and packet queueing behavior under IPS approaches the behavior under Locking.

- The algorithm supports any number of independent stacks. Namely, it is backwards-compatible with processor-scaled IPS. In that case, IPS's are simply never queued, and the newer algorithm reduces to the earlier one.

We implemented this algorithm in the multiprocessor simulation model. Figure 13 shows the performance across values of $\bar{B}$, when the number of independent stacks is matched to the number of admitted streams (i.e., under connection-scaled IPS). For a comparison with processor-scaled IPS, compare Figures 13 and 10. Note that the per-packet latency is reduced by about $20\mu s$: the stream affinity curve in Figure 13a drops to about $110\mu s$, whereas its minimum was about $130\mu s$ under processor-scaled IPS (Figure 10a). As a result, the maximum number of supportable stream increases to 751 (from 629 in Figure 10a).

Significantly, code affinity scheduling now outperforms stream affinity scheduling across the broad range of arrival rates. Not only is the code affinity curve in Figure 13 below the stream affinity curve, but it remains low for a larger range in the number of streams. Further, the difference between the policies magnifies as $\bar{B}$ increases. These phenomena directly reflect the increased concurrency available under connection-scaled IPS. Yet since stream affinity scheduling now provides lower packet execution time (since code affinity scheduling no longer "wires" IPS's to processors by default, as under processor-scaled IPS), it supports a higher number of streams—751 versus 736 for code affinity scheduling in Figure 13.

Finally, although code affinity scheduling is semi-work conserving under connection-scaled IPS, it is not fully work conserving (as under Locking). Thus as $\bar{B}$ varies, the minimum delay under code affinity scheduling increases from $115\mu s$ to $130\mu s$ to $210\mu s$ in Figure 13, whereas it remains relatively stable across values of $\bar{B}$ under Locking (Figure 9).

9 Summary

In this paper, we have presented evidence that strengthens the argument for affinity-based scheduling in parallel networking.

- We have established the applicability of affinity-based scheduling to parallelized send-side UDP/IP/FDDI protocol processing, and evaluated its performance. We find the technique performs well, in agreement with our earlier receive-side results.
- We have examined the impact of stream burstiness and source locality as captured by the Packet-Train source model [16]. We find that the performance of the affinity-based policies is relatively insensitive to these source characteristics.
- We have experimentally measured the impact of a copying uncached packet data on protocol execution time, and incorporated the overhead into our multiprocessor simulation model with the aid of a published UDP/IP/FDDI packet-size distribution. We find the benefit of affinity-based scheduling remains significant.

In addition, we have explored connection-scaled IPS, which scales the number of independent stacks with the number of admitted streams. We find that the performance of IPS improves dramatically, due to lower packet processing times and improved packet queueing behavior.

One compelling avenue of future work is to examine affinity-based scheduling of TCP/IP processing, as TCP accounts for a large fraction of wide-area network traffic. Although TCP is a far more complex protocol than UDP, our results are likely to hold directly for TCP, for two reasons. First, the breakdowns of overall processing time overheads for TCP and UDP packets are very similar (compare for example graphs 3a and 3b in [18]). Second, as pointed out in [18], at its most influential (i.e., for 1-byte packets), TCP-specific processing only accounts for around 15% of overall packet execution time (for the DEC Ultrix 4.2a TCP/IP/FDDI in-kernel protocol stack studied in that work).

Our work has shown that affinity-based scheduling can significantly lower per-packet latency through the network subsystem, enabling the host to provide higher protocol bandwidth to individual streams and to increase the maximum number of streams that can be concurrently supported. These benefits are increasingly important as high-speed networks and large-scale multiprocessor servers move into the mainstream. Given recent processor and memory speed growth trends, it is likely that affinity-based scheduling techniques will continue to provide significant performance gain.

Acknowledgments

Thanks to Erich Nahum and David Yates for the uniprocessor port of the *x*-kernel (version 3.2) to the IRIX platform, which was the starting point of the experimental component of this research.

References

[1] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 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] Mats Björkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications*, pages 74–83, San Francisco, CA, September 1993.

[4] Torsten Braun and Claudia Schmidt. Implementation of a parallel transport subsystem on a multiprocessor architecture. In *Second International Symposium on High-Performance Distributed Computing*, Spokane, Washington, July 1993.

[5] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[6] Peter B. Danzig, Sugih Jamin, Ramón Cáceres, Danny J. Mitzel, and Deborah Estrin. An empirical workload model for driving wide-area TCP/IP network simulations. *Journal of Internetworking: Research and Experience*, 3(1):1–26, 1992.

[7] Murthy Devarakonda and Arup Mukherjee. Issues in implementation of cache-affinity scheduling. In *Proceedings of the Winter 1992 USENIX Conference*, pages 345–357, San Francicso, CA, January 1992.

[8] Arun Garg. Parallel STREAMS: A multi-processor implementation. In *Proceedings of the Winter 1990 USENIX Conference*, pages 163–176, Washington, D.C., January 1990.

[9] 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.

[10] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, May 1991.

[11] Riccardo Gusella. A measurement study of diskless workstation traffic on an Ethernet. *IEEE Transactions on Communications*, 38(9):1557–1568, September 1990.

[12] Ian Heavens. Experiences in parallelisation of streams-based communications drivers. *OpenForum Conference on Distributed Systems*, November 1992.

[13] 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.

[14] Mabo Ito, Len Takeuchi, and Gerald Neufeld. A multiprocessing approach for meeting the processing requirements for OSI. *IEEE Journal on Selected Areas in Communications*, 11(2):220–227, February 1993.

[15] Niraj Jain, Mischa Schwartz, and Theordore R. Bashkow. Transport protocol processing at Gbps rates. In *Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications*, pages 188–199, Philadelphia, PA, September 1990. ACM.

[16] Raj Jain and Shawn Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):986–995, September 1986.

[17] Matthias Kaiserswerth. The parallel protocol engine. *IEEE/ACM Transactions on Networking*, 1(6):650–663, December 1993.

[18] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications*, pages 259–268, San Francisco, CA, September 1993.

[19] Jonathan Kay and Joseph Pasquale. Measurement, analysis, and improvement of UDP/IP throughput for the DECstation 5000. In *Proceedings of the Winter 1993 USENIX Conference*, pages 249–258, January 1993.

[20] Thomas F. LaPorta and Mischa Schwartz. Performance analysis of MSP: Feature-rich high-speed transport protocol. *IEEE/ACM Transactions on Networking*, 1(6):740–753, December 1993.

[21] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[22] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[23] Jeffrey C. Mogul. Network locality at the scale of processes. In *Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications*, pages 273–284, Zürich, Switzerland, September 1991.

[24] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 125–137, Monterey, CA, November 1994.

[25] William Nowicki. Personal communication, April 1995. Silicon Graphics, Inc.

[26] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[27] Vern Paxson. Empirically-derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, August 1994.

[28] Vern Paxson and Sally Floyd. Wide area traffic: The failure of Poisson modeling. In *Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications*, pages 257–268, London, England UK, August 1994.

[29] David Presotto. Multiprocessor STREAMS for Plan 9. In *United Kingdom UNIX User's Group*, January 1993.

[30] Erich Rütsche and Mattias Kaiserswerth. TCP/IP on the parallel protocol engine. *Fourth IFIP TC6.1/WG6.4 International Conference on High Performance Networking*, pages 119–134, December 1992.

[31] James Salehi, James Kurose, and Don Towsley. Scheduling for cache affinity in parallelized communication protocols. Technical Report UM-CS-1994-075, University of Massachusetts, Amherst, October 1994. Available via FTP from `gaia.cs.umass.edu` in `pub/Sale94:Scheduling.ps.Z`.

[32] James Salehi, James Kurose, and Don Towsley. The performance impact of scheduling for cache affinity in parallel network processing. In *Fourth IEEE International Symposium on High-Performance Distributed Computing*, Pentagon City, VA, August 1995. Available via FTP from `gaia.cs.umass.edu` in `pub/Sale95:Performance.ps.Z`.

[33] James Salehi, James Kurose, and Don Towsley. Scheduling for cache affinity in parallelized communication protocols (extended abstract). In *Proceedings of 1995 SIGMETRICS/Performance International Conference on Measurement and Modeling of Computer Systems;* Ottawa, Canada, May 1995. Available via FTP from `gaia.cs.umass.edu` in `pub/Sale95:Scheduling.ps.Z`.

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

[35] Douglas C. Schmidt and Tatsuya Suda. Measuring the impact of alternative parallel process architectures on communication subsystem performance. In *Proceedings of the* 4^{th} *International Workshop on Protocols for High-Speed Networks*, Vancouver, British Columbia, August 1994. IFIP.

[36] Douglas C. Schmidt and Tatsuya 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, April 1995. IEEE.

[37] Jaswinder Pal Singh, Harold S. Stone, and Dominique F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, July 1992.

[38] Mark S. Squillante and Edward D. Lazowska. Using processor cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[39] Dominique F. Thiebaut and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[40] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 26–40, Pacific Grove, CA, October 1991. ACM.

[41] C. Murray Woodside and R. Greg Franks. Alternative software architectures for parallel protocol execution with synchronous IPC. *IEEE/ACM Transactions on Networking*, 1(2):178–186, April 1993.