Scheduling for Cache Affinity in Parallelized Communication Protocols

James D. SALEHI, James F. KUROSE
and Don TOWSLEY

CMPSCI Technical Report 94-75

October, 1994

Scheduling for cache affinity in parallelized communication protocols*

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

October 15, 1994

Abstract

In this paper, we explore the benefits of processor cache affinity scheduling of parallelized network protocol processing. We find that affinity scheduling, which has not previously been shown to be of significant benefit to common applications, can provide large performance gain in the context of parallelized protocol processing.

We conduct a set of multiprocessor experiments designed to measure packet processing time in a UDP/IP/FDDI protocol stack in the *x*-kernel on an SGI Challenge XL multiprocessor. These measurements are then used to parameterize a combined simulation/analytic model of multiprocessor protocol processing. Our simulation results show that affinity scheduling can significantly reduce message delay associated with protocol processing, allowing a host to support a greater number of concurrent streams, to provide a higher maximum throughput to individual streams, and to decrease the end-to-end latency seen by an application. We find the reduction in end-to-end latency can be significant even when the fixed-overhead components of end-to-end delay are large with respect to message processing time. In addition, we compare two implementation approaches to enabling protocol parallelism, and find that the approach which maximizes cache affinity delivers much lower message latency, and enables much higher message throughput capacity. Yet this approach exhibits limited intra-stream scalability and poor response to intra-stream burstiness. Finally, we show that a "hybrid" approach performs best for a specific class of traffic streams—yielding high message throughput, high intra-stream scalability, and robustness in the presence of bursty arrivals.

*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

In many modern computer architectures, there is a significant difference in the amount of time needed to reference a memory location cached locally versus one held in main memory. For example, on the SGI Challenge XL multiprocessor (the experimental platform used in this paper) a reference can be serviced by the first-level cache in one processor cycle, whereas the fastest main memory access over the shared bus exceeds 100 cycles[1]. In multiprocessor architectures, these 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.

In this paper, we evaluate several different affinity-based scheduling policies for parallelized protocol processing, an application which has recently generated considerable interest [4, 6, 8, 10, 13, 16, 17, 18, 19]. We consider protocol parallelization paradigms in which each message, during the course of its processing, visits a single processor and executes within the context of a single thread[2]. This captures the parallelization found in several multiprocessor protocol implementations, including parallelizations of the x-kernel [2, 13], the STREAMS implementation in Plan 9 [17], and the ASX framework [19]. A related form of parallelism is found in the STREAMS implementations in many commercial operating systems [18, 4, 6].

We present two sets of results. First, we show that affinity scheduling can significantly reduce message delay associated with protocol processing, enabling the host to support a greater number of concurrent streams, to provide a higher maximum throughput to individual streams, and to decrease the end-to-end latency seen by an application. We find the reduction in end-to-end latency can be significant even when the fixed-overhead components of end-to-end delay are large with respect to message processing time.

Second, we compare implementations of two approaches to parallelizing protocol processing, and find that the approach which maximizes cache affinity delivers much lower message delay and much higher message throughput capacity. However, this approach exhibits limited intra-stream scalability and poor response to intra-stream burstiness. We thus propose a "hybrid" approach for a specific class of streams, and show that it offers the best overall performance—yielding high message throughput, high intra-stream scalability, and robustness in the presence of bursty arrivals.

We establish our results using experimental measurements in conjunction with simulation and analytic techniques. Specifically, we measure message processing times of a UDP/IP/FDDI protocol stack running in a controlled multiprocessor environment and under specific conditions of cache affinity. These measurements are then used to parameterize the analytic component of a simulation model of multiprocessor protocol processing, under various

[1] As measured on our platform.

[2] We use the terms *thread* and *process* interchangeably.

affinity-based scheduling policies. The hardware platform serving as the basis for the experimental component of our study is an 8-processor SGI Challenge XL running the IRIX 5.2 operating system. Protocols are implemented using the *x*-kernel framework in user space [8, 14].

Previous work on affinity-based scheduling [3, 5, 22, 26] has not established a consensus on its efficacy, even reaching seemingly conflicting conclusions (e.g., [26] vs. [22]). None of the work has found affinity-based scheduling techniques to be of significant benefit to common applications. For affinity scheduling to be effective, the enabled processing speedup must be large in comparison to the task's per-visit processor execution time. We show that multiprocessor protocol processing satisfies this criterion.

Several aspects of our methodology may facilitate research on affinity scheduling in other application domains. First, we demonstrate a systematic decomposition of the task's memory reference stream into disjoint classes, for which distinct scheduling policies for specific resources can be identified. This allows us to evaluate the marginal contributions of the individual policies, and assess their relative performance gains. Second, we present extensions to an existing analytic model of the execution time of an affinity-scheduled task in a multitasking environment [25, 22]. The primary contribution here is to relax the requirement of having identified the task's *footprint*—the set of cache lines currently referenced by the executing task. In practice, it can be hard to determine a task's footprint, especially for large, multithreaded, multiprocessor applications. We show how to parameterize the analytic model with experimental timing measurements, which are much easier to obtain. Finally, in the design of the experiments formulated to yield these measurements, we illustrate an experimental method for isolating the individual components of affinity-based overhead.

This paper is organized as follows. Section 2 presents the problem formulation. In section 3 we outline the simulation model of multiprocessor protocol processing. Section 4 presents the analytic model of message execution time, and section 5 discusses the implementation-based experiments performed to measure the parameters needed by the analytic model. Performance results are presented in section 6. We discuss related work in section 7, and summarize our work in section 8.

2 Problem formulation

2.1 Protocol processing

Let us begin by considering a simplified view of in-kernel, receive-side protocol processing. Figure 1 depicts a multiprocessor with an FDDI network connection. Processors are labeled P_i, kernel-level protocol threads T_i, and application threads A_i. In the figure, P_3 and P_4 are running protocol threads (i.e., threads executing a communication protocol or protocol stack), and P_1 and P_2 are running non-protocol threads. For simplicity, we will assume that processors are kept busy running non-protocol tasks when not executing protocol code, that protocol processing

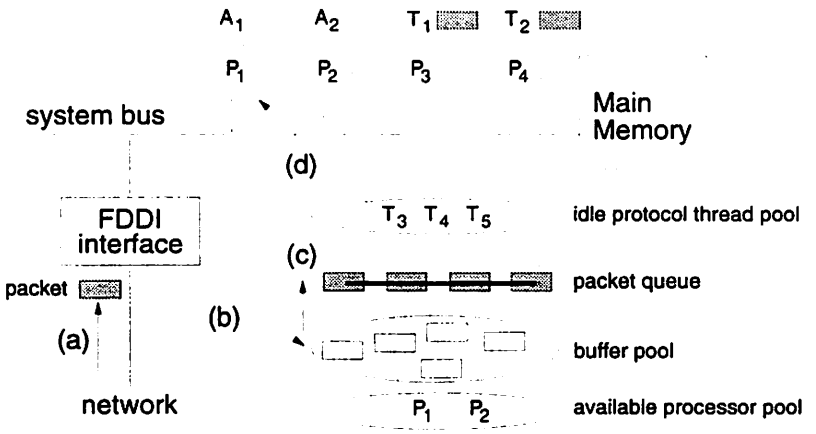


Figure 1: Conceptual model of multiprocessor protocol processing

receives priority over non-protocol processing, and that when unblocked, a protocol thread is scheduled immediately. Figure 1 also shows a pool of "available" processors (i.e., processors not performing protocol processing) and a pool of idle protocol threads.

Now consider the actions taken upon arrival of a packet, generating a hardware interrupt at the FDDI interface (a). The interrupt handler allocates a buffer from the buffer pool, and initiates DMA (b). A second interrupt is generated when DMA is complete, and the buffer is added to the queue of packets awaiting protocol processing (c). At this point, a scheduling decision must be made: either no additional protocol processing is initiated, or an available processor and a thread are selected and protocol processing subsequently begins. (We will discuss the specific protocol schedulability criterion in section 3). In the example in Figure 1, protocol thread T_3 is scheduled to run on P_1 (d).

The activity of the protocol thread is shown in Figure 2, which highlights certain aspects of the UDP/IP/FDDI receive-side path through the *x*-kernel (version 3.2). This *x*-kernel implementation runs as user-space application above the native IRIX 5.2 operating system. A simulated device driver emulates the protocol functionality associated with managing an FDDI network interface attachment. The diagram does not reflect support for protocol concurrency—it depicts an unparallelized *x*-kernel.

During protocol processing, the packet visits protocol modules (i.e., accesses and executes protocol code) and references stream-specific data structures known as *session objects*. In the FDDI protocol layer, a lookup in the FDDI *active map* (a data structure which records the protocol's active sessions) enables demultiplexing of the packet to the correct FDDI session. There, a session reference counter is incremented to register outstanding packet processing on the session. The packet is then pushed up to VNET, a "virtual" protocol for managing multiple link-layer protocols (such as FDDI). In the case of receive-side processing, VNET acts as a "pass-through"; the packet flows directly to the VNET session and then up through IP and UDP. At the top of the stack, packet data is made available to the

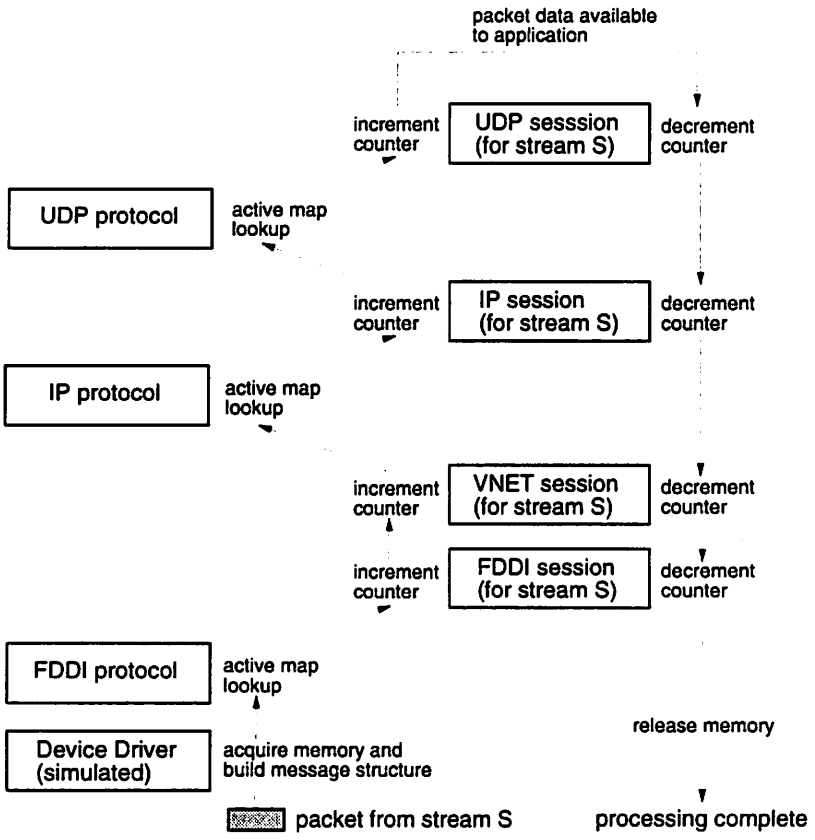


Figure 2: x-kernel UDP/IP/FDDI receive-side protocol processing

application. The protocol thread then unwinds its runtime stack, decrementing the UDP, IP, VNET and FDDI session reference counters along the way.

Each protocol active map is implemented as a hash table indexed by a key composed primarily of specific fields from the packet header. To speed up demultiplexing in the anticipated case of subsequent arrivals which visit the same session, each map maintains a history of the most recent map lookup. For a subsequent packet from the same stream, the actual lookup is avoided. For a packet from another stream, a full lookup is performed, and a new history value is written.

In the device driver, the protocol thread first builds an x-kernel "message structure" around the packet buffer written by the device interface (Figure 2). A portion of this structure is dynamically allocated; this memory is subsequently released when the packet's processing is complete. We assume the protocol subsystem maintains its own free-memory pool (implemented as an array of pointers to segments of available memory, with access serialized by a software lock), so that this activity[3] can be affinity scheduled.

[3] Specifically, to acquire memory, the protocol thread acquires the pool's synchronization lock, increments the array index, and returns a pointer to the available memory. This memory is subsequently written by the protocol thread. To release memory, the thread acquires the lock, decrements the index, and stores the pointer to the memory being released.

2.2 Parallelization alternatives

To modify the protocol stack of Figure 2 to support concurrency, accesses to the protocol active maps during demultiplexing and to the session reference counters during increment or decrement operations must be protected. Consider the following two parallelization approaches.

- **Locking.** One approach is to add a software lock to each protocol active map and session object, and to perform each access under the protection of the appropriate lock. In our implementation of Locking, we use non-blocking IRIX spin locks. Since these user-space locks do not incur the overhead of crossing the user/kernel boundary, they are light-weight. A single lock can be acquired and released in 0.7-0.8μs—when the underlying data structures are cache-resident. But when they must migrate (which can occur frequently when multiple processors access an individual shared item), this overhead increases by a nearly a factor of three, to around 2.5μs. The inherent computing demand of about 50μs to process a packet (section 5) implies that each uncached lock adds about 5% overhead to the packet receive time. This suggests that a parallelization approach which avoids software locks may yield higher performance.

- **Independent Protocol Stacks (IPS).** A second approach to parallelism is to implement multiple independent protocol stacks. Each individual stack supports no protocol concurrency, and therefore does not require software locks. The key idea is to identify the non-read-only protocol data structures in the unparallelized stack, and to create N copies of each. Typically, N would equal the number of processors, although this is not a requirement. In Figure 2, 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. A single software lock is added to each stack to ensure serialization of its processing. Under IPS, session objects need not be duplicated: an individual stream is associated with one of the N independent stacks.

  The advantage of IPS is that it avoids the cache-related overheads associated with software locks. The disadvantage is that it does not allow concurrent processing of packets from the same stream, or from streams mapped to the same individual protocol stack. This raises the possibility of higher intra-stream delay. We will investigate these tradeoffs in section 6.

We have implemented both Locking and IPS through modifications to the original multi-threaded, uniprocessor x-kernel (Figure 2). These implementations form the basis of our experimental measurements in section 5.

2.3 Affinity scheduling

To clarify the notion of affinity scheduling in this context, consider the memory reference stream issued during message processing. This stream consists of a mix of reads and writes to individual cache lines. Our approach is

to partition the stream into disjoint classes, and within each class to propose policies or organizations designed to expose the endpoints of performance with respect to affinity scheduling.

Consider references to protocol code and read-only data, which for simplicity we collectively label "code". Achieving *code affinity* (i.e., avoiding cache misses on code references) is a processor scheduling issue[4] since the cache coherence protocol allows read-only references to be replicated among processor caches. Intuitively, scheduling protocol processing where it more recently executed should result in higher code affinity. Thus, we consider MRU and LRU management of the available processor pool. That is, when selecting a processor, the interrupt handler either selects the processor that most-recently or least-recently executed protocol code.

Code affinity does not encompass initial references by the protocol thread to the packet itself, even though these references are read-only. Since the packet was DMA'd by the network interface to main memory (Figure 1), it cannot be cache resident. Therefore we do not consider affinity scheduling of the references to the packet itself. (Furthermore, we design our experiments in section 5 to ensure these references result in cache misses.)

For each write reference, the cache coherence protocol requires the processor to first gain exclusive ownership of the underlying cache line. This operation invalidates copies of the line in all other processors' caches. Thus each write reference results in a cache miss when the line was most recently written by some other processor. We achieve affinity for write references by ensuring that it is the same processor performing the write each time the line is written.

The protocol thread writes into its stack area for each packet received. These write references generate cache misses whenever the thread executes at a new processor. Achieving *thread stack affinity* (i.e., avoiding coherence-based misses on initial writes to the thread stack area) is a thread scheduling issue[5] and is reflected in the organization of the idle protocol thread pool. We consider global versus per-processor organizations. Under LRU or MRU management of a global pool, threads tend to migrate among processors, resulting in coherence-based stack misses. When thread pools are organized on a per-processor basis, these misses are eliminated.

The protocol thread performs writes when it acquires or releases memory. Whether we achieve *free-memory affinity* (i.e., avoid coherence-based misses associated with these writes) depends on the organization of the free-memory pool. We consider global versus per-processor organizations. When a global pool is managed LRU, the overhead of migrating the underlying cache lines *may* be incurred, depending on which processor last accessed the pool. Under a per-processor organization, these misses are eliminated.

Finally, the protocol thread writes stream-specific data—the four session reference counters, their locks under Locking, and one element of IP state—for each packet received. We can achieve *stream affinity* by "wiring" streams to processors. This in turn is a processor scheduling decision. Under "Wired-Streams" processor scheduling, the

[4]That is, relates to the choice of which processor to select to run a protocol thread when one is unblocked (see (d) in Figure 1).
[5]That is, relates to the choice of which protocol thread to unblock when one is unblocked.

Resource managed	*Objective*	*Policy*
Processors	schedule for code affinity schedule for stream affinity baseline	MRU Wired-Streams LRU
Threads	schedule for stack affinity baseline	per-processor pools global pool
Free-memory	schedule for free-memory affinity baseline	per-processor pools global pool

Table 1: Scheduling protocol processing for cache affinity.

processor selected when scheduling a protocol thread is determined by the stream identifier of the packet about to be processed. Under IPS but not Locking, Wired-Streams scheduling results in incidental affinity scheduling of the protocol active map history writes and the write to the IPS concurrency lock. (Under Locking we do not consider affinity scheduling of the history writes, nor of the writes performed in acquiring and releasing the active map locks). This suggests that Wired-Streams scheduling may have a greater impact under IPS than under Locking.

We have now considered affinity scheduling of every reference in the memory reference stream issued during message processing (with the two exceptions, as noted, of the initial references to the packet data, and the writes associated with the active maps under Locking). Table 1 summarizes the resources which can be managed to achieve affinity scheduling of the protocol reference stream, the scheduling objectives, and the scheduling policy considered.

3 Simulation model of multiprocessor protocol processing

We evaluate the benefits of affinity scheduling through a multiprocessor simulation model that closely follows the behavior in Figure 1. Consider first the simulation of Locking, in which there are N processors and N protocol threads.

The packet arrival process is batched Poisson[6]. We consider both deterministic and geometric batch size distributions. Upon an arrival, the batch size is computed, a stream identifier is assigned (by sampling a uniform distribution) and the arriving packet(s) are queued for protocol processing.

The protocol processing scheduling criterion is as follows. The simulation attempts to immediately schedule a processor P_i for the first packet of the batch. Under MRU or LRU processor scheduling, if the available processor list is non-empty, a processor is selected. Under Wired-Streams processor scheduling, a lookup based on the packet's stream identifier indicates the destination processor, which is selected if it is available. Otherwise the packet waits.

Before a packet can go into service, a protocol thread must be selected. When scheduling for thread stack affinity, the simulation maintains per-processor pools of available threads, and a thread is dequeued from the appropriate

[6] Batched arrivals are intended to model communication activity such as fragmented UDP/IP/FDDI, or multi-packet RPC [12, 9].

pool. Otherwise, the simulation maintains a global pool (implemented as a list), which is managed either MRU or LRU to encourage thread stack migration.

The packet runtime (i.e., the amount of time taken to process the packet up through the layers of the protocol stack) is then computed. The computation depends on whether the thread stack migrates, whether the written stream-specific cache lines migrate, whether the protocol active map history locations are overwritten (and if so whether they migrate), whether the lines written in accessing the free-memory pool must migrate, and finally, the length of time since the processor last executed protocol processing. Details of this computation are provided in section 4.

The packet enters service with its computed runtime. Upon packet completion, whether the protocol thread continues with another packet depends jointly on the processor scheduling policy and the state of the packet queue. Under MRU and LRU processor scheduling, if any packet waits, the protocol thread continues. Under Wired-Streams, the thread continues if any packet from the same stream is waiting. When the thread continues, a packet is dequeued FIFO among eligible packets, and its runtime computed as above. For all policies, when no packet can be selected by the thread, the processor is released to non-protocol processing. Note that protocol processing is work-conserving under MRU and LRU processor scheduling, but not under Wired-Streams.

The simulation of IPS is similar to that of Locking, with the following three distinctions. First, after the stream identifier is assigned to an incoming arrival, the identifier of the destination protocol stack is retrieved by lookup. Before selecting a processor, the simulation checks whether this stack is busy; if so, the packet waits. Second, the runtime computation involves a set of IPS-specific experimental measurements. Third, when a packet completes processing at a given protocol stack, if any packets wait for the same stack, one is selected FIFO; else, the processor is released. Note that protocol processing is non-work-conserving under all processor scheduling policies.

We now turn to the computation of the runtime of an individual packet.

4 Analytic model of packet execution time

Consider Figure 3, which presents a simplified view of the memory architecture of our SGI Challenge. Each 100MHz, R4400 processor has separate on-chip 16KB instruction and data caches (L1), each with a 32-byte line size. Each processor also has a private 1MB unified second-level cache (L2), with a 128-byte line size. All caches are direct mapped. When a memory reference is issued, first L1 is checked (a); on a miss, L2 is checked (b). A miss in L2 generates a request sent over the shared bus (c), which is satisfied by the owner of the cache line—either another processor's cache, or main memory.

In this section, we incrementally develop a model of the execution time of protocol processing on this architecture. The initial model reflects the impact of non-protocol processing on packet execution time, assuming a single-level processor cache, and excluding potential migration overheads of caches lines written during protocol processing.

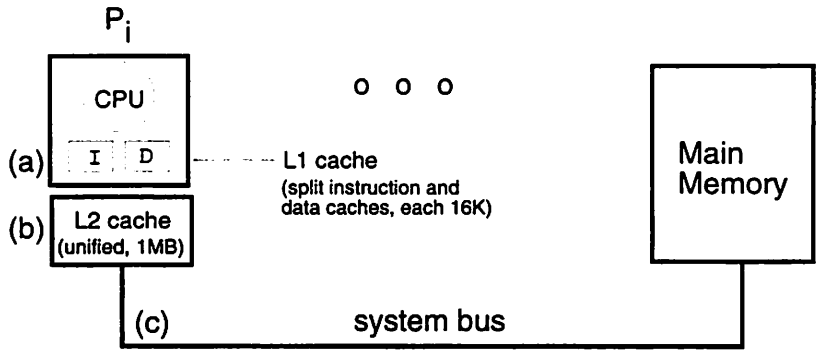


Figure 3: A simplified view of the memory hierarchy of the SGI Challenge.

Next, we extend the model to reflect a second-level processor cache, with distinct size, organization, and miss service time. Finally, we incorporate the migration overheads.

4.1 The initial model

We initially assume each processor has a single-level direct-mapped cache, organized into S lines each of size L bytes.[7] Whenever a processor is not executing protocol code, it runs some other workload. This non-protocol code displaces a fraction of the processor cache contents. When the protocol code revisits P_i after time x_i, its execution time t_i depends on (and is roughly an increasing function of) x_i. The displaced protocol footprint must be reloaded into the cache, and in general a larger fraction of the footprint is displaced when a greater length of time has passed since the most recent execution of a protocol thread at the processor.

The initial model depends on two experimentally-measured parameters. t_{cold} is the time required to process a packet when the protocol footprint is entirely flushed from the cache (i.e., the cache is "cold"). This is the maximum execution time. t_{hot} is the time required to process a packet immediately following the completion of a previous packet (i.e., in the absence of intervening processing). This is the minimum execution time.

We formulate t_i as a monotonically increasing function of x_i, bounded below by t_{hot} and above by t_{cold}. To do so, we rely on the work of Singh, Stone and Thiebaut [20]. Let m denote the average memory reference rate of the intervening processing. We assume m is stationary. $R(x_i) = x_i/m$ is the average number of references issued during x_i. Let $u(R(x_i), L)$ denote the *footprint* function, defined as the number of unique memory lines referenced at the processor in R references, for a cache line size L bytes. In [20], the authors show that this function is closely modeled by an expression of the form

$$u(R(x_i), L) = WL^a R^b d^{\log L \log R(x_i)} \quad (1)$$

[7] The model can be easily extended to a set-associative cache.

where the constants W, a, b, and d relate to working set size, spatial locality, temporal locality, and interactions between spatial and temporal locality, respectively, of the intervening processing. (The fact that $u(R(x_i), L)$ is a power function of $R(x_i)$ for fixed L was observed independently by Thiebaut [23, 24], and Kobayashi and MacDougall [11].)

In [20], the authors show equation (1) to be consistent with data given by Smith [21], and Agarwal, Horowitz and Hennessy [1]. They also demonstrate its accuracy through detailed validation on segments of a 200-million-reference trace of a multiprogrammed IBM/370 MVS workload, consisting of a representative workload of user applications and operating system activity. We use the specific parameters derived by the authors for this workload ($W = 2.19827, a = 0.033233, b = 0.827457$ and $\log d = -0.13025$) to model the non-protocol activity in our system.

Equation (1) enables us to accurately capture the behavior of the intervening processing, with the exception of a small x_i. As Singh et. al. point out, inaccuracy in this region is most likely due to the fact that most tasks generate a higher number of misses when initially loaded. $u(R(x_i), L)$ grows at rate faster than (1) initially, then follows (1) when the entire working set has been referenced. Thus for small x_i, we underestimate the number of lines displaced by the intervening processing, and therefore underestimate the benefit of affinity scheduling.

Let $F(x_i)$ denote the fraction of the cache which has been flushed by these $u(R(x_i), L)$ references. We compute $F(x_i)$ by assuming the references map independently into cache sets (an assumption also made in [22] and [25] in similar context). Let the random variable X denote the number of the $u(R(x_i), L)$ references that map to a randomly chosen set. X has binomial distribution with parameters $n = u(R(x_i), L)$ and $p = \frac{1}{S}$. Since we are modeling a direct-mapped cache (i.e., with set-associativity 1), we have $P(X = 0) = (1 - \frac{1}{S})^{u(R(x_i),L)}$, and therefore $F(x_i) = 1 - P(X = 0)$.

Figure 4 shows $F(x_i)$ for the L1 and L2 sizes and organizations of our SGI Challenge. $F(x_i)$ has been computed for the 100-Mz clock rate of MIPS R4400, assuming an average of 5 clock cycles per memory reference ($m = 5$). Note that the protocol footprint is flushed much more slowly from L2 than from L1, reflecting its much larger size.

Finally, we predict the protocol execution time as a linear interpolation[8] of (t_{hot}, t_{cold}) based on F:

$$t_i = (1 - F(x_i))t_{hot} + F(x_i)t_{cold}. \tag{2}$$

[8]Task execution time as the linear interpolation of the maximum "reload transient" is also the approach taken in [22]. 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 $D + RC$.

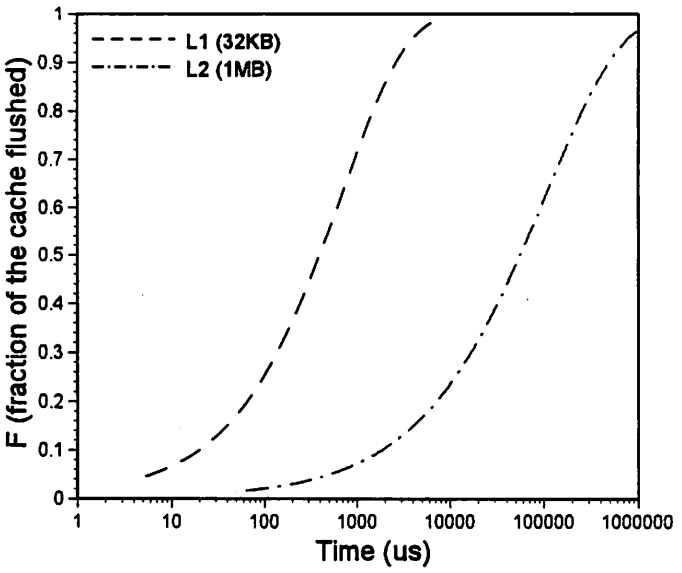


Figure 4: Impact of intervening processing on L_1 and L_2.

4.2 Extension to multilevel cache hierarchy

To extend the model to include a per-processor second-level cache, we expand the measured times to $t_{hot,\,hot}$ (execution time in the absence of intervening processing), $t_{cold,\,hot}$ (execution time when L1 is cold, and the footprint comes from L2), and $t_{cold,\,cold}$ (execution time when the protocol footprint is entirely flushed from both caches).

Although each processor on the Challenge has separate 16K instruction and data caches at L1, we make the simplifying assumption of a 32K *unified* cache at L1. This results in only a small change to $F(x_i)$ under the assumption that the reference stream is split approximately between the two caches. Experimental measurements support this assumption (see for example Table 1 of [7]).

We compute $u(R(x_i), L_1)$ and $u(R(x_i), L_2)$, where L_1 and L_2 are the line sizes of the L1 and L2 caches, respectively. From these, $F_1(x_i)$ and $F_2(x_i)$ (the fractions of L1 and L2 flushed) are computed. Finally, t_i is modeled as

$$t_i(x_i) = (1 - F_1(x_i))t_{hot,\,hot} + F_1(x_i)((1 - F_2(x_i))t_{cold,\,hot} + F_2(x_i)t_{cold,\,cold}). \quad (3)$$

As in equation (2), the F's are used to scale the extremes of protocol execution time. Each term represents the runtime when the entire protocol footprint is found at the corresponding level in the memory hierarchy, weighted by the fraction of the footprint found at that level. For example, the second term represents the runtime contribution made by the fraction $F_1(x_i)(1 - F_2(x_i))$ of the footprint retrieved from L2 at P_i.

4.3 Incorporating migration overheads

The final step is to incorporate potential migration overhead associated with the cache lines written during protocol processing, and to reflect the specific parallelization alternative.

The solution is to parameterize equation (3) with experimental measurements for $t_{hot,\ hot}$ and $t_{cold,\ hot}$ specific to (a) whether the parallelization approach is Locking or IPS, (b) whether the thread stack migrates, (c) whether session-specific lines migrate, and (d) whether protocol active map history locations are overwritten, and if so whether the underlying cache lines must migrate. Thus, there are 24 pairs of values for $t_{hot,\ hot}$ and $t_{cold,\ hot}$. In addition, there are two experimental values for $t_{cold,\ cold}$; since the entire footprint must migrate for $t_{cold,\ cold}$, it depends only on the parallelization alternative.

The overheads of migrating the lines associated with the free-memory pool (when modeling a global pool) are incorporated in a slightly different manner. The simulation tracks the most recent processor P_m to access the global pool. After equation (3) is used to compute the execution time of a packet at P_i, a check is made whether $P_i = P_m$; if not, the execution time is inflated by *overhead_acquire*, a measured value reflecting the overhead of acquiring free memory when the underlying cache lines must migrate. When protocol processing for the packet is complete (at which time the protocol thread releases memory, as in Figure 2), if some other processor has accessed the global pool in the interim, packet execution time is extended by *overhead_release*, another experimentally-measured value.

Section 5 discusses how these 28 measurements are obtained, and presents and discusses the experimental results.

4.4 Discussion

There are three points which warrant emphasis. First, our model does not capture the positive impact of affinity scheduling toward reducing bus utilization and memory contention (as is done, e.g., in [22]). To some degree the simulation therefore underestimates the benefits of affinity scheduling.

Second, while our model is similar to the model of task execution time developed in [22], there are several important distinctions. First, our approach does not require identifying the footprint of the task being affinity scheduled. In practice, it can be hard to acquire the memory reference trace, especially for large, multithreaded multiprocessor applications. Instead, our approach is based on direct timing measurements, which are much easier to obtain. Second, we have extended the model to address a multi-level cache hierarchy. Third, the simulation model dynamically instantiates the analytic model with timings which reflect the relevant migration overheads. We show in section 5 that these overheads account for a large variation in protocol execution time. We thus contend they should be considered explicitly in evaluating the benefits of affinity-based scheduling techniques.

Third, the model does not reflect the impact of contention for software locks, which would inflate the protocol

execution time (e.g. [18]). However, UDP/IP has been shown to scale up nearly linearly (to about 20 processors) in a multiprocessor parallelization of the x-kernel [2]. Moreover, this behavior was observed within a single UDP/IP stream, whereas inter-stream scalability would be higher. Similar results are reported in [13]. These facts support our decision to neglect lock contention in the range of processor parallelization that we consider, which generally does not exceed 16.

5 Experimental measurements

To acquire the parameters needed by the runtime model, we measured the times 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. In each experiment, packets were constructed in-memory, simulating the arrival of a packets at the FDDI interface; a hardware timer with microsecond accuracy was then used to time receive-side protocol processing up the (suitably parallelized) UDP/IP/FDDI stack in the x-kernel.

The measurements reported here reflect protocol processing without software checksumming of the packet data. Checksumming can be performed in the x-kernel on our SGI Challenge at the rate of 32 bytes per microsecond [13]. We have verified experimentally that our measurements can be extended to reflect per-packet execution time with software checksumming by adding the appropriate fixed overhead (i.e., the per-byte checksumming overhead weighted by the number of bytes of data carried by the packet).

We start by demonstrating how we measure the six $t_{hot,\,hot}$ values for a packet executing at processor P_i, under Locking when threads do not migrate. The session-specific cache lines, which are always written, were previously written either (A) at P_i, or (B) at $P_j \neq P_i$. The protocol-specific cache lines (i.e., the active-map history values) are written if and only if the stream identifier changes from one packet to the next; thus, the lines are either (1) not written, (2) written and last written at P_i, or (3) written and last written at $P_j \neq P_i$. Let A1, A2, A3 and B1, B2, B3, denote the six cases.

To measure the corresponding $t_{hot,\,hot}$ values, experiments were designed with the general structure depicted in Figure 5. Thread T_0 running on processor P_0 acts as a producer thread and constructs packets. Each packet is from one of two streams, S_1 or S_2, which are endpoints on distinct hosts. Protocol threads T_1 and T_2 are wired to P_1 and P_2, respectively, and act as consumer threads. In all tests, producer and consumer threads synchronize via IRIX semaphores. The producer generates a packet from one of the endpoints, signals one of the waiting consumers, and blocks on its semaphore. The consumer receives the signal, starts its timer, receives the packet, unwinds its stack, stops the timer, signals the producer, and waits on its semaphore. A run is performed yielding the mean time over 1000 received packets. The $t_{hot,\,hot}$ value is computed as the mean over sufficiently large number of independent runs (100 in our experiments) to ensure that the 95% confidence interval half-width do not exceed 1% of the overall

Producer thread (on processor P_0)

```
forever {
   construct packet
   signal consumer
   await signal from consumer
}
```

Consumer threads (on P_1 and P_2)

```
forever {
   await signal from producer
   start timer
   receive stack (FDDI/IP/UDP)
   unwind stack
   stop timer
   signal producer
}
```

Figure 5: Thread behavior in the experimental environment

mean.

By carefully choosing the pattern of 1000 {stream ID, consumer ID} pairs which constitute one run of an individual experiment, we ensure the measured receive times include the appropriate migration overheads. The first five packets of each experiment are shown in Figure 6. Consider experiment A1, which is required to measure receive time when *(i)* the session-specific data was last written at P_i, and *(ii)* the protocol-specific data is not written. Requirement *(i)* is met since each packet is received by T_1, and *(ii)* is met since each packet is from stream S_1.

As a second example consider experiment A3, which requires that *(i)* session-specific data was last written at P_i, *(ii)* protocol-specific data is written, and *(iii)* protocol-specific data was last written at $P_j \neq P_i$. In the figure, we see that odd-numbered packets are from S_1 and are received by T_1, and even-numbered packets are from S_2 and are received by T_2. The fact that streams do not migrate ensures *(i)*, that stream identifier alternates between subsequent packets ensures *(ii)*, and that subsequent packets alternate between processors ensures *(iii)*.

Finally, consider as a third example experiment B2, which requires that *(i)* session writes were last written on $P_j \neq P_i$, *(ii)* protocol-specific lines are written, and *(iii)* protocol-specific lines were last written on P_i. In the figure we see that packets numbered $4j$-2 and $4j$-1 ($j \geq 1$) are received by T_2, and all others are received by T_1. Furthermore, odd-numbered packets are from S_1, even-numbered packets are from S_2, and only packets S_1 are timed. The fact that subsequent packets from S_1 alternate processors ensures *(i)*, that subsequent packets are from alternate streams ensures *(ii)*, and that (untimed) packet immediately preceding a timed packet was received by the same processor ensures *(iii)*.

With a slight modification, each of the six experiments yields a companion $t_{cold,\,hot}$ timing. Prior to starting its timer, the consumer issues sequential code and data references which exceed the size of the L1 cache. This ejects the protocol footprint from L1 since the L1 cache of the R4400 is virtually-indexed.

With another slight modification, the six experiments yield $t_{hot,\,hot}$ and $t_{cold,\,hot}$ timings when the thread

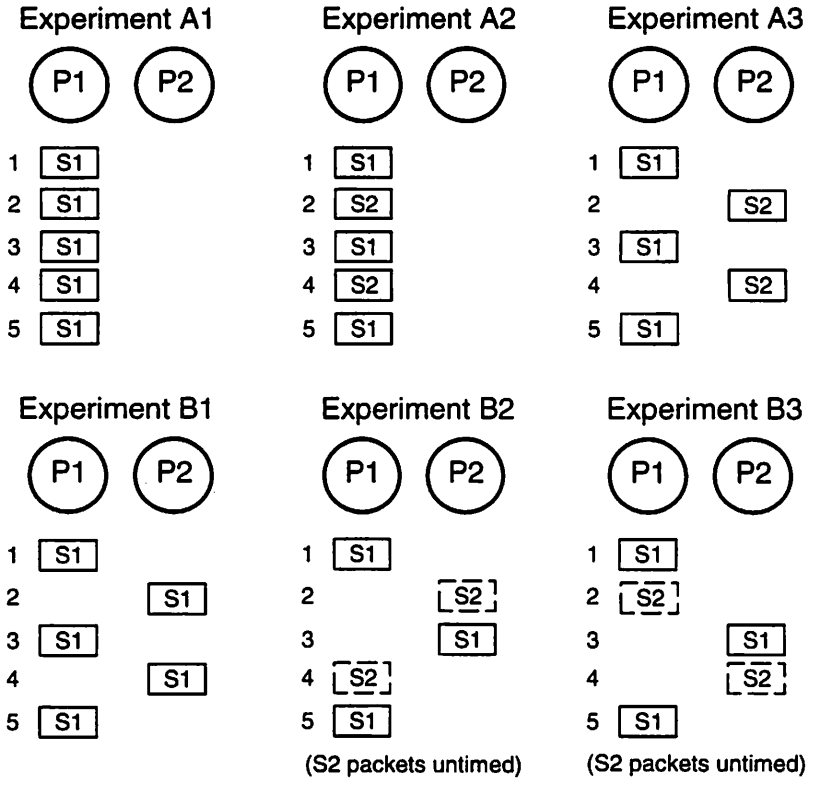


Figure 6: Design of micro-timing experiments. Each experiment consists of 1000 received packets. Each packet is from session S_1 or S_2, and is processed by the protocol thread on P_1 or P_2. The pattern of {stream ID, processor ID} pairs is chosen such that the measured receive time captures a particular combination of migration overheads.

Exp.	Overhead (see text)				Thread stack cached				Thread stack migrates			
					$t_{hot,\,hot}$		$t_{cold,\,hot}$		$t_{hot,\,hot}$		$t_{cold,\,hot}$	
	1	2	3	4	Locking	IPS	Locking	IPS	Locking	IPS	Locking	IPS
A1					66.6	57.4	91.9	80.5	86.8	77.3	110.2	98.5
A2			√		83.3	74.0	111.4	100.2	103.9	94.0	130.5	118.4
A3	√		√	√	103.0	87.5	131.2	113.8	124.6	108.4	148.9	130.7
B1	√	√			105.1	77.0	129.0	100.1	125.8	97.4	146.9	117.0
B2		√	√		111.3	89.7	139.9	116.9	130.8	108.7	155.3	131.6
B3	√	√	√	√	130.2	102.5	158.1	128.9	149.9	121.9	173.1	143.9

Table 2: x-kernel UDP/IP/FDDI receive times (in μs), for a single packet without checksumming.

stack migrates. Prior to signaling the consumer, the producer zeros the consumer's thread stack, migrating it to P_0. Each of the consumer's timings then includes the overhead of migrating the thread stack.

Table 2 shows the results. (Since these are receive times for packets without software checksumming, the timings are independent of packet size.) To help interpret the data, we have indicated the categories of execution-time overhead which are incurred in each experiment. Category 1 indicates the overhead of migrating the locks which enable protocol parallelization (under Locking, per-protocol active map locks; under IPS, a single per-stack concurrency lock). Category 2 is the overhead of migrating session-specific cache lines (the session reference counters, IP-specific data, and associated synchronization locks under Locking). Category 3 is the overhead of active-map lookups and writing of each active map's history location when it is cached. Category 4 is the overhead of migrating the active map history locations.

Note in Table 2 that the best-case per-packet mean delays (experiment A1) are 57.4 and 66.6 μs under IPS and Locking, respectively. Thus when all locks are cached, parallelization by Locking adds just 16% overhead to packet delay. But when sessions migrate (experiment B1), the overhead grows to 37% (105.1μs under Locking vs. 77.0μs under IPS). The large increase reflects the migration of the cache lines underlying the software locks. Thus the overhead of Locking (over IPS) is dominated by the *migration* of locks when sessions migrate, not the lock acquisition and release times. Also, note in the table that the overhead of migrating the thread stack is about 20μs, a significant fraction of packet execution time. Finally, the data indicate that code affinity is important: $t_{cold,\,hot}$ is 20-40% larger than $t_{hot,\,hot}$ across all measurements.

The two $t_{cold,\,cold}$ measurements are obtained as follows. After some initialization activity in which consumer threads T_1 and T_2 receive packets at P_0, T_1 times the reception of 7 packets from S_1 received at P_1 through P_7 respectively. Since each processor cache is completely flushed at the outset of the test, these measurements yield (seven) $t_{cold,\,cold}$ timings for the given parallelization alternative.

We find that $t_{cold,\,cold} = 284.3\mu s$ under Locking, and 232.2μs under IPS. These times are 2-4 times larger

than $t_{cold,\,hot}$, reflecting the relative slowness of main memory. This further suggests that processor scheduling is important, and that code affinity can have a potentially much higher impact on protocol execution time than the other affinities.

Finally, to measure *overhead_acquire*, before signaling the consumer, the producer writes the free-memory pool data structures which are accessed by the consumer in acquiring free memory. This ensures that overhead of their migration will be incorporated into the consumer's timing. Similarly, to measure *overhead_release*, the producer writes the free-memory pool data structures which are accessed by the consumer in releasing free memory. The results are $4.05\mu s$ and $4.52\mu s$, respectively. Although these are relatively small in comparison to the measurements of Table 2, keep in mind that at a high arrival rate both would be incurred on a per-packet basis.

Having discussed the experimental design and presented the timing measurements used to parameterize the analytic component of the simulation model, we now turn our attention to the simulation results.

6 Results

In this section, we examine the impact that affinity scheduling has on performance under Locking, and then under IPS. Next, we compare Locking with IPS, for both unbatched and batched arrivals. We show that a "hybrid" approach, which can be utilized by a particular class of streams, gains the performance advantages of each. We also compare each of the parallelization alternatives along the dimension of intra-stream scalability.

We focus on the case of packet processing without software checksumming. While this is advantageous from the perspective of affinity scheduling, there are several mitigating considerations. Checksumming may be supported in hardware, as is the case with the FDDI interface board on our SGI Challenge, obviating the need for software checksumming. In addition, it has been shown that in many high-traffic environments, the majority of packets are frequently small. Since a small packet can be checksummed relatively quickly (e.g., on our platform a 256-byte packet can be checksummed in about $8\mu s$), for such common workloads our conclusions about affinity scheduling would still hold. Finally, we illustrate explicitly in section 6.3 how our simulation results can be interpreted to reflect the overhead of software checksumming.

We give results for a host with $N = 8$ processors. Results for N varying from 2 to 32 have been obtained, in all cases exhibiting similar trends. For simplicity, the number of admitted UDP/IP/FDDI streams is held equal to the number of processors across all tests. Under IPS, the number of independent stacks also matches the number of processors, and streams are assigned one-to-one to independent stacks. Thus, unless otherwise stated, all graphs that follow are for 8 processors, 8 streams, and (where applicable) 8 independent stacks.

6.1 Performance of affinity scheduling under Locking

Figures 7 and 8 explore the performance of affinity scheduling under Locking, plotting mean packet delay as a function of packet arrival rate. To isolate the marginal performance contributions, we consider the impact of processor scheduling (Figure 7(a)) independently from that of thread and free-memory scheduling (Figure 7(b)).

Figure 7(a) demonstrates that MRU outperforms LRU and Wired-Streams processor scheduling across the broad range of packet arrival rates. MRU schedules for protocol code affinity because the processor most recently visited by protocol processing is the one most likely to have the protocol footprint cache-resident. LRU, on the other hand, selects the processor whose cache is *least* likely to contain the protocol footprint; Wired-Streams selects processors randomly since an arrival's stream identifier is assigned randomly. Thus, we see that in the lower arrival rates, MRU does much better than LRU (decreasing packet delay by about 25%), with Wired-Streams falling in between. The three curves rise as the arrival rate decreases because non-protocol processing flushes the processor caches. As the packet arrival rate tends to 0, the protocol receive time tends to $t_{cold,\,cold}$ (284.3μs), because protocol processing more frequently finds a completely cold cache hierarchy at every processor. Toward the other extreme, as the arrival rate becomes large, all three curves rise due to packet queueing. Note that the Wired-Streams curve rises much faster than the LRU and MRU curves. This is because Wired-Streams management is non-work-conserving.

In general, the benefit of scheduling for code affinity diminishes with increasing packet arrival rate, because the read-only protocol code, which can be replicated by the cache coherency protocol, is increasingly retained in the processor caches (i.e., is less frequently displaced by non-protocol processing). Thus, the MRU and LRU curves converge at high arrival rate.

In Figure 7(b), there are four curves, corresponding to whether or not threads and free-memory are affinity scheduled. It is evident that thread scheduling yields a greater reduction in packet delay (10-15%) than free-memory scheduling (5-10%); both yield significant delay reduction across the broad range of arrival rates.

In order to expose the influence of affinity scheduling on the host's maximum supportable arrival rate, we focus in Figure 8 on the high arrival rate regions in Figure 7. There are two items of note. First, in Figure 8(a), under Wired-Streams processor scheduling the maximum supportable arrival rate is about 20% higher than under MRU and LRU scheduling (which converge under high load, as previously noted). Wired-Streams yields a higher throughput due to its lower per-packet processing time (103μs) than MRU and LRU (125 μs) at high load, which is due to the fact that the stream-specific cache lines do not migrate under Wired-Streams scheduling, but do under the other policies. Second, in Figure 8(b), we see that free-memory pool scheduling increases the maximum supportable throughput by about 5%, reflecting the fact that the cache lines associated with the free-memory pool migrate with every packet at high arrival rate. In contrast, thread scheduling does not impact the maximum supportable arrival rate, since threads rarely release processors under high arrival rate.

6.2 Performance of affinity scheduling under IPS

To examine the benefits of affinity scheduling under IPS, consider Figures 9 and 10, which are the analogs of Figures 7 and 8, respectively.

Figure 9(a) shows that the processor scheduling policies behave differently under IPS than under Locking. Across the broad range of arrival rates, Wired-Streams performs best. This is because under IPS, all policies are work-conserving (which is not the case under Locking), and Wired-Streams performs best since it minimizes cache misses at high load. In further constrast to the behavior under Locking, Figure 10(a) illustrates that under IPS, the processor scheduling policies are undifferentiable with respect to the hosts's maximum supportable arrival rate. This is because under IPS, maximum affinity is achieved at high arrival rates regardless of processor scheduling policy (which is not the case under Locking). Under IPS, streams are statically assigned to independent stacks, and stacks rarely migrate under high load since processors are infrequently released by protocol threads.

Figure 9(b) (in conjunction with Figure 7(b)) demonstrates that the general behavior of thread and free-memory scheduling is independent of parallelization alternative. Note however that under IPS, the increase in the host's maximum supportable arrival rate enabled by affinity-scheduling free-memory is significantly higher (15%, Figure 10(b)) than it is under Locking (5%, Figure 8(b)). This is due to the lower per-packet processing time under IPS ($57.4\mu s$) than under Locking ($125\mu s$).

6.3 Decreased latency in end-to-end communication

Low-latency request/response communication, as measured from the perspective of a client application, necessitates an end-to-end metric capturing the total time spent in client protocol processing, network access and propagation delay, server protocol processing, server computation, and the return trip. Because affinity scheduling only impacts the protocol processing component of this communication path, its benefit is necessarily diminished in the context of an end-to-end metric. Furthermore, in the common case of a uniprocessor client communicating with a multiprocessor server, the benefits of affinity scheduling are limited to the decrease in protocol processing time at the server.

To formalize the end-to-end metric, let p_c denote client protocol processing time, τ denote the propagation time, p_s denote server protocol processing time, and C denote the server computation time. The end-to-end communication latency, E, is the sum of the components along the end-to-end path: $E = 2p_c + 2\tau + 2p_s + C$. We compute E with p_s computed under affinity scheduling (call this E_a), and with p_s when not affinity scheduling (call this E_n). The fractional reduction in end-to-end latency enabled by affinity scheduling is given by $(E_n - E_a)/E_n$.

Figures 11(a) and 11(b) graph this metric under Locking and IPS, respectively. In the figures, the fixed-overhead components of end-to-end latency $(2p_c + 2\tau + C)$ have been aggregated and expressed as a multiple V of the overall best-case receive time ($57.4\mu s$). The figures show the maximum fractional reduction in end-to-end latency

$(E_n - E_a)/E_n$ when the fixed overhead components are $V = 0, 5, 10$, and 20 times larger than the minimum packet receive time. The $V = 0$ curve is in fact an *upper bound*, since it models infinitely fast propagation and server response time.

The figure shows that affinity scheduling can result in a significant reduction in end-to-end latency, even when the fixed overhead components of communication are large with respect to message processing time. The reduction is greater under IPS than Locking, reflecting the lower per-packet protocol processing cost under IPS. The upper bound on the reduction (as given by the $V = 0$ curves) is around 40-50%.

Figures 11(a,b) can be used to discern the impact of software checksumming on the benefits of affinity scheduling. Checksumming represents a fixed, per-packet overhead of about $1\mu s$ for every 32 bytes of packet data (section 5). Consider the worst case (from the perspective of affinity scheduling) of a stream transmitting the largest possible FDDI packets, each with 4432 bytes of data. The fixed overhead would be $139\mu s$ per packet. Although the corresponding value of $V = 2.4$ is not plotted in Figures 11(a,b), we can see from the $V = 5$ curves that affinity scheduling would still yield significant reduction in packet delay.

6.4 Comparing parallelization alternatives

We now turn to a comparison of Locking with IPS. Figure 12 shows the performance of both MRU and Wired-Streams processor scheduling under both Locking and IPS. For all curves, threads and free-memory are affinity-managed. The figure demonstrates that mean packet delay under IPS is about 50% lower than under Locking (which is advantageous from the perspective of an individual stream). In addition, the host's maximum supportable arrival rate is 79-118% higher under IPS than under Locking (which is advantageous from the perspective of the host). For unbatched arrivals, then, we conclude that IPS offers significantly better performance than Locking, from both the perspective of an individual stream, and the perspective of the host.

Figure 12 does not emphasize that MRU processor scheduling under Locking is work-conserving, whereas Wired-Streams under Locking and both policies under IPS are not. To highlight this distinction, we examine the behavior of Locking and IPS under batched arrivals.

Figures 13(a,b) compare Locking and IPS for batched arrivals under MRU processor scheduling, where the batch size distribution is either deterministic (Figure 13(a)) or geometric (Figure 13(b)) with mean b. In each figure, curves for b =2,4,8 are plotted. Figure 13(a) shows that Locking is much more robust to batched arrivals than IPS, providing lower mean packet delay for all b. Note however that for all b, IPS supports a much higher maximum arrival rate. Figure 13(b) shows further deterioration of the performance under IPS when the batch size distribution is geometric. This is due to the increased burstiness of the arrival process under the geometric batch size distribution. Under batched arrivals, then, we conclude that Locking performs better from the perspective of an individual stream,

but IPS performs better from the perspective of the host.

6.5 A hybrid approach

An intriguing question is whether there exists a hybrid approach exhibiting simultaneously robustness under bursty arrivals (á la Locking) and high-capacity maximum supportable arrival rate at the host (á la IPS). We can identify such an approach under a particular workload constraint. Under IPS, we have assumed that each incoming stream is assigned to one independent stack. This is a necessary restriction for streams which have inter-packet dependencies in terms of the data structures that must be referenced during packet processing. For example, the packets sent on a TCP stream must visit the same independent stack, since TCP ensures reliable, in-order delivery. The same holds for fragmented IP, for which incoming fragments must visit the same independent stack for reassembly. However, for a stream without inter-packet dependencies (such as an unfragmented UDP/IP/FDDI stream) the assumption is overly restrictive. Instead, the receiver might perform "multiple open's" on the independent stacks, thereby enabling any incoming packet to be routed to any protocol stack. We refer to such streams as *multiple-open streams*. Intuitively, multiple-open streams under IPS should exhibit robustness in the presence of bursty arrivals (since there can be a single global packet queue), while simultaneously enabling high throughput capacity at the host (since processing is IPS-based).

Figure 14 shows the performance of this "Hybrid" approach, along with that of Locking and single-open streams under IPS, for geometric batch size distribution with mean batch sizes b=1,4. For all curves, processor scheduling is MRU and threads and free-memory are organized for maximum affinity. The figure demonstrates that the host's maximum supportable arrival rate under the Hybrid is close to that of single-open streams IPS. In addition, it is evident that the Hybrid exhibits response to intra-stream burstiness which is similar to that of Locking. This suggests that for streams transmitting packets that can be processed independently (such as unfragmented datagram traffic), performing multiple-open's on the independent stacks under IPS may offer the best overall performance.

6.6 Intra-stream scalability

Finally, we compare the parallelization alternatives (Locking, IPS under single-open streams, and the Hybrid approach) along the dimension of intra-stream scalability. Figure 15 shows the host's maximum supportable arrival rate for each parallelization alternative, when receiving a single incoming stream, as a function of number of available processors. The slopes of the curves are computed as follows. For the Hybrid curve, under Wired-Streams processor scheduling, the slope derives from the 57.4μs packet receive time of IPS experiment A1 (Table 2). For the Locking curve, the slope derives from the 105.1μs packet receive time of Locking experiment B1. Finally, the slope of the IPS single-open curve is zero, since intra-stream parallelism is not permitted. The figure demonstrates that for

multiple-open streams, the Hybrid approach provides the best intra-stream scalability. For single-open streams, however, Locking outperforms IPS when more than two processors are available for protocol processing.

7 Related Work

Vaswani and Zajorian [26] show experimentally that affinity scheduling within kernel-level processor space-sharing scheduling policies provides little benefit. Their workload is a mix of three types of parallel applications executing on a Sequent Symmetry multiprocessor. The measured reduction in task response time enabled by affinity scheduling does not exceed 1%. The explanation lies in the fact that among these applications, the upper bound on the time to completely reload the processor cache (about 1-2ms) is small in comparison to the processor reallocation interval (about 200-500ms). In contrast, we demonstrate (on our platform) that protocol cache reload times are large in comparison to protocol execution times—and that affinity scheduling is effective.

Squillante and Lazowska [22] conduct a modeling study designed to gain insight into the general class of scheduling policies which consider the state of processor caches. The study examines the performance of a range of in-kernel affinity scheduling policies on a multiprocessor system running multiple independent single-threaded processes. This work motivated our own by demonstrating that when the cache reload time is large with respect to the task's inherent computing demands, affinity scheduling can have a significant impact. The authors use a variety of queueing-theoretic techniques and employ an analytic cache model developed by Thiebaut and Stone [25]. We rely on the same model—with the distinctions noted in section 4.4—coupled with additional analytic results from Singh, Stone and Thiebaut [20]. However, Squillante and Lazowska do not identify specific applications which stand to benefit from affinity scheduling. We have shown that parallelized protocol processing is one such application.

In [3], Devarakonda and Mukherjee explore implementation issues in affinity scheduling, both in-kernel and within a user-level thread scheduler, on an 8-processor Encore Multimax running Mach 2.5. A schedulable task is defined to have affinity strictly for the processor it most recently visited. They consider two real parallel applications and a synthetic application designed explicitly to benefit from affinity scheduling. Although experimental measurements do not support in-kernel affinity scheduling, within the user-level thread scheduler the authors find affinity scheduling yields a a 12% reduction in execution time for one of the two real applications. The authors conclude that affinity scheduling may be beneficial within user-level thread scheduler for some multithreaded parallel applications.

In a trace-driven simulation study, Gupta, Tucker and Urushibara [5] consider in-kernel affinity scheduling of parallelized applications on a shared memory platform. Their approach is to simulate a multiprocessor system by interleaving the process execution traces obtained from a set of parallel scientific applications. The simulation assumes a simple architectural model: each of 12 processors has a private 64KB unified cache which can service a

request in a single cycle on a cache hit, and in 20 cycles on a cache miss. Otherwise, instruction execution time is one cycle; overhead due to bus contention or cache coherence is not captured. The simulation results show that affinity scheduling yields a small but consistently positive impact across all applications, increasing processor utilization by an average of about 3% overall.

8 Summary

High-performance protocol processing is in-demand as high-speed networks and large-scale servers gain prevalence. We have presented evidence that affinity scheduling can reduce packet delay by a substantial margin, in turn enabling the host to concurrently support a higher number of streams and offer higher throughput capacity to individual streams. The delay reduction can translate to a large fractional decrease in a client's end-to-end latency, even when the fixed-overhead components of communication delay are large with respect to message processing time. A broad class of applications stand to benefit, e.g. those which perform IPC or RPC in a distributed multiprocessor environment.

We have implemented two approaches to parallelizing protocol processing in our experimental environment and measured their performance. Locking achieves parallelism by protecting access to shared data structures with software locks. IPS achieves parallelism through multiple independent protocol stacks. The key ideas are that *(i)* cache misses are unavoidable when multiple processors access a shared item protected by a software lock; *(ii)* a single cache miss adds a relatively large overhead (around 5%) to packet processing time; and *(iii)* IPS avoids these cache misses by parallelizing without software locks. We find IPS generally delivers much lower latency to the client and enables a much higher throughput capacity at the server. Yet because IPS does not parallelize processing on individual streams, it offers limited intra-stream scalability and is sensitive to intra-stream burstiness. These observations lead us to propose a hybrid approach for a particular class of streams, which we show exhibits the best overall performance.

We have demonstrated that affinity scheduling in this domain involves concurrent management of multiple distinct resources. Importantly, we have restricted ourselves to relatively simple scheduling algorithms with relatively straightforward implementations. For high performance, protocol threads and free-memory pools should be organized on a per-processor basis. Under Locking, processors should be managed MRU—except under high arrival rate, when Wired-Streams scheduling performs better. Under IPS, independent stacks should be wired to processors—except under low arrival rate, when MRU processor scheduling performs better.

There are several possible extensions to the work presented in this paper. First, we plan to examine the performance of affinity scheduling for send-side protocol processing. We conjecture a higher performance gain since the send-side execution path is shorter (e.g., there is no packet demultiplexing). Second, recent work indicates

that while some classes of network traffic are well-modeled as Poisson, others are not [15]. We plan to investigate how alternative models of packet arrivals ([9, 15]) impact our results. Third, it will be interesting to assess the performance of affinity scheduling as a function of recent architectural trends. We plan to perform affinity-based measurements across the evolution of SGI multiprocessor platforms, including the older Power Series (with 33-MHz R3000 processors and single-level processor caches), Challenges with higher processor clock rates (e.g., the 150-MHz R4400), and the new Power Challenge (with R8000 processors). Fourth, it will be interesting to consider the performance of affinity scheduling of protocol code with greater complexity, such as IP with fragmentation or a more complex transport protocol such as TCP. Finally, it may prove interesting to more carefully examine the packet queueing discipline. At low arrival rates it may be advantageous for a packet to wait for a executing protocol thread, instead of going immediately into service on a "cold" processor. A simple heuristic may perform well, e.g., given N executing protocol threads, to unblock an idle protocol thread when the length of the packet queue exceeds $N t_{cold,\,cold}/t_{hot,\,hot}$.

Our conclusions contrast with those of several earlier studies [3, 5, 26] and support the contention [22] that there are platforms and common workloads for which affinity scheduling is worthwhile. We hope to have demonstrated techniques and a methodology which will facilitate further research in this area. In light of current architectural trends in which processor speedups outpace those of main memory, we anticipate affinity scheduling to be of increasing interest.

Acknowledgments

Erich Nahum and David Yates are credited with 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] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] Mark D. Hill and Alan J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[8] 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.

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

[10] Steve Kleinman. Symmetric multiprocessing in Solaris 2.0. In *IEEE Spring COMPCON*, San Francisco, CA, 1992.

[11] M. Kobayashi and M. MacDougall. The stack growth function: Cache line reference models. *IEEE Transactions on Computers*, 38(6):798–805, June 1989.

[12] 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.

[13] 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)*, Monterey, CA, November 1994.

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

[15] 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, UK, August 1994.

[16] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from multithreading System V Release 4. In *Proceedings of the Third USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pages 77–91, Newport Beach, CA, March 1992.

[17] David Presotto. Multiprocessor STREAMS for Plan 9. In *UKUUG*, January 1993.

[18] 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.

[19] 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.

[20] 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.

[21] A. J. Smith. Line (block) size choice for CPU cache memories. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

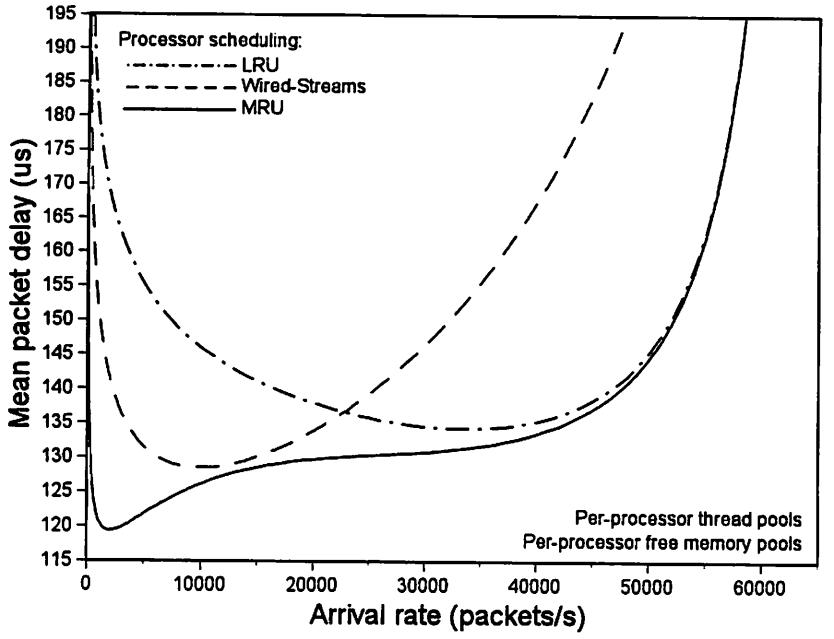
[22] Mark S. Squillante and Edward D. Lazowska. Using processor cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[23] Dominique F. Thiebaut. *Influence of program transients in computer cache-memories*. PhD thesis, Univ. Massachusetts, 1989.

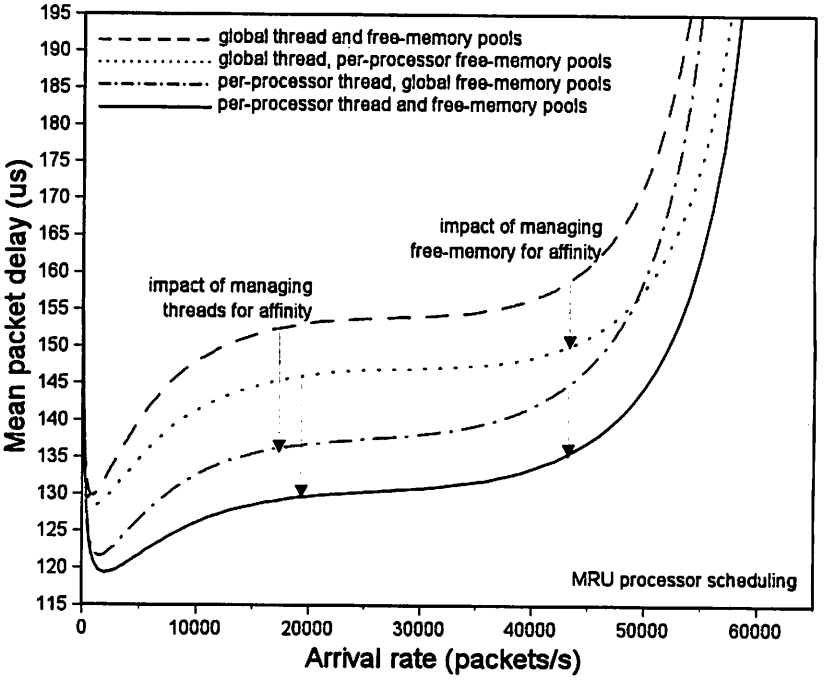
[24] Dominique F. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7):1012–1026, July 1989.

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

[26] 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.

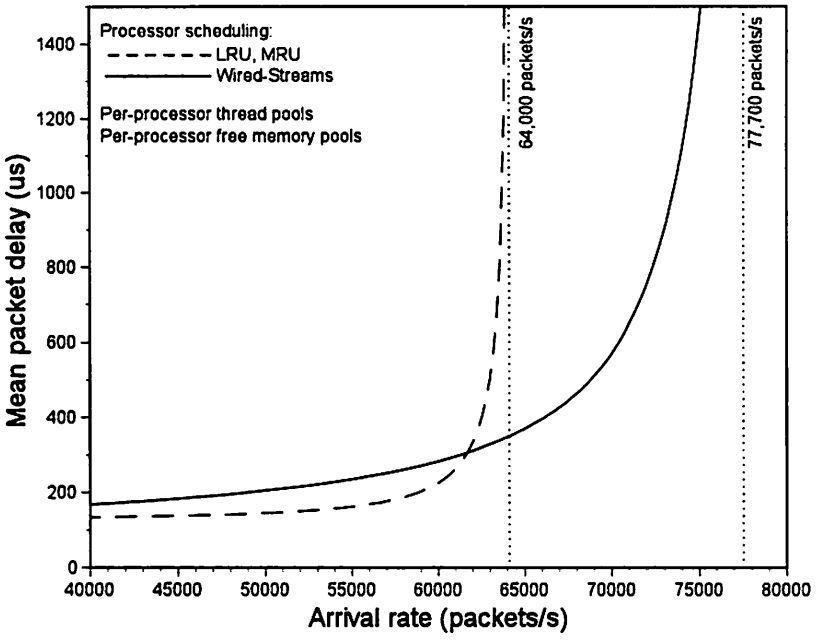


(a) Impact of processor scheduling policy

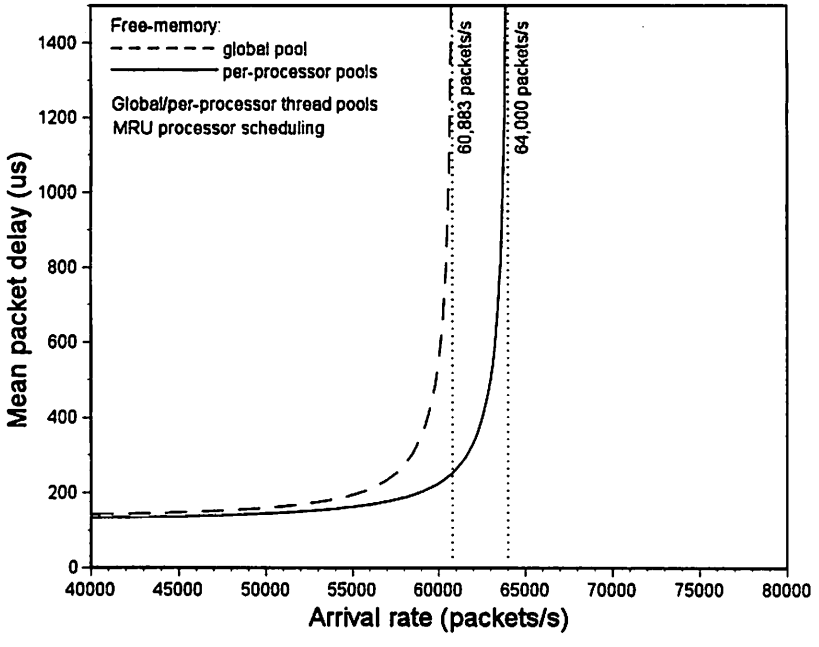


(b) Impact of thread pool and free-memory pool organization

Figure 7: Components of affinity scheduling under Locking.

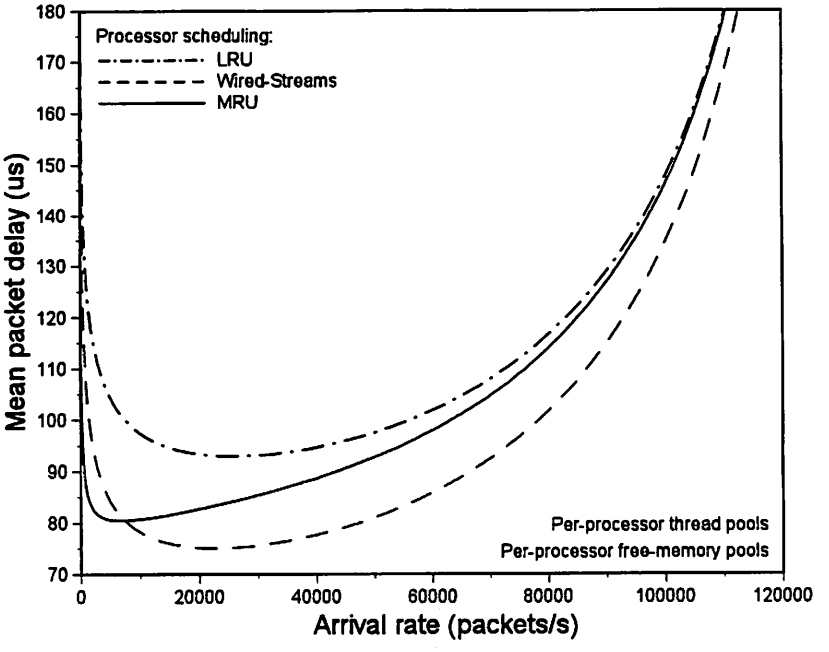


(a) Impact of processor scheduling policy

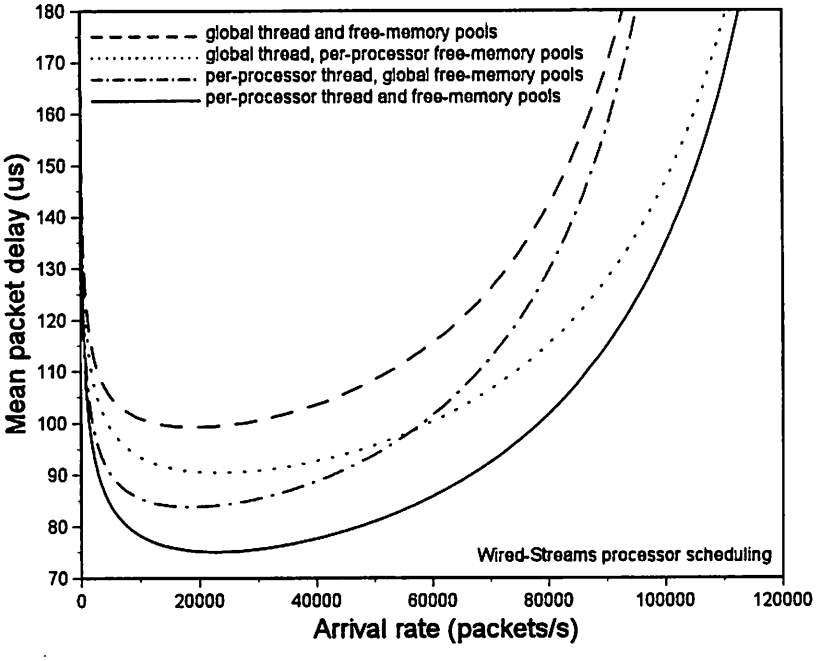


(b) Impact of thread pool and free-memory pool organization

Figure 8: Influence of affinity scheduling on maximum supportable arrival rate under Locking.

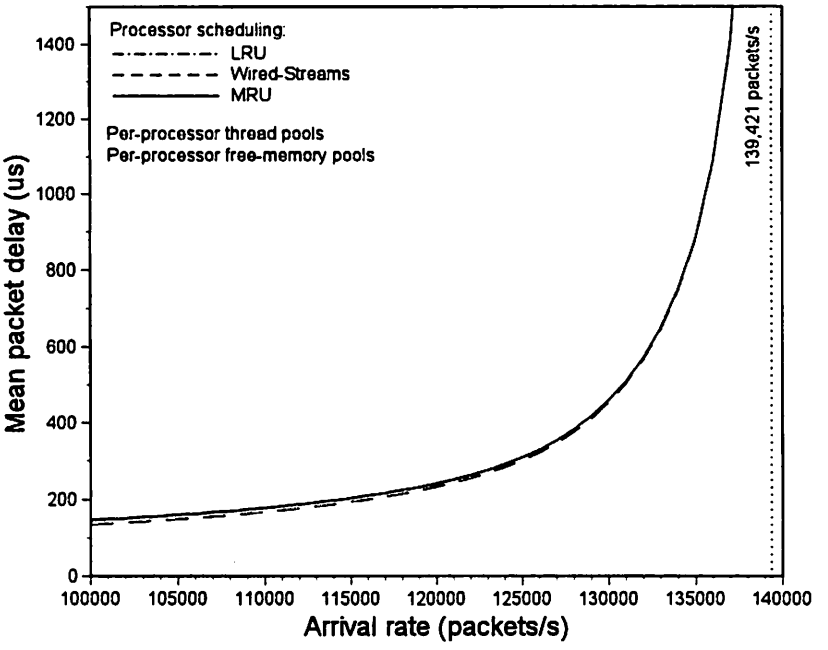


(a) Impact of processor scheduling policy

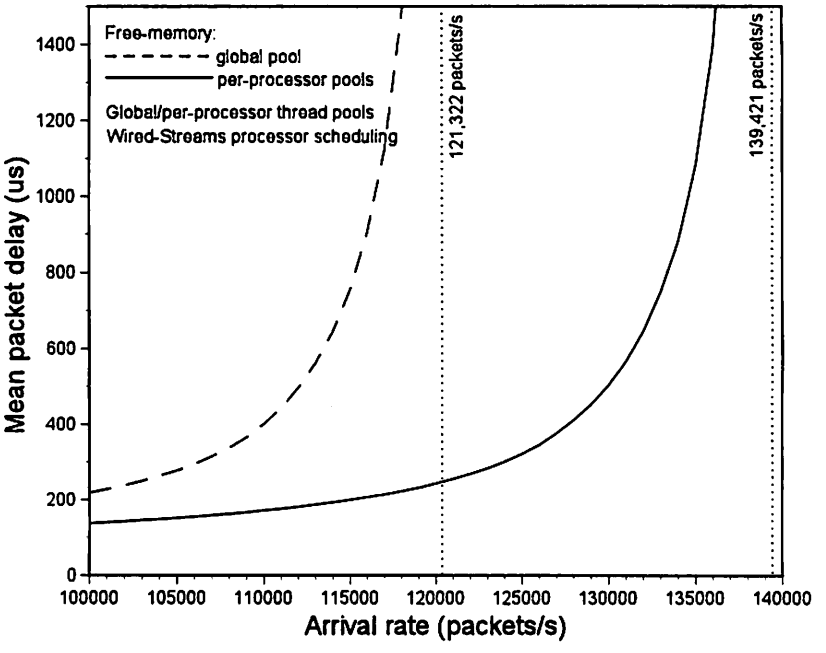


(b) Impact of thread pool and free-memory pool organization

Figure 9: Components of affinity scheduling under IPS.

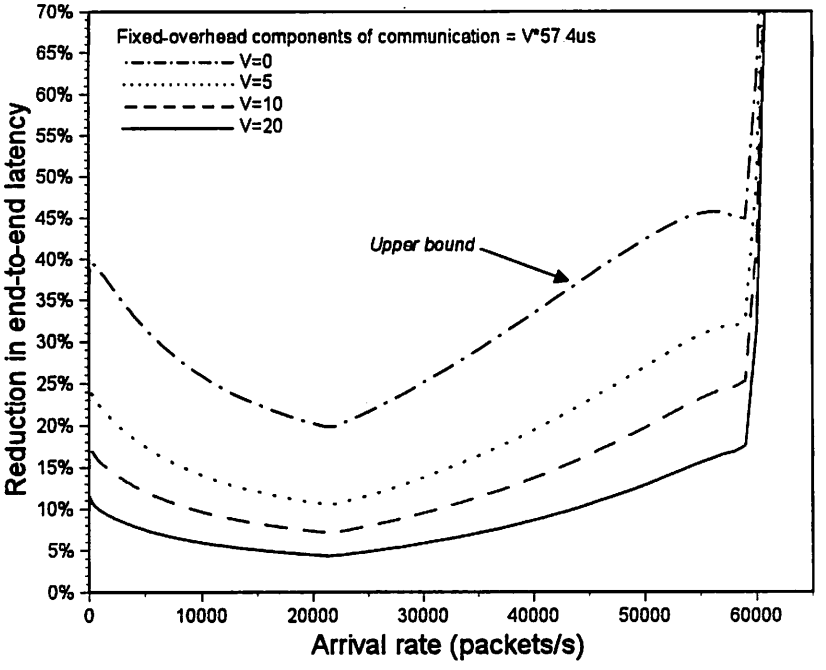


(a) Impact of processor scheduling policy

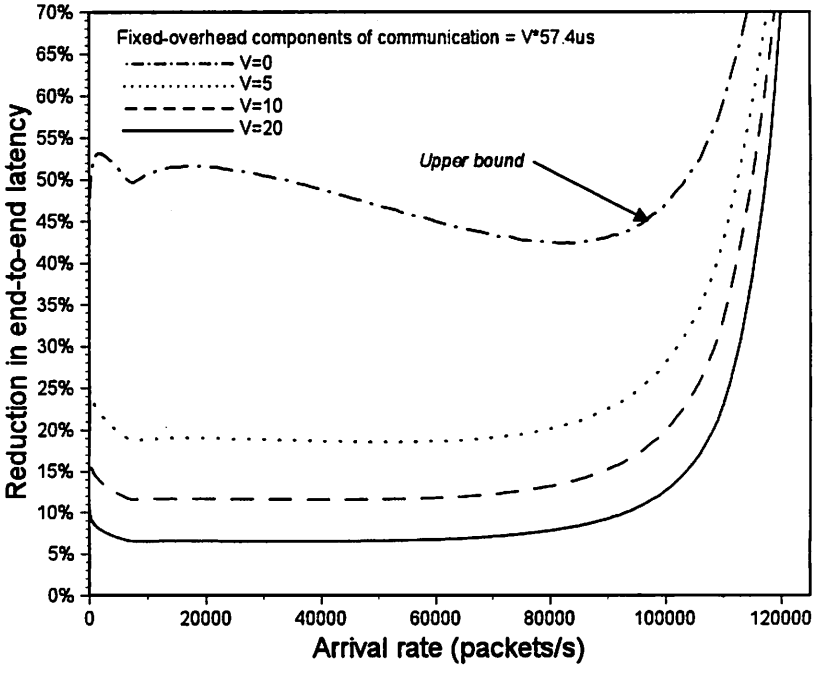


(b) Impact of thread pool and free-memory pool organization

Figure 10: Influence of affinity scheduling on maximum supportable arrival rate under IPS.



(a) Locking



(b) Independent Protocol Stacks

Figure 11: Maximum fractional reduction in end-to-end latency enabled by affinity scheduling. Fixed-overhead components of communication—client networking, propagation delay, and server computation time—have been expressed as a multiple V of the best-case packet receive time.

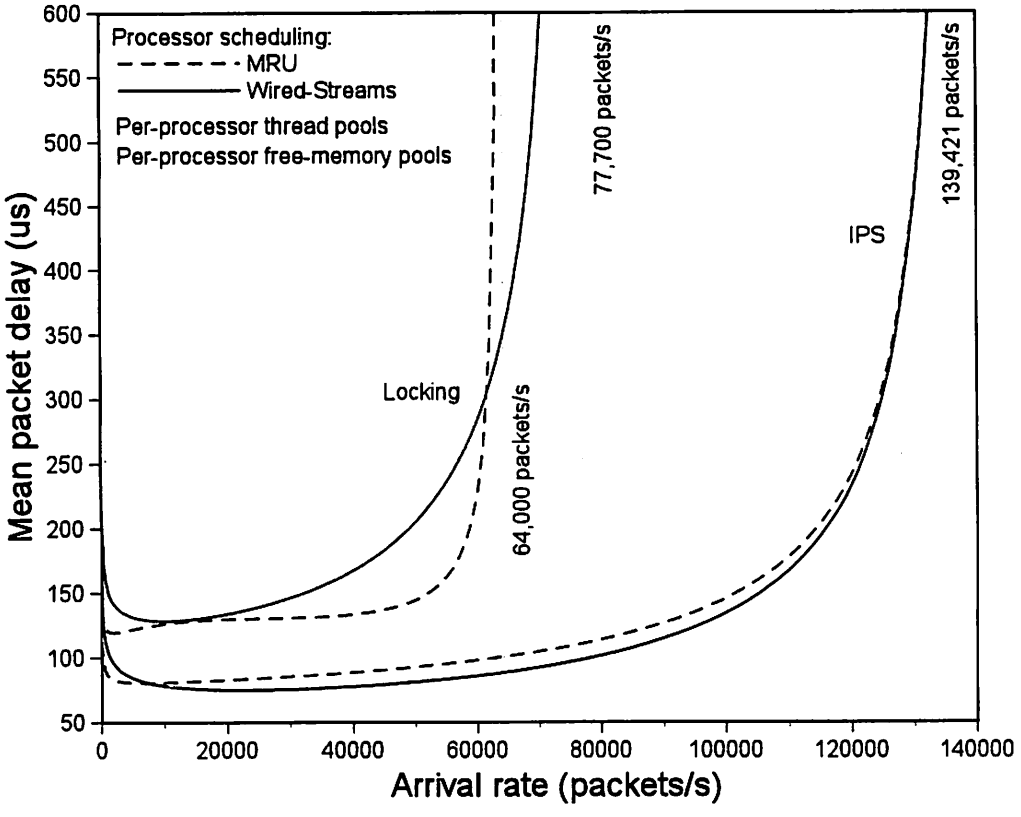
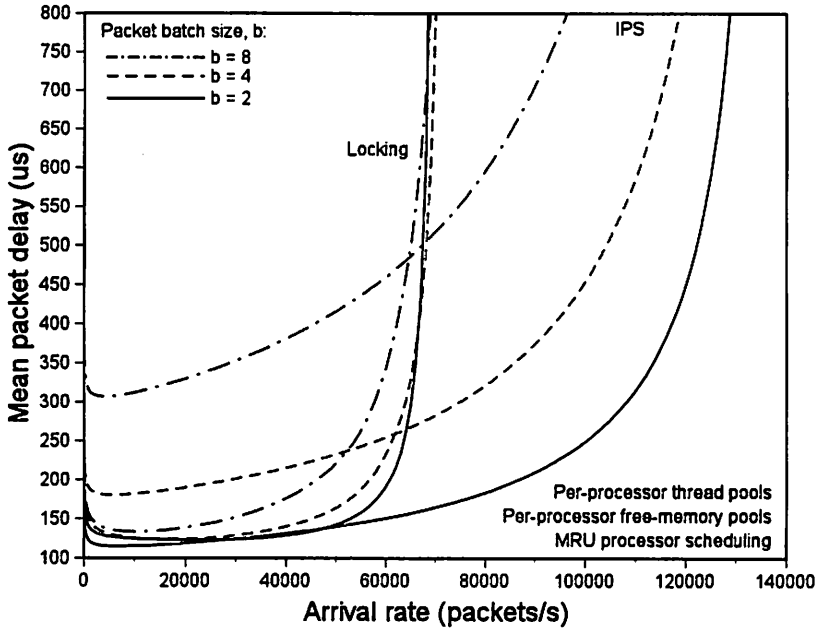
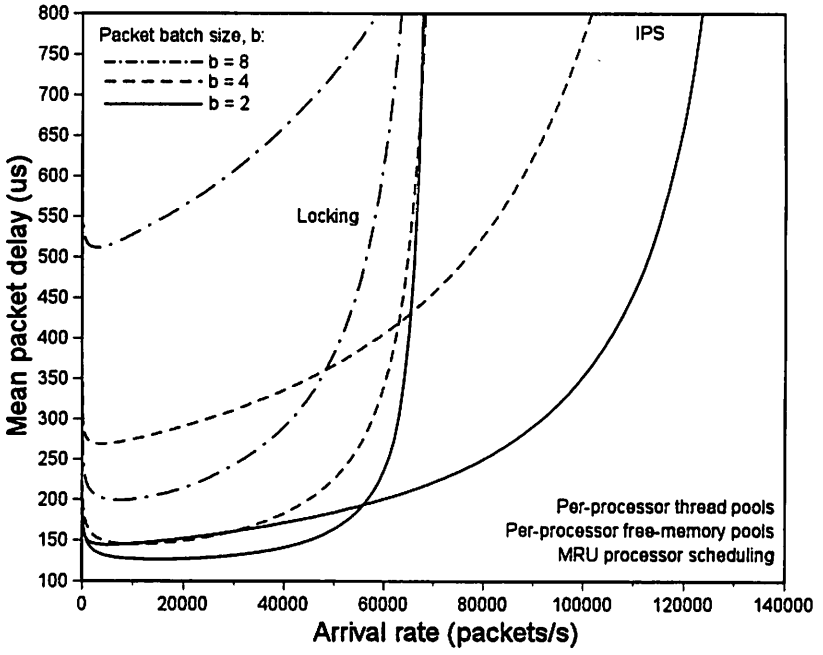


Figure 12: Comparing parallelization alternatives for unbatched arrivals.



(a) Deterministic batch size distribution



(b) Geometric batch size distribution

Figure 13: Comparing parallelization alternatives for batched arrivals.

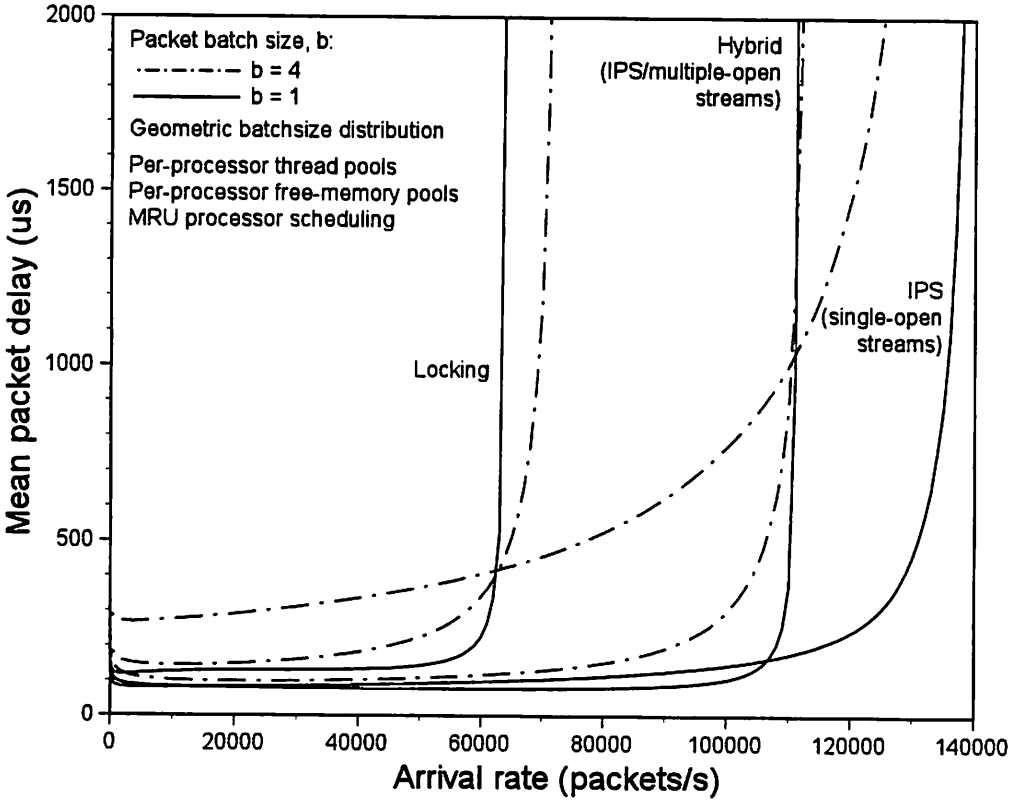


Figure 14: Performance of the Hybrid approach.

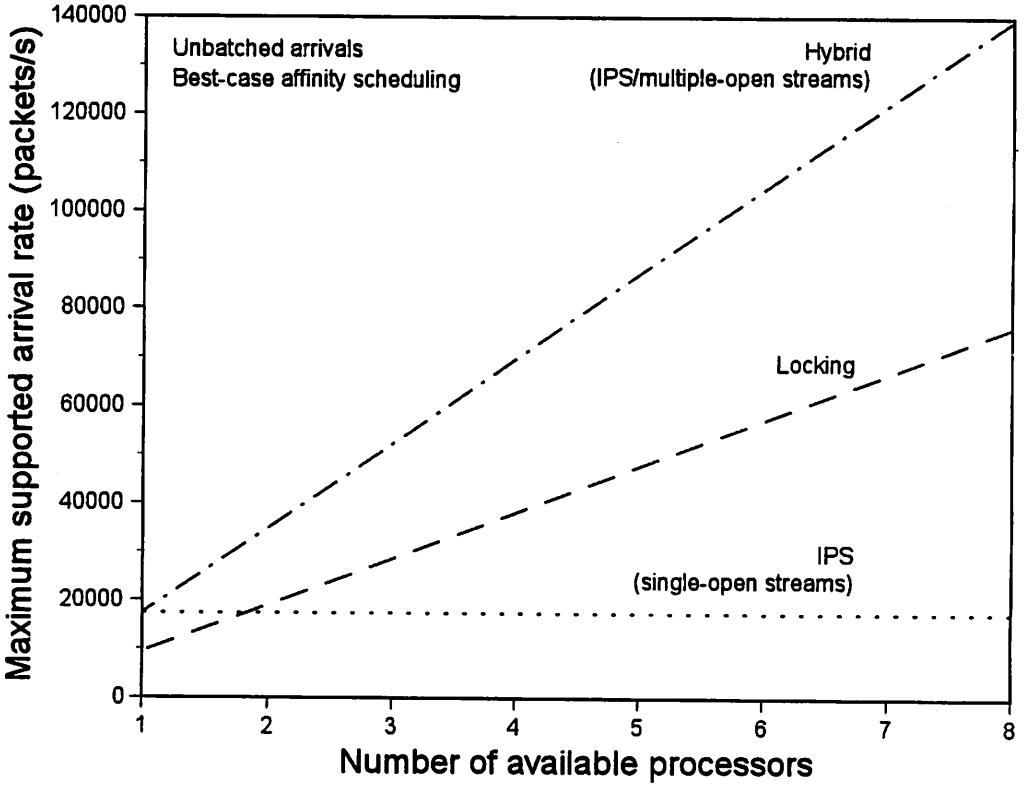


Figure 15: Comparing parallelization alternatives with respect to intra-stream scalability (1 stream).