The Performance of Alternative Threading Architectures for Parallel Communication Subsystems

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Comp. Sci.
Washington University
St. Louis, MO 63130
314 935-6160

Tatsuya Suda
suda@ics.uci.edu
Info. and Comp. Sci. Dept.
University of California, Irvine
Irvine, California, 92717
(714) 856-4105[1]

This paper has been submitted to the Journal of Parallel and Distributed Computing.

Abstract

A communication subsystem consists of protocol tasks and operating system mechanisms that support the configuration and execution of protocol stacks composed of protocol tasks. To parallelize a communication subsystem effectively, careful consideration must be given to the threading architecture. The threading architecture binds processing elements with the protocol tasks and the messages associated with protocol stacks in a communication subsystem. This paper makes two contributions to the study and application of threading architectures. First, it reports performance results from empirical comparisons of two protocol stacks (based on the connectionless and connection-oriented transport protocols UDP and TCP) using different threading architectures on a 20 CPU multi-processor platform. The results demonstrate how and why different threading architectures affect performance. Second, the paper provides guidelines based on these results that indicate when and how to apply appropriate threading architectures.

1 Introduction

Advances in VLSI and fiber optic technology are shifting performance bottlenecks from the underlying networks to the communication subsystem [1, 2]. A communication subsystem consists of *protocol tasks* and *operating system mechanisms*. Protocol tasks include connection establishment and termination, end-to-end flow control, remote context management, segmentation/reassembly, demultiplexing, error protection, session control, and presentation conversions. Operating system mechanisms include process and thread management, timer-based and I/O-based event demultiplexing, message buffering, and layer-to-layer flow control. Together, protocol tasks and operating system mechanisms support the implementation and execution of communication protocol stacks composed of protocol tasks [3].

A promising technique for increasing protocol processing performance is to multi-thread protocol stacks and execute them on multi-processors [1]. Significant increases in performance are possible, however, only if the speed-up obtained from parallelism outweighs the *context switching* and *synchronization* overhead associated with parallel processing. A context switch is triggered when an executing thread relinquishes its associated processing element (PE) voluntarily or involuntarily. Depending on the underlying OS and hardware platform, a context switch may require dozens to hundreds of instructions to flush register windows, memory caches, instruction pipelines, and translation look-aside buffers [4]. Synchronization overhead arises from locking mechanisms that serialize access to shared resources (such as message buffers, message queues, protocol connection records, and demultiplexing maps) used during protocol processing [5].

A number of threading architectures [5, 6, 7, 8] have been proposed as the basis for parallelizing communication subsystems. There are two fundamental types of threading architectures: *task-based* and *message-based*. Task-based threading architectures are formed by binding one or more PEs to units of protocol functionality (such as presentation layer formatting, transport layer end-to-end flow control, and network layer fragmentation and reassembly). In these architectures, parallelism is achieved by executing protocol tasks in separate PEs, and passing data messages and control messages between the tasks/PEs. In contrast, message-based threading architectures are formed by binding the PEs to data messages and control messages received from applications and network interfaces. In these architectures, parallelism is achieved by simultaneously escorting multiple data messages and control messages on separate PEs through a stack of protocol tasks.

Protocol stacks (such as the TCP/IP protocol stack and the ISO OSI 7 layer protocol stack) may be implemented using either task-based or message-based threading architectures. However, the choice of threading architecture yields significantly different performance characteristics that vary across operating system and hardware platforms. For instance, on shared memory multi-processor platforms, task-based threading architectures often exhibit high context switching and data movement overhead due to scheduling and

[1]This research is supported in part by grants from the University of California MICRO program, Hughes Aircraft, Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, Tokyo Electric Power Company, Hewlett Packard (HP), and Siemens Medical Engineering.

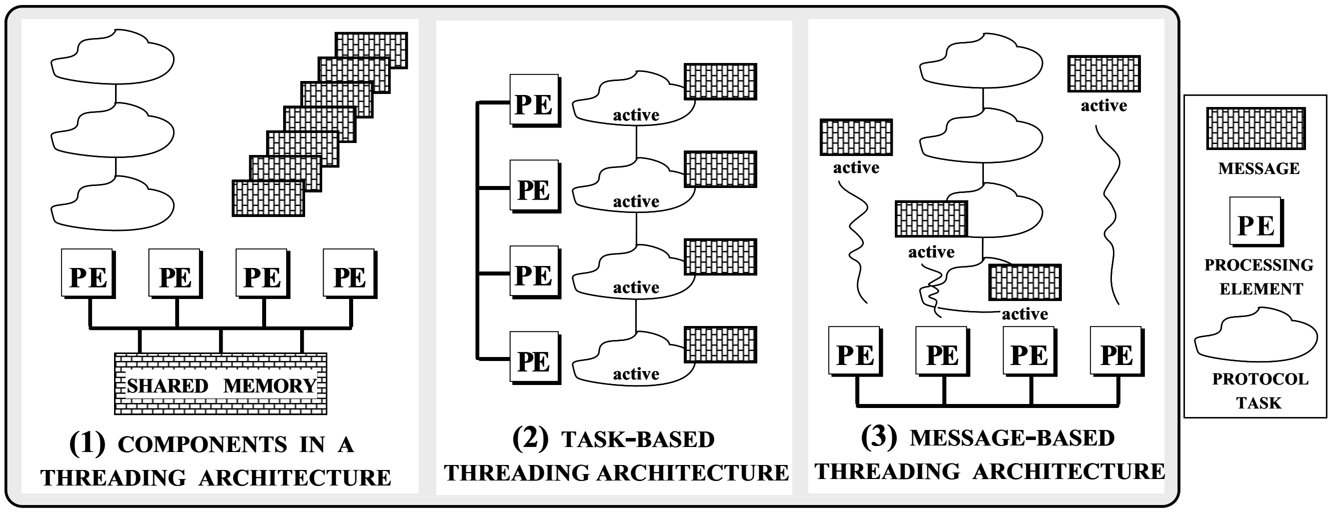
SHARED MEMORY
PE
(1) COMPONENTS IN A THREADING ARCHITECTURE
active
(2) TASK-BASED THREADING ARCHITECTURE
(3) MESSAGE-BASED THREADING ARCHITECTURE
MESSAGE
PROCESSING ELEMENT
PROTOCOL TASK

Figure 1: Threading Architecture Components and Interrelationships

caching properties of the OS and hardware [9]. In contrast, in a message-passing multi-processor environment, message-based threading architectures exhibit high levels of synchronization overhead due to high latency access to global resources such as shared memory, synchronization objects, or connection context information [6].

Prior work [1, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17] has generally selected a single task-based or message-based threading architecture and studied it in isolation. Moreover, earlier studies have been conducted on different OS and hardware platforms using different protocol stacks and implementation techniques. This diversity of threading architectures, platforms, and protocols makes it hard to compare results meaningfully.

In this paper, we describe the design and implementation of an object-oriented framework that enables controlled experiments with various threading architectures on multi-processor platforms. The framework controls for a number of relevant factors (such as protocol functionality, concurrency control strategies, application traffic characteristics, and network interfaces). By controlling these factors, our framework enables precise measurement of the performance impact of different threading architectures when parallelizing communication protocol stacks. This paper reports the results of systematic, empirical comparisons of the performance of several message-based and task-based threading architectures implemented on a 20 CPU shared memory multi-processor platform.

The paper is organized as follows: Section 2 outlines the two fundamental types of threading architectures and classifies related work accordingly; Section 3 describes the key threading and synchronization components in our object-oriented framework; Section 4 examines empirical results from experiments performed using the framework; Section 5 summarizes the results and outlines guidelines for using threading architectures effectively; and Section 6 presents concluding remarks.

2 Alternative Threading Architectures

Figure 1 (1) illustrates the three basic components of a threading architecture, which include:

- *Data messages and control messages* – which are sent and received from one or more applications and network devices;
- *Protocol tasks* – which are the units of protocol functionality that process the control messages and data messages;
- *Processing elements* (PEs) – which execute protocol tasks.

The two fundamental types of threading architectures structure these three basic components in the following ways:

1. Task-based threading architectures: which bind one or more PEs to protocol processing tasks (shown in Figure 1 (2)). In this architecture, tasks are the active entities, whereas messages processed by the tasks are the passive entities.

2. Message-based threading architectures: which bind the PEs to the control messages and the data messages received from applications and network interfaces (shown in Figure 1 (3)). In this architecture, messages are the active entities, whereas tasks that process the messages are the passive entities.

The remainder of this section outlines several alternative threading architectures in each of the two categories.

2.1 Examples of Task-based Threading Architectures

Task-based threading architectures associate threads[2] with

[2]In this paper, the term "thread" is used to refer to a series of instructions executing within an address space; this address space may be shared with other threads. Different terminology (such as lightweight processes [15, 18]) has also been used to denote the same basic concepts.

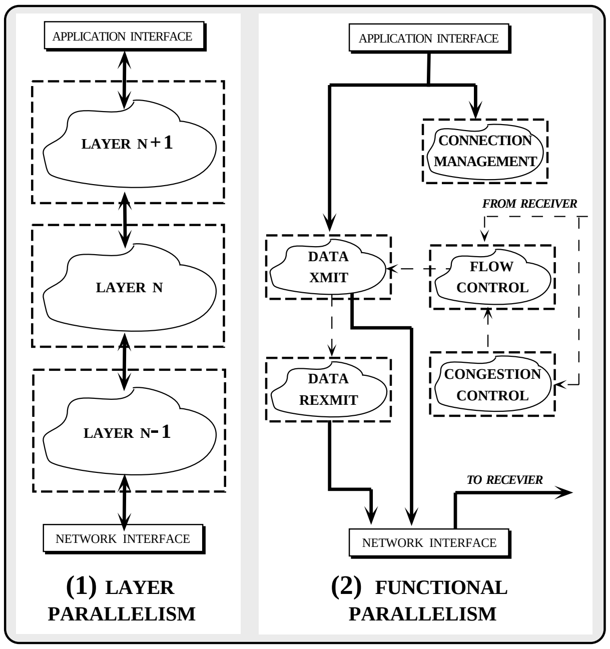

Figure 2: Task-based Threading Architecture Examples

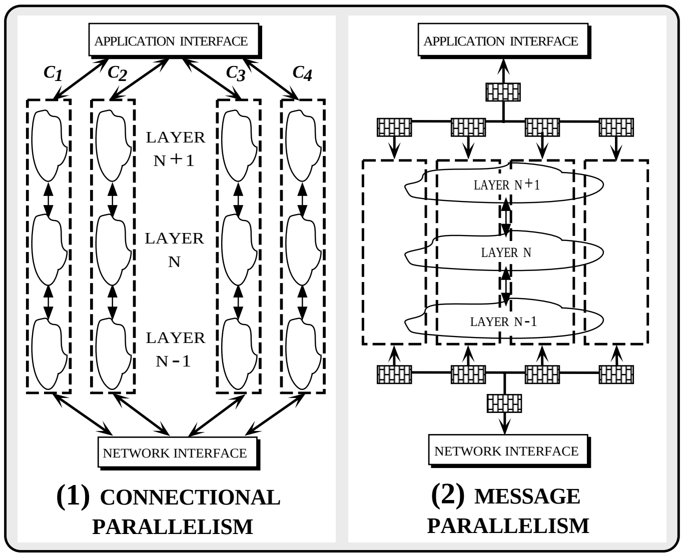

Figure 3: Message-based Threading Architecture Examples

clusters of one or more protocol tasks. Two common examples of task-based threading architectures are *Layer Parallelism* and *Functional Parallelism*. The primary difference between these two threading architectures involves the granularity of the protocol processing tasks. Protocol layers are more coarse-grained than protocol tasks because they cluster multiple tasks together to form a composite service (such as the end-to-end transport service defined by the ISO OSI transport layer).

As shown in Figure 2 (1), Layer Parallelism associates a separate thread with each layer (*e.g.*, the presentation, transport, and network layers) in a protocol stack. Certain protocol header and data fields in the messages may be processed in parallel as they flow through a pipeline of layers in a protocol stack. Buffering and flow control may be necessary within a protocol stack if processing activities in each layer execute at different rates.

Functional Parallelism associates a separate thread with various protocol tasks (such as header composition, acknowledgement, retransmission, segmentation, reassembly, and routing). As shown in Figure 2 (2), these protocol tasks execute in parallel and communicate by passing control messages and data messages to each other to coordinate parallel protocol processing.

2.2 Examples of Message-based Threading Architectures

Message-based threading architectures associate threads with messages, rather than with protocol layers or protocol tasks. Two common examples of message-based threading architectures are *Connectional Parallelism* and *Message Parallelism*. The primary difference between these threading architectures involves the point at which messages are demultiplexed onto a thread. Connectional Parallelism demultiplexes all messages bound for the same connection onto the same thread, whereas Message Parallelism demultiplexes messages onto an available thread.

Connectional Parallelism uses a separate thread to handle the messages associated with each open connection. As shown in Figure 3 (1), connections C_1, C_2, C_3, and C_4 reside in separate threads that execute a stack of protocol tasks on all messages associated with their respective connection. Within a connection, multiple protocol tasks are invoked sequentially on each message flowing through the protocol stack. Outgoing messages generally borrow the thread of control from an application and use it to escort messages down a protocol stack [18]. For incoming messages, a network interface or packet filter [19] typically performs demultiplexing operations to determine the connection/thread to associate with each message.

Message Parallelism associates a separate thread with every incoming or outgoing message. As illustrated in Figure 3 (2), a thread receives a message from an application or network interface and escorts that message through the protocol processing tasks in the protocol stack. As with Connectional Parallelism, outgoing messages typically borrow the thread of control from the application that initiated the message transfer.

2.3 Related Work

A number of studies have investigated the performance characteristics of task-based threading architectures that ran on either message passing or shared memory platforms. [9] measured the performance of several implementations of the transport and session layers in the ISO OSI reference model using an ADA-like rendezvous-style of Layer Parallelism in a nonuniform access shared memory multi-processor platform. [10] measured the performance of Functional Parallelism for presentation layer and transport layer functionality on a shared memory multi-processor platform. [11] measured the performance of a de-layered, function-oriented transport system [1] using Functional Parallelism on a message passing transputer multi-processor platform. An earlier study [6] measured the performance of the ISO OSI transport layer and network layer, also on a transputer platform. Likewise, [12] used a multi-processor transputer platform to parallelize several data-link layer protocols.

Other studies have investigated the performance characteristics of message-based threading architectures. All these studies utilized shared memory platforms. [13] measured the performance of the TCP, UDP, and IP protocols using Message Parallelism on a uniprocessor platform running the *x*-kernel. [5] measured the impact of synchronization on Message Parallelism implementations of TCP and UDP transport protocols built within a multi-processor version of the *x*-kernel. Likewise, [14] examined performance issues in parallelizing TCP-based and UDP-based protocol stacks using a different multi-processor version of the *x*-kernel. [15] measured the performance of the Nonet transport protocol on a multi-processor version of Plan 9 STREAMS developed using Message Parallelism. [7] measured the performance of the ISO OSI protocol stack, focusing primarily on the presentation and transport layers using Message Parallelism. [16] measured the performance of the TCP/IP protocol stack using Connectional Parallelism in a multi-processor version of System V STREAMS.

The results presented in this paper extend existing research in three ways. First, we measure the performance of several representative task-based and message-based threading architectures in a controlled environment. Second, our experiments report the impact of both context switching and synchronization overhead on communication subsystem performance. Third, in addition to measuring data link, network, and transport layer performance, our experiments also measure presentation layer performance. The presentation layer is widely considered to be a major bottleneck in high-performance communication subsystems [2].

3 A Framework for Experimenting with Threading Architectures

To facilitate controlled experiments with alternative threading architectures, we developed an object-oriented communication framework called the "ADAPTIVE Service eXecutive" (ASX). This framework is an integrated collection of components that provide an infrastructure for developing and testing protocol stacks within communication subsystems.

The ASX framework helps control for several key factors such as protocol functionality, concurrency control strategies, application traffic characteristics, and network interfaces. This enables precise measurement of the performance impact from using different threading architectures to parallelize protocol stacks in a multi-processor platform. For instance, in the experiments described in Section 4, the ASX framework is used to hold protocol functionality constant, while allowing the threading architecture to be systematically altered and measured in a controlled manner.

The ASX framework incorporates concepts from existing communication frameworks such as System V STREAMS [20], the *x*-kernel [13], and the Conduit [21]. These frameworks support the flexible configuration of communication subsystems built from reusable protocol components (such as message managers, timer-based event dispatchers, and connection demultiplexers [13] and other reusable protocol mechanisms [22]). In addition to supplying building-block protocol components, the ASX framework also extends the features provided by existing communication frameworks. In particular, ASX components decouple protocol-specific functionality from the following structural and behavioral characteristics of a communication subsystem:

- *The type of locking mechanisms used to synchronize access to shared resources*;
- *The selection of message-based and task-based threading architectures*;
- *The use of kernel-level vs. user-level threads*.

The ASX framework components described below simplify development of, and experimentation with, protocol stacks that are functionally equivalent, but possess significantly different threading architectures.

The overall object-oriented design and implementation of the ASX framework has been described elsewhere [23]. The remainder of this section focuses on the threading and synchronization components in ASX. The goal is to illustrate how the ASX framework simplifies the development and testing of multi-threaded protocol stacks on multi-processor platforms.

3.1 ASX Components for Flexibly Composing Layer Protocol Stacks

The ASX framework coordinates the installation-time and/or run-time configuration of one or more *Streams*. A Stream is a composite object used to configure and execute protocol stacks in the ASX framework. As illustrated in Figure 4, a Stream contains a series of interconnected `Module` objects.[3] `Module` objects are used to decompose a protocol stack into functionally distinct levels. Each level implements a cluster of related protocol tasks (such as an end-to-end transport service or a presentation layer service).

[3]Throughout the paper, components in the ASX framework are illustrated with Booch notation [24]. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate a link exists between two objects. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a small circle at one end indicates either a composition or a uses relation between two classes. Solid rectangles indicate class categories, which combine a number of related classes into a common name space.

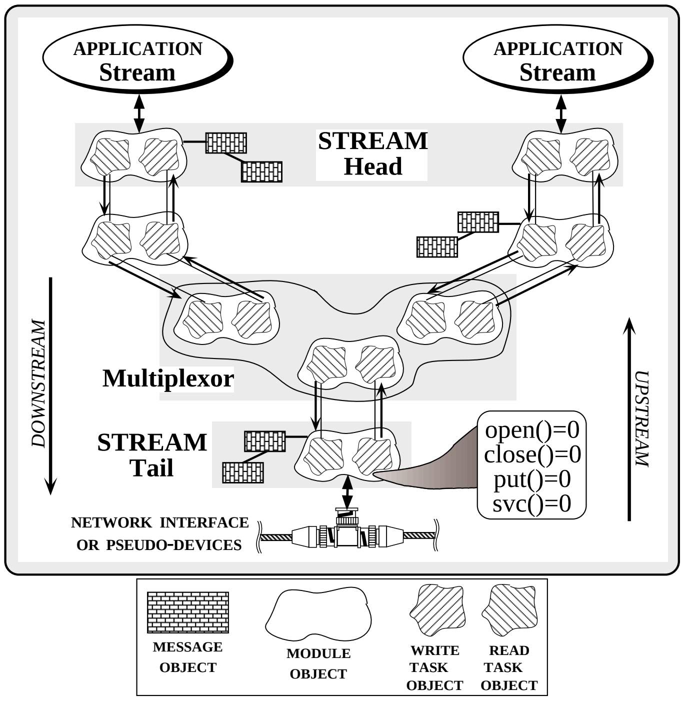

Figure 4: Components in the ASX Framework

Any level that requires multiplexing and/or demultiplexing of messages between one or more related Streams may be developed using a `Multiplexor` object. A `Multiplexor` is a container that provides mechanisms to route messages between `Modules` in adjacent Streams. Both `Module` and `Multiplexor` objects may be flexibly configured into a Stream by developers at installation-time, as well as by applications at run-time.

Every `Module` contains a pair of `Task` objects that partition a level into the read-side and write-side functionality required to implement a particular protocol task or layer. A `Task` can be specialized to target a specific domain (such as the domain of communication protocol stacks or the domain of network management applications [23]). Each `Task` contains a `Message_Queue`, which is a reusable ASX component that buffers data messages and control messages for subsequent processing. Protocol tasks in adjacent `Modules` communicate by exchanging typed messages via a uniform message passing interface defined by the `Task` class.

The `Task` class defines an interface that subclasses inherit and selectively override to provide the read-side and write-side protocol functionality in a `Module`. The `Task` class is *abstract* since it defines pure virtual methods (such as `put` and `svc`). Pure virtual methods decouple the protocol-independent components (such as message objects, message lists, and message demultiplexing mechanisms) provided by the ASX from the protocol-specific subclasses (such as those implementing the data-link, IP, TCP, UDP, and XDR protocols) that inherit and use these components. This decoupling enhances component reuse and simplifies development and configuration of Stream-based protocol stacks.

3.2 ASX Components for Flexibly Synchronizing Multi-Threaded Protocol Stacks

The ASX framework provides a set of reusable components that are responsible for spawning, executing, synchronizing, controlling, and gracefully terminating services via one or more threads at run-time [25]. These threads of control execute protocol tasks and pass messages between `Tasks` in a protocol stack. The following outlines the synchronization components in ASX and illustrates how they can be configured to create highly flexible threading architectures.

3.2.1 Low-level ASX Synchronization Mechanisms

The two basic types of synchronization mechanisms provided by the ASX framework are the `Mutex` and `Condition` objects:

• **ASX Mutex objects:** These objects ensure the integrity of a shared resource that may be accessed concurrently by multiple threads. A `Mutex` object serializes the execution of multiple threads by defining a critical section where only one thread of control may execute its code at a time. To enter a critical section, a thread invokes the `Mutex::acquire` method. When a thread leaves its critical section, it invokes the `Mutex::release` method.

The `Mutex` methods are implemented via adaptive spin-locks that ensure mutual exclusion by using an atomic hardware instruction. An adaptive spin-lock polls a designated memory location using the atomic hardware instruction until either (1) the value at this location is changed by the thread that currently owns the lock (this signifies that the lock has been released and may now be acquired by the spinning thread) or (2) the thread that is holding the lock goes to sleep (at this point, the spinning thread also puts itself to sleep to avoid unnecessary polling [26]).

• **ASX Condition objects:** These objects allow one or more cooperating threads to suspend their execution until an expression involving shared data attains a particular state. Unlike the adaptive spin-lock `Mutex` objects, a `Condition` object enables a thread to suspend itself indefinitely (via the `Condition::wait` method) until an expression involving shared data attains a particular state. When another cooperating thread indicates that the state of the shared data has changed (by invoking the `Condition::signal` method), the associated `Condition` object wakes up a thread that is suspended on that `Condition` object. The newly awakened thread then re-evaluates its expression and potentially resumes processing if the shared data has attained an appropriate state.

`Condition` object synchronization is not implemented using spin-locks, which consume excessive resources if a thread must wait an indefinite amount of time for a particular condition to become signaled. Instead, `Condition` objects are implemented with sleep-locks that trigger a context switch to allow another thread to execute. Section 4 illustrates the

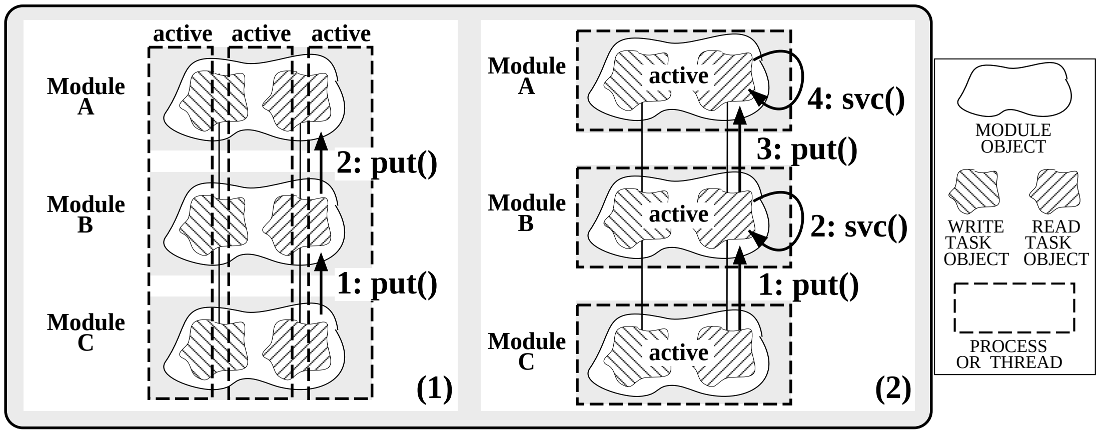

Figure 5: Alternative Methods for Invoking `put` and `svc` Methods

consequences of context switching and synchronization on threading architecture performance.

3.2.2 Higher-level ASX Threading Architecture Components

The low-level ASX `Mutex` and `Condition` objects form the basic building blocks for configuring higher-level message-based and task-based threading architectures. A distinguishing key feature of these threading architectures is how they use synchronous and asynchronous processing. The following discusses how the ASX framework allows both forms of processing to be supported uniformly. Section 4 then illustrates the performance impact of using synchronous and asynchronous processing to parallelize communication protocol stacks on a shared memory multi-processor platform.

• **Synchronous execution for message-based threading architectures:** Message-based threading architectures perform most of their protocol-specific processing *synchronously* with respect to other `Tasks` in a Stream. For instance, the ASX-based implementation of the Message Parallelism threading architecture associates a separate thread of control with each message arriving at a network interface. This thread is obtained from a pool of pre-initialized threads (which are labeled as the active entities in Figure 5 (1)). The incoming message passes through a series of interconnected `Tasks` in a Stream.

In a message-based threading architecture, each `Task` in a Stream passes messages to adjacent levels in a protocol stack by invoking a synchronous calls to the `put` method in neighboring `Tasks`. The `put` method runs *synchronously* with respect to its caller by borrowing the thread of control from the `Task`.

• **Asynchronous execution for task-based threading architectures:** Task-based threading architectures perform most of their processing *asynchronously* with respect to other `Tasks` in the Stream. For instance, each protocol layer in the ASX-based Layer Parallelism threading architecture is implemented in a separate `Module` object, which is associated with a separate thread. Messages arriving from a network interface are passed between `Tasks` running in the separate threads (which are labeled as the active entities in Figure 5 (2)).

In a task-based threading architecture, each `Task` in a Stream executes its protocol-specific functionality in its `svc` method within a separate thread. Thus, the `svc` methods perform protocol-specific processing asynchronously. Each `svc` method runs an event loop that waits continuously for messages to be inserted into the `Task`'s `Message_Queue`. When messages are inserted into a `Message_Queue`, the `svc` method dequeues the messages and performs the protocol-specific processing tasks defined by the `Task` subclass.

3.2.3 Transparently Parameterizing Threading Architecture Synchronization Policies

The ASX components (such as `Tasks`, `Modules`, and Streams) for composing protocol stacks described in Section 3.1 provide the minimal amount of locking necessary to avoid race conditions. This minimalistic synchronization strategy avoids over-constraining the granularity of a threading architecture's concurrency control policies. Therefore, protocol developers have greater flexibility in configuring threading architectures that map onto the underlying hardware and software platforms efficiently.

The ASX framework enables different threading architectures to configure their synchronization policies flexibly by instrumenting protocol tasks with various combinations of `Mutex` and `Condition` objects. When used in conjunction with higher-level ASX framework components (such as `Tasks`) and C++ language features (such as parameterized types), these synchronization objects help to decouple protocol processing functionality from the synchronization policies used by a particular threading architecture. The following example illustrates how ASX framework synchronization objects can be transparently parameterized into communication protocol software.

In the threading architecture experiments described in Section 4, the `Map_Manager` class is used to demultiplex incoming network messages to the appropriate `Module` [27]. `Map_Manager` is a template class that is parameterized by an external identifier (`EXT_ID`), an internal identifier (`INT_ID`), and a mutual exclusion mechanism (`LOCK`), as follows:

```
template <class EXT_ID, class INT_ID, class LOCK>
class Map_Manager
{
public:
  // Associate EXT_ID with INT_ID
  bool bind (EXT_ID, INT_ID *);
  // Break an association
  bool unbind (EXT_ID);
  // Locate INT_ID corresponding to EXT_ID
  bool find (EXT_ID, INT_ID &);
  // ...

private:
  // Parameterized synchronization object
  LOCK lock;
  // Perform the lookup
  bool locate_entry (EXT_ID, INT_ID &);
};
```

The `find` method of the `Map_Manager` template class is implemented using the technique illustrated in the code below (the `bind` and `unbind` methods are implemented in a similar manner):

```
template <class EXT_ID, class INT_ID, class LOCK>
bool Map_Manager<EXT_ID, INT_ID, LOCK>
::find (EXT_ID ext_id, INT_ID &int_id)
{
  // Acquire lock in mon constructor
  Guard<LOCK> mon (this->lock);

  if (this->locate_entry (ext_id, int_id))
    return true;
  else
    return false;
  // Release lock in mon destructor
}
```

The code shown above uses the constructor of the `Guard` class to acquire the `Map_Manager` lock when an object of the class is created. Likewise, when the `find` method returns, the destructor for the `Monitor` object releases the `Mutex` lock. Note that the `Mutex` lock is released automatically, regardless of which arm in the `if/else` statement returns from the `find` method. Moreover, the lock is released even if an exception is raised within the body of the `find` method.

The experiments described in Section 4 implement connection demultiplexing operations in the connection-oriented protocol stacks via the `Map_Manager` template class. In the experiments, the `Map_Manager` class is instantiated with a `LOCK` parameter whose type is determined by the threading architecture being configured. For instance, the `Map_Manager` used by the Message Parallelism implementation of the transport layer in the protocol stack from in Section 4.3.1 is instantiated with the following `EXT_ID`, `INT_ID`, and `LOCK` type parameters:

```
typedef Map_Manager
        <TCP_Addr, // Segment address.
         TCB, // Internal control block.
         RW_Mutex> // Readers/writer lock.
        ADDR_MAP;
```

This particular instantiation of `Map_Manager` locates the transport control block (`TCB`), which is an internal data structure that is associated with the address of an incoming `TCP` message (`TCP_Addr`), which is the external identifier. Instantiating the `Map_Manager` class with the `RW_Mutex` class ensures that its `find` method executes as a critical section. The `RW_Mutex` is the `ASX`-implementation of a readers/writer lock that efficiently prevents race conditions from occurring with other threads that are inspecting or inserting entries into the `Map_Manager` in parallel.

In contrast, the Layer Parallelism implementation of the transport layer in the protocol stack described in Section 4.3.2 uses a different type of concurrency control. In this case, serialization is performed at the transport layer using the synchronization mechanisms provided by the `Message_Queue` defined in the `Task` class. Therefore, the `Map_Manager` used for the Layer Parallelism implementation of the protocol stack is instantiated with a different `Mutex` class, as follows:

```
typedef Map_Manager
        <TCP_Addr, // Segment address.
         TCB, // Internal control block.
         Null_Mutex> // "Null" lock.
        ADDR_MAP;
```

The implementation of the `acquire` and `release` methods in the `Null_Mutex` class are "no-op" inline functions that are removed completely by the compiler optimizer. In general, templates generate efficient object code that exacts no additional run-time overhead for the increased flexibility.

The definition of the `Map_Manager` address map may be conditionally compiled using template class arguments corresponding to the type of threading architecture that is required, *i.e.*:

```
#if defined (MSG_BASED_PA)
// Select a message-based threading architecture.
typedef Mutex LOCK;
#elif defined (TASK_BASED_PA)
// Select a task-based threading architecture.
typedef Null_Mutex LOCK;
#endif

typedef Map_Manager<TCP_Addr, TCB, LOCK> ADDR_MAP;
```

As shown below, this allows the majority of the protocol code to remain unaffected, regardless of the choice of threading architecture, as follows:

```
ADDR_MAP addr_map;
TCP_Addr tcp_addr;
TCP      tcb;

// ...

if (addr_map.find (tcp_addr, tcb))
  // Perform connection-oriented processing
```

4 Communication Subsystem Performance Experiments

This section presents performance results obtained by measuring the data reception portion of protocol stacks implemented using several different threading architectures.

Two different types of protocol stacks were implemented: *connection-oriented* and *connectionless*. Three different variants of task-based and message-based threading architectures were used to parallelize the protocol stacks: *Layer Parallelism* (which is a task-based threading architecture), as well as *Message-Parallelism* and *Connectional Parallelism* (which are message-based threading architectures). The following section describes the multi-processor platform and the measurement tools used in the experiments, the communication protocol stacks and threading architectures developed using `ASX` framework components, and the performance results.

4.1 Multi-Processor Platform

All experiments were conducted on an otherwise idle Sun SPARCcenter 2000 shared memory symmetric multi-processor. This SPARCcenter platform contained 640 Mbytes of RAM and 20 superscalar SPARC 40 MHz processing elements (PEs). The operating system used for the experiments was release 5.4 of SunOS (also known as Solaris 2.4). SunOS 5.4 provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel on the SPARCcenter platform [26]. All the threading architectures in the experiments execute protocol tasks using separate SunOS unbound threads. These unbound threads are multiplexed over $1, 2, 3, \ldots 20$ SunOS lightweight processes (LWPs) within an OS process. The SunOS 5.4 scheduler maps each LWP directly onto a separate kernel thread. Since kernel threads are the units of PE scheduling and execution in SunOS, multiple LWPs run protocol tasks in parallel on the 20 PEs of the SPARCcenter 2000.

Our tests measured the memory bandwidth of the SPARCcenter 2000 platform to be approximately 750 Mbits/sec. In addition to memory bandwidth, communication subsystem throughput is significantly affected by the context switching and synchronization overhead of the multi-processor platform. Scheduling and synchronizing a SunOS LWP requires a kernel-level context switch. This context switch flushes register windows and updates instruction and data caches, instruction pipelines, and translation lookaside buffers [4]. Measurements of these activities indicated that it takes approximately 50 μsecs to perform between LWPs running in the same process. During this time, the PE incurring the context switch does not execute any protocol tasks.

`ASX` `Mutex` and `Condition` objects were both used in the experiments. `Mutex` objects were implemented using SunOS adaptive spin-locks and `Condition` objects were implemented using SunOS sleep-locks [26]. Synchronization methods invoked on `Condition` objects were approximately two orders of magnitude more expensive compared with methods on `Mutex` objects. For instance, measurements indicated that approximately 4 μsecs were required to acquire or release a `Mutex` object when no other PEs contended for the lock. In contrast, when all 20 PEs contended for a `Mutex` object, the time required to perform the locking methods increased to approximately 55 μsecs.

Approximately 300 μsecs were required to synchronize LWPs using `Condition` objects when no other PEs contended for the lock. Conversely, when all 20 PEs contended for a `Condition` object, the time required to perform the locking methods increased to approximately 520 μsecs. The two orders of magnitude difference in performance between `Mutex` and `Condition` objects was caused by the implementation of the `Condition` object methods. In particular, performing the `wait` method on a `Condition` object incurred a context switch, which increased the synchronization overhead. In contrast, performing an `acquire` method on a `Mutex` object rarely triggered a context switch since the `Mutex` was implemented with an adaptive spin-lock.

4.2 Functionality of the Communication Protocol Stacks

Two types of protocol stacks were investigated in the experiments. One was based on the connectionless UDP transport protocol; the other was based on the connection-oriented TCP transport protocol. Both protocol stacks contained data-link, network, transport, and presentation layers. The presentation layer was included in the experiments since it represents a major bottleneck in high-performance communication subsystems [2, 7, 28].

Both the connectionless and connection-oriented protocol stacks were developed by specializing reusable components in the `ASX` framework via inheritance and parameterized types. As discussed in Section 3.2, inheritance and parameterized types were used to hold protocol stack functionality constant, while the threading architecture was systematically varied. Each layer in a protocol stack was implemented as a `Module`, whose read-side and write-side inherit interfaces and implementations from the `Task` abstract class described in Section 3.1. The synchronization and demultiplexing mechanisms required to implement different threading architectures were parameterized using C++ template class arguments. As illustrated in Section 3.2, these templates were instantiated based upon the type of threading architecture being tested.

Data-link layer processing in each protocol stack was performed by the `DLP Module`. This `Module` transformed network packets received from a network interface into the canonical message format used internally by the interconnected `Task` components in a Stream. Preliminary tests conducted with the widely-available `ttcp` benchmarking tool indicated that the SPARCcenter multi-processor platform processed messages through a protocol stack much faster than our 10 Mbps Ethernet network interface was capable of handling. Therefore, the network interface in our threading architecture experiments was simulated with a single-copy pseudo-device driver operating in loop-back mode. Our approach is consistent with those used in similar experiments on multi-processor platforms [5, 7, 14].

The network and transport layers of the protocol stacks were based on the IP, UDP, and TCP implementations in the

BSD 4.3 Reno release [29]. The 4.3 Reno TCP implementation contains the TCP header prediction enhancements, as well as the TCP slow start algorithm and congestion avoidance features [30]. The UDP and TCP transport protocols were configured into the ASX framework via the UDP and TCP Modules. Network layer processing was performed by the IP Module. This Module handled routing and segmentation/reassembly of Internet Protocol (IP) packets.

Presentation layer functionality was implemented in the XDR Module using marshaling routines produced by the ONC eXternal Data Representation (XDR) stub generator. The ONC XDR stub generator translates type specifications into marshaling routines. These marshaling routines encode/decode implicitly-typed messages before/after exchanging them among hosts that may possess heterogeneous processor byte-orders.

The ONC presentation layer conversion mechanisms consist of a type specification language (XDR) and a set of library routines that implement the appropriate encoding and decoding rules for built-in integral types (*e.g.,* char, short, int, and long), as well as real types (*e.g.,* float and double). These library routines may be combined to produce marshaling routines for arbitrarily complex user-defined composite types (such as record/structures, unions, arrays, and pointers). Messages exchanged via XDR are implicitly-typed, which improves marshaling performance at the expense of run-time flexibility.

The XDR routines generated for the connectionless and connection-oriented protocol stacks converted incoming and outgoing messages into and from variable-sized arrays of structures containing a set of integral and real values. The XDR processing involved byte-order conversions, as well as dynamic memory allocation and deallocation.

4.3 Structure of the Threading Architectures

The remainder of this section outlines the structure of the message-based and task-based threading architectures used to parallelize the connectionless and connection-oriented protocol stacks described above.

4.3.1 Structure of the Message-based Threading Architectures

• **Connectional Parallelism:** The protocol stack depicted in Figure 6 (1) illustrates an ASX-based implementation of the Connectional Parallelism (CP) threading architecture outlined in Section 2.2. Each thread performs the data-link, network, transport, and presentation layer tasks sequentially for a single connection. Protocol tasks are divided into four interconnected Modules, corresponding to the data-link, network, transport, and presentation layers. Data-link processing is performed in the CP_DLP Module. The Connectional Parallelism implementation of this Module performs "eager demultiplexing" via a packet filter [19] at the data-link layer. Thus, the CP_DLP Module uses its read-side svc method to

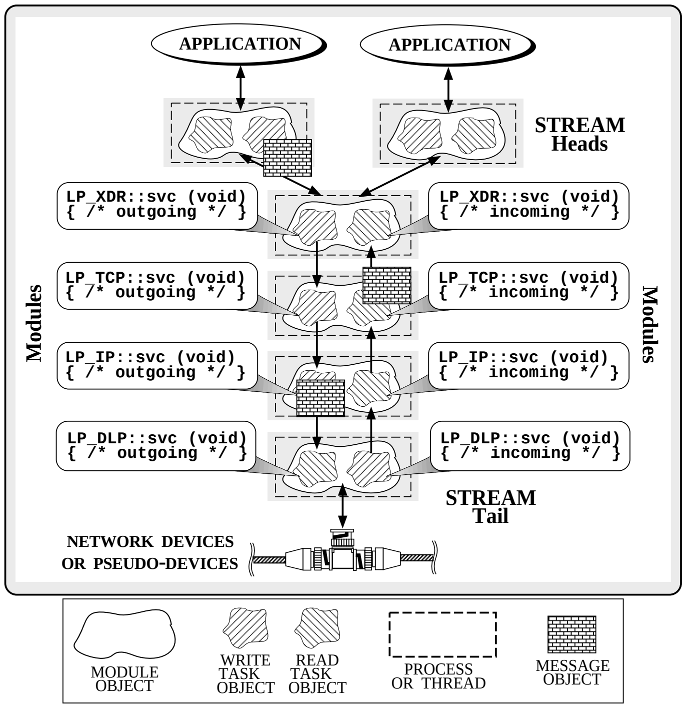
APPLICATION
APPLICATION
STREAM Heads
LP_XDR::svc (void) { /* outgoing */ }
LP_XDR::svc (void) { /* incoming */ }
LP_TCP::svc (void) { /* outgoing */ }
LP_TCP::svc (void) { /* incoming */ }
LP_IP::svc (void) { /* outgoing */ }
LP_IP::svc (void) { /* incoming */ }
LP_DLP::svc (void) { /* outgoing */ }
LP_DLP::svc (void) { /* incoming */ }
Modules
Modules
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
MODULE OBJECT
WRITE TASK OBJECT
READ TASK OBJECT
PROCESS OR THREAD
MESSAGE OBJECT

Figure 7: Layer Parallelism

demultiplex incoming messages onto the appropriate transport layer connection. In contrast, the CP_IP, CP_TCP, and CP_XDR Modules perform their processing synchronously in their respective put methods. To eliminate extraneous data movement overhead, the Task::put_next method passes a pointer to a message between protocol layers.

• **Message Parallelism:** Figure 6 (2) depicts the Message Parallelism (MP) threading architecture used for the TCP-based connection-oriented protocol stack. When an incoming message arrives, it is handled by the MP_DLP::svc method. This method manages a pool of pre-spawned SunOS unbound threads. Each message is associated with an unbound thread that escorts the message synchronously through a series of interconnected Tasks that form a protocol stack. Each layer of the protocol stack performs the protocol tasks defined by its Task. When these tasks are complete, an upcall [31] is used to pass the message to the next adjacent layer in the protocol stack. The upcall is performed by invoking the put method in the adjacent layer's Task. This put method borrows the thread of control from its caller and executes the protocol tasks associated with its layer.

The Message Parallelism threading architecture for the connectionless protocol stack is similar to the one used to implement the connection-oriented protocol stack. The primary difference between the two protocol stacks is that the connectionless stack performs UDP transport functionality, which is less complex than TCP. For example, UDP does not generate acknowledgements, keep track of round-trip time estimates, or manage congestion windows. In addition, the

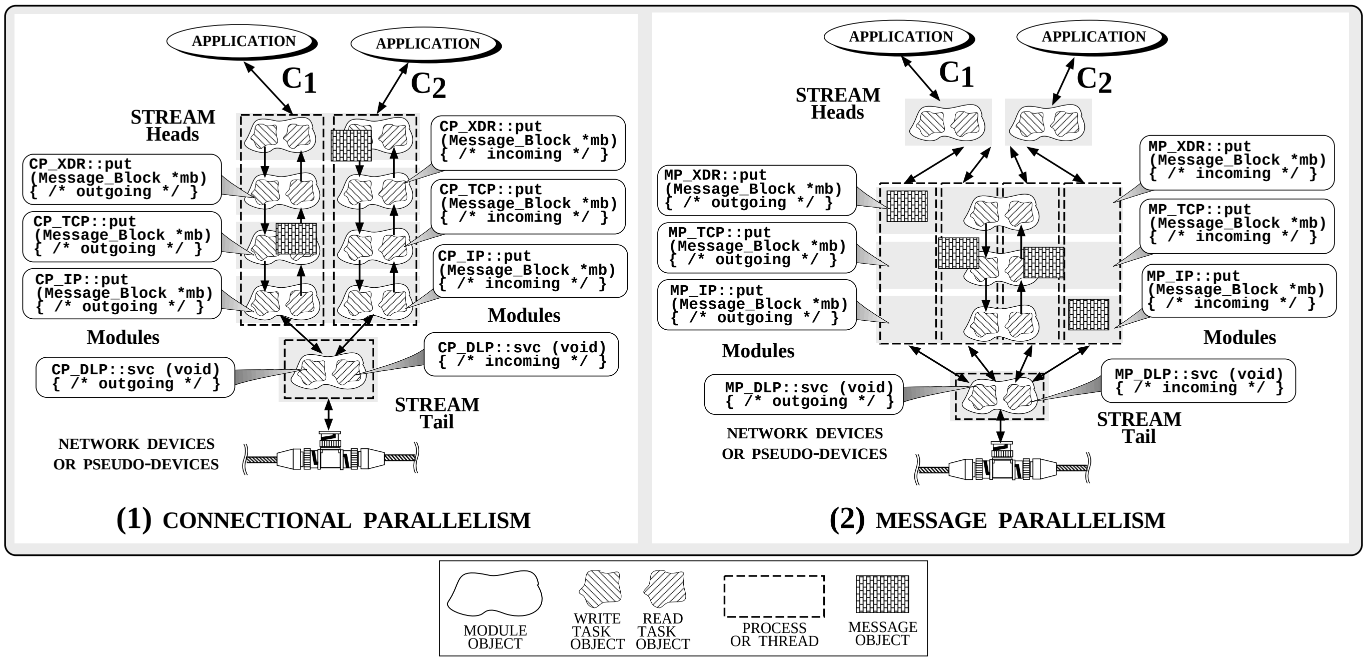
APPLICATION
APPLICATION
C1
C2
STREAM Heads
CP_XDR::put (Message_Block *mb) { /* outgoing */ }
CP_TCP::put (Message_Block *mb) { /* outgoing */ }
CP_IP::put (Message_Block *mb) { /* outgoing */ }
CP_XDR::put (Message_Block *mb) { /* incoming */ }
CP_TCP::put (Message_Block *mb) { /* incoming */ }
CP_IP::put (Message_Block *mb) { /* incoming */ }
Modules
CP_DLP::svc (void) { /* outgoing */ }
CP_DLP::svc (void) { /* incoming */ }
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
(1) CONNECTIONAL PARALLELISM
MP_XDR::put (Message_Block *mb) { /* outgoing */ }
MP_TCP::put (Message_Block *mb) { /* outgoing */ }
MP_IP::put (Message_Block *mb) { /* outgoing */ }
MP_XDR::put (Message_Block *mb) { /* incoming */ }
MP_TCP::put (Message_Block *mb) { /* incoming */ }
MP_IP::put (Message_Block *mb) { /* incoming */ }
MP_DLP::svc (void) { /* outgoing */ }
MP_DLP::svc (void) { /* incoming */ }
(2) MESSAGE PARALLELISM
MODULE OBJECT
WRITE TASK OBJECT
READ TASK OBJECT
PROCESS OR THREAD
MESSAGE OBJECT

Figure 6: Message-based Threading Architectures

connectionless MP_UDP::put method handles each message concurrently and independently, without explicitly preserving inter-message ordering.

In contrast, the connection-oriented MP_TCP::put method utilizes several Mutex synchronization objects. For instance, as separate messages from the same connection ascend the protocol stack in parallel, these Mutex objects serialize access to per-connection control blocks. Serialization is required to protect shared resources (such as message queues, protocol connection records, TCP segment reassembly, and demultiplexing tables) against race conditions.

Both Connectional Parallelism and Message Parallelism optimize message management by using SunOS thread-specific storage [26] to buffer messages as they flow through a protocol stack. This optimization leverages off the cache affinity [17] properties of the SunOS shared memory multiprocessor. In addition, it minimizes the cost of synchronization operations used to manage the global dynamic memory heap.

4.3.2 Structure of the Task-based Threading Architecture

• **Layer Parallelism:** Figure 7 illustrates the ASX framework components that implement a Layer Parallelism (LP) threading architecture for the TCP-based connection-oriented protocol stack. The connectionless UDP-based protocol stack for Layer Parallelism was designed in a similar manner. The primary difference between them was that the read-side and write-side Tasks in the connectionless transport layer Module (LP_UDP) implement the simpler UDP functionality.

Protocol-specific processing at each protocol layer shown in Figure 7 is performed in the Task::svc method. Each svc method is executed in a separate thread associated with the Module that implements the corresponding protocol layer (*e.g.,* LP_XDR, LP_TCP, LP_IP, and LP_DLP). These threads cooperate in a producer/consumer manner, operating in parallel on message header and data fields corresponding to their particular protocol layer. Every svc method performs its protocol layer tasks before passing a message to an adjacent Module running in a separate thread.

All threads share a common address space, which eliminates the need to copy messages that are passed between adjacent Modules. However, even if pointers to messages are passed between threads, per-PE data caches may be invalidated by hardware cache consistency protocols. Cache invalidation degrades performance by increasing the level of contention on the SPARCcenter system bus. Moreover, since messages are passed between PEs, it is not feasible to use the thread-specific storage memory management optimization techniques described in Section 4.3.1.

A strict implementation of Layer Parallelism would limit parallel processing to only include the number of protocol layers that could run on separate PEs. On a platform with 2 to 4 PEs this is not a serious problem since the protocol stacks used in the experiments only had 4 layers. On the 20 PE SPARCcenter platform, however, this approach would have greatly constrained the ability of Layer Parallelism to utilize the available processing resources. To alleviate this constraint, the connection-oriented Layer Parallelism threading architecture was implemented to handle a cluster of connections (*i.e.,* 5 connections per 4 layer protocol stack, with one PE per-layer). Likewise, the connectionless Layer Parallelism threading architecture was partitioned across 5 network interfaces to utilize the available parallelism.

4.4 Measurement Results

This section presents results obtained by measuring the data reception portion of the protocol stacks developed using the threading architectures described in Section 4.3. Three types of measurements were obtained for each combination of threading architecture and protocol stack: *average throughput*, *context switching overhead*, and *synchronization overhead*. Average throughput is a blackbox metric that measures the impact of parallelism on protocol stack performance. In contrast, context switching and synchronization are white-box metrics that help explain the variation in the throughput measurements.

Average throughput was measured by holding the protocol functionality, application traffic, and network interfaces constant, while systematically varying the threading architecture in order to determine the impact on performance. Each benchmarking run measured the amount of time required to process 20,000 4 Kbyte messages. In addition, 10,000 4 Kbyte messages were transmitted through the protocol stacks at the start of each run to ensure that all the PE caches were fully initialized (the time required to process these initial 10,000 messages was not used to calculate the average throughput). Each test was run using $1, 2, 3, \ldots 20$ PEs, with each test replicated a dozen times and the results averaged. The purpose of replicating the tests was to insure that the amount of interference from process and thread management tasks internal to the OS did not perturb the results.

Various statistics were collected using an extended version of the widely available `ttcp` protocol benchmarking tool [28]. The `ttcp` tool measures the amount of OS processing resources, user-time, and system-time required to transfer data between a transmitter thread and a receiver thread. The flow of data is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters (such as the number of data buffers transmitted and the size of data buffers and protocol windows) may be selected at run-time.

The version of `ttcp` used in our experiments was modified to use ASX-based connection-oriented and connectionless protocol stacks. These protocol stacks were configured in accordance with the threading architectures described in Section 4.3. The `ttcp` tool was also enhanced to allow a user-specified number of connections to be active simultaneously. This extension enabled us to measure the impact of multiple connections on the performance of the connection-oriented protocol stacks using message-based and task-based threading architectures.

4.4.1 Throughput Measurements

• **Connection-oriented Performance:** Figures 8, 9, and 10 depict the average throughput for the message-based threading architectures (Connectional Parallelism and Message Parallelism) and the task-based threading architecture (Layer Parallelism) used to implement the connection-oriented protocol stacks. Each test run for these connection-oriented threading architectures used 20 connections. These figures report the average throughput (in Mbits/sec), measured both with and without presentation layer processing. The figures illustrate how throughput is affected as the number of PEs increase from 1 to 20.

Figures 11, 12, and 13 indicate the relative speedup that resulted from successively adding another PE to each threading architecture. Relative speedup is computed by dividing the average aggregated throughput for n PEs (shown in Figures 8, 9, and 10, where $1 \leq n \leq 20$) by the average throughput for 1 PE.

The results from Figures 8, 9, 11, and 12 indicate that increasing the number of PEs generally improves the average throughput in the message-based threading architectures. Connection-oriented Connectional Parallelism exhibited the highest performance, both in terms of average throughput and in terms of relative speedup. As shown in Sections 4.4.2 and 4.4.3, these results stem from the low context switching and synchronization overhead of Connectional Parallelism, relative to the other threading architectures.

The following paragraphs examine the results for each threading architecture in detail:

- *Connectional Parallelism* – As shown in Figure 8, the average throughput of Connectional parallelism with presentation layer processing peaks at approximately 100 Mbits/sec. The average throughput without presentation layer processing peaks at just under 370 Mbits/sec. These results indicate that the presentation layer represents a significant portion of the overall protocol stack overhead. As shown in Figure 11, the relative speedup of Connectional Parallelism without presentation layer processing increases steadily from 1 to 20 PEs. The relative speedup with presentation layer processing is similar up to 12 PEs, at which point it begins to level off. This speedup curve flattens due to the additional overhead from data movement and synchronization performed in the presentation layer.

- *Message Parallelism* – As shown in Figure 9, the average throughput achieved by connection-oriented Message Parallelism without presentation layer processing peaks at just under 130 Mbits/sec. When presentation layer processing is performed, the average throughput is 1.5 to 3 times lower, peaking at approximately 90 Mbits/sec. Note, however, that the relative speedup without presentation layer processing (shown in Figure 12) flattens out after 8 CPUs. This flattening occurs for two reasons. First, there is increased contention for shared synchronization objects at the transport layer (discussed further in Section 4.4.3). Second, there is also increased contention for the shared memory bus [5], which ultimately limits the performance of protocol parallelism.

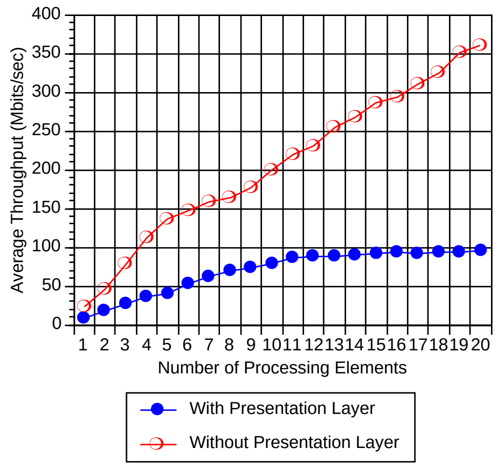

Figure 8: Connection-oriented Connectional Parallelism Throughput

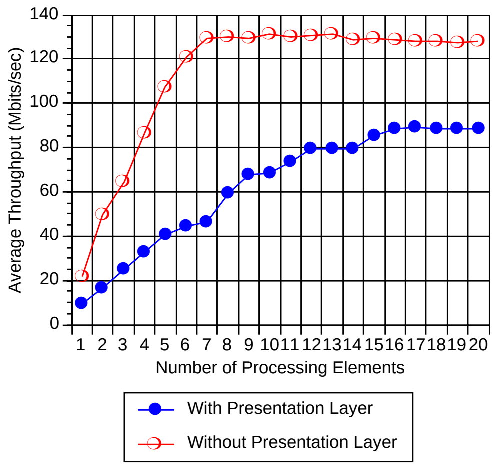

Figure 9: Connection-oriented Message Parallelism Throughput

In contrast, the relative speedup of connection-oriented Message Parallelism with presentation layer processing grows steadily from 1 to 20 PEs. This behavior indicates that connection-oriented Message Parallelism benefits more from parallelism when the protocol stack contains presentation layer processing (which is consistent with findings reported in [14]). The reason for this behavior is that presentation layer processing involves very little synchronization overhead compared with the connection-oriented transport layer. Therefore, the protocol stacks have more opportunity to exploit the parallelism available from the 20 PEs.

- *Layer Parallelism* – In contrast to Connectional Parallelism and Message Parallelism, the performance of the connection-oriented Layer Parallelism (shown in Figures 10 and 13) did not scale up as the number of PEs increased. The average throughput with presentation layer processing (shown in Figure 10) peaks at approximately 36 Mbits/sec. This amount is well below half of the throughput achieved by Connectional Parallelism and Message Parallelism. The throughput exhibited by Layer Parallelism peaks at 40 Mbits/sec when presentation layer processing is omitted. This is over 3 times lower than Message Parallelism and approximately 9 times lower than Connectional Parallelism. As shown in Figure 13, the relative speedup both with and without presentation layer processing increases until after 10 and 7 PEs, respectively. After peaking, average throughput levels off and gradually begins to decrease.

  The poor overall performance of Layer Parallelism stems from its high levels of context switching and synchronization overhead. The reasons for this overhead is discussed in Section 4.4.2 and Section 4.4.3.

• **Connectionless Performance:** Figures 14 and 15 depict the average throughput for the message-based and task-based threading architectures used to implement the connectionless

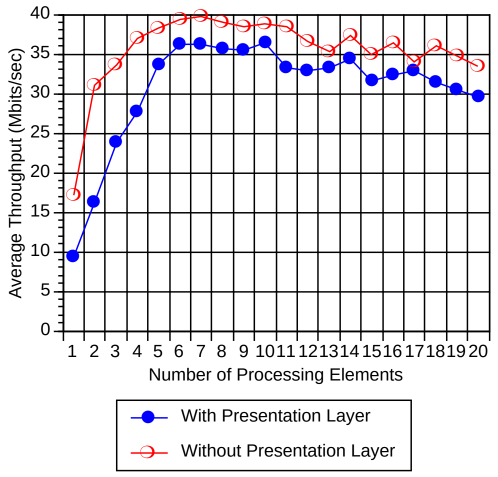

Figure 10: Connection-oriented Layer Parallelism Throughput

protocol stacks[4] As with the other tests, these figures report the throughput measured both with and without presentation layer processing. Figures 16 and 17 indicate the speedup of each threading architecture, relative to its single PE case, for the data points reported in Figures 14 and 15.

- *Message Parallelism* – The connectionless Message Parallelism threading architecture (Figure 14) significantly outperforms the task-based threading architecture (Figure 15). This behavior is consistent with the results from the connection-oriented tests shown in Figure 8 through Figure 13. With presentation layer processing, the throughput and relative speedup of connectionless Message Parallelism is slightly higher than the connection-oriented version shown in Figures 9 and 12. However, without presentation layer processing, the

[4]Note that Connectional Parallelism is not applicable for a connectionless protocol stack.

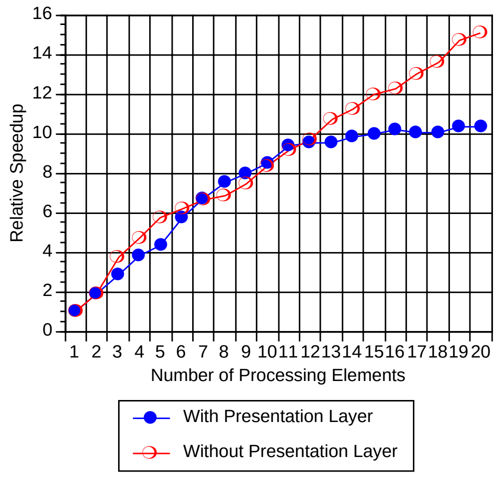

Figure 11: Relative Speedup for Connection-oriented Connectional Parallelism

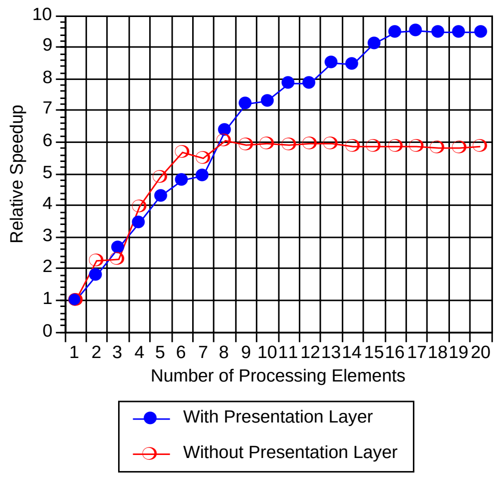

Figure 12: Relative Speedup for Connection-oriented Message Parallelism

throughput and relative speedup of connectionless Message Parallelism are substantially higher (500 Mbits/sec vs. 130 Mbits/sec). In addition, note that the relative speedup of connectionless Message Parallelism without presentation layer processing exceeds that attained with presentation layer processing (shown in Figure 16). This behavior is the reverse of the connection-oriented Message Parallelism results (shown in Figure 12). The difference in performance is due to the fact that connectionless Message Parallelism incurs much lower levels of synchronization overhead (synchronization is discussed further in Section 4.4.3). After 11 PEs, the speedup curve flattens out, indicating that the hardware limitations of the SPARCcenter's shared memory bus have been reached.

- *Layer Parallelism* – The throughput of the connectionless Layer Parallelism (shown in Figure 15) suffered from the same problems as the connection-oriented version (shown in Figure 10). As shown in Figure 13, the relative speedup increases only up to 6 PEs, regardless of whether presentation layer processing was performed or not. At this point, the performance levels off and begins to decrease. This behavior is accounted for by the high levels of context switching incurred by the Layer Parallelism threading architecture, as discussed in the following section.

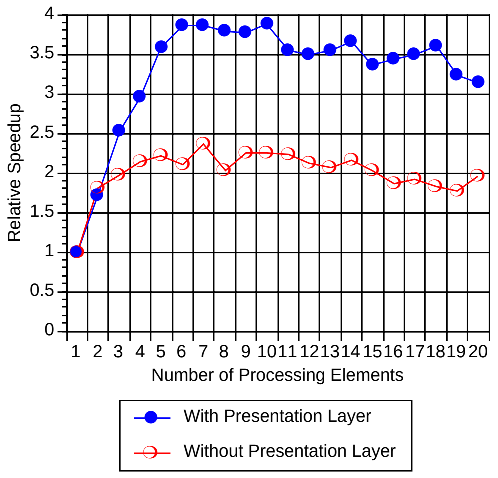

Figure 13: Relative Speedup for Connection-oriented Layer Parallelism

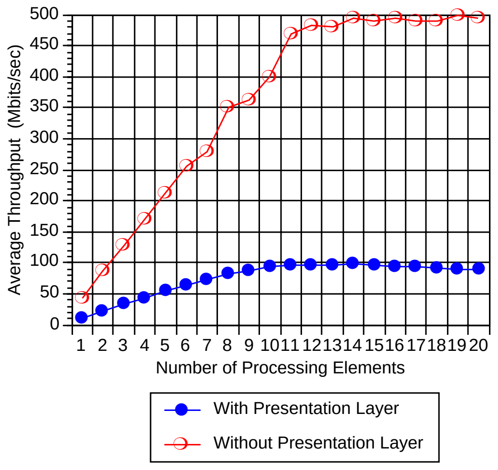

Figure 14: Connectionless Message Parallelism Throughput

4.4.2 Context Switching Measurements

Measurements of context switching overhead were obtained by modifying the `ttcp` benchmarking tool to use the SunOS 5.4 `/proc` file system. The `/proc` file system provides access to the executing image of each OS process and LWP in the system. It reports the number of voluntary and involuntary context switches incurred by SunOS LWPs within a process. Figures 18 through 22 illustrate the number of voluntary and involuntary context switches incurred by trans-

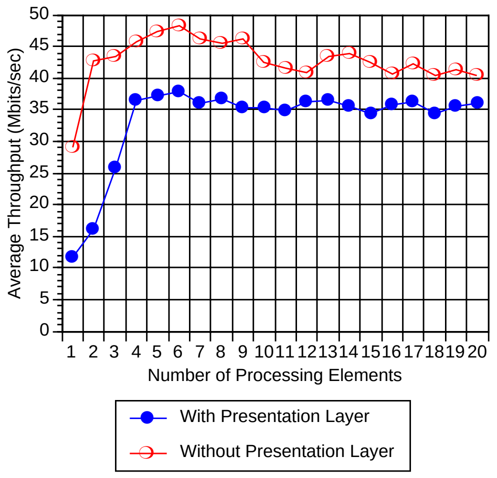

Figure 15: Connectionless Layer Parallelism Throughput

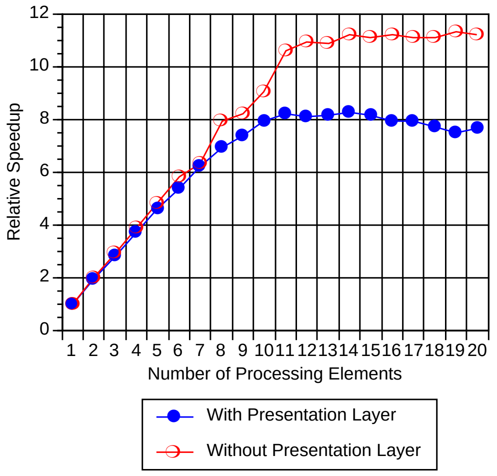

Figure 16: Relative Speedup for Connectionless Message Parallelism

mitting the 20,000 4 Kbyte messages through the threading architectures and protocol stacks measured in this study.

A voluntary context switch is triggered when a protocol task puts itself to sleep awaiting certain resources (such as I/O devices or synchronization locks) to become available. For example, a protocol task may attempt to acquire a resource that is not available immediately (such as obtaining a message from an empty list of messages in a `Task`). In this case, the protocol task puts itself to sleep by invoking the `wait` method of an ASX `Condition` object. This method causes the SunOS kernel to preempt the current thread of control and perform a context switch to another thread of control that is capable of executing protocol tasks immediately. For each combination of threading architecture and protocol stack, voluntary context switching increases fairly steadily as the number of PEs increase from 1 through 20 (shown in Figures 18 through 22).

An involuntary context switch occurs when the SunOS kernel preempts a running unbound thread in order to schedule another thread of control to execute other protocol tasks. The SunOS scheduler preempts an active thread of control every 10 milliseconds when the time-slice alloted to its LWP expires. Note that the rate of growth for involuntary context switching shown in Figures 18 through 22 remains fairly consistent as the number of PEs increase. Therefore, it appears that most of the variance in average throughput performance is accounted for by voluntary context switching, rather than by involuntary context switching.

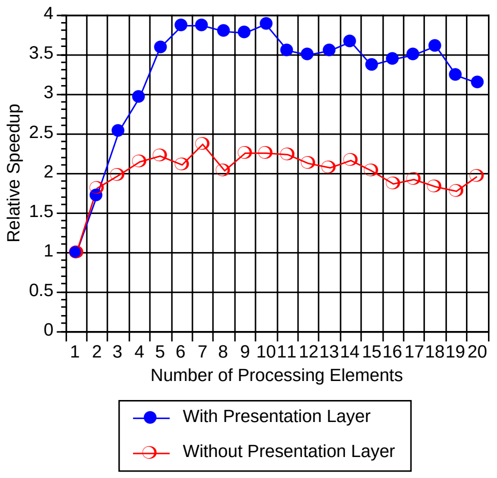

Figure 17: Relative Speedup of Connectionless Layer Parallelism

The following examines the context switching overhead of each threading architecture in detail:

- *Connectional Parallelism* – As shown in Figure 18, Connectional Parallelism incurred the lowest levels of context switching for the connection-oriented protocol stacks. In this threading architecture, after a message has been demultiplexed onto a connection, all that connection's context information is directly accessible within the address space of the associated thread of control. Therefore, a thread of control in Connectional Parallelism can process its connection's messages without incurring additional context switching overhead.
- *Message Parallelism* – As shown in Figure 19, Message Parallelism incurs a higher level of context switching than Connectional Parallelism. This is caused by the higher level of contention for `Mutexes` used in the connection-oriented Message Parallelism implementation to serialize access to resources like transport layer connection control blocks. Since adaptive spin-locks are used, this additional contention yields a slightly higher level of context switching.
- *Layer Parallelism* – The task-based Layer Parallelism threading architectures (shown in Figures 21 and 22) exhibited approximately 4 to 5 times higher levels of voluntary context switching than the message-based threading architectures (shown in Figures 18, 19, and 20). This difference stems from the synchronization mechanisms

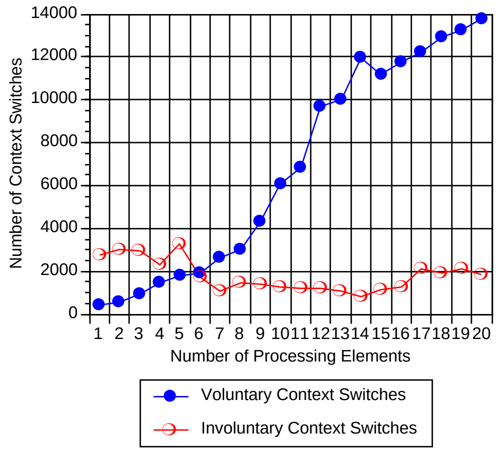

Figure 18: Connection-oriented Connectional Parallelism Context Switching

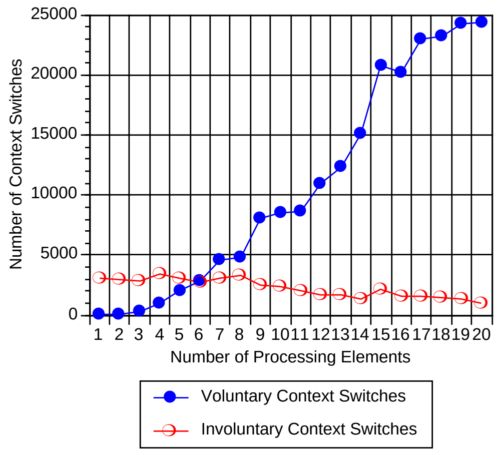

Figure 19: Connection-oriented Message Parallelism Context Switching

used for the Layer Parallelism threading architecture. Layer Parallelism uses sleep-locks to implement flow control between protocol stack layers running in separate PEs.

Layer-to-layer flow control is necessary within a Stream since processing activities in each layer may execute at different rates. In SunOS, the `wait` and `signal` methods on `Condition` objects are implemented using sleep-locks, which trigger voluntary context switches. In contrast, Connectional Parallelism and Message Parallelism use adaptive spin-lock synchronization, which is less costly since it typically does *not* trigger voluntary context switches. The substantially lower-levels of voluntary context switching exhibited by Connectional Parallelism and Message Parallelism helps to account for their significantly higher overall throughput and greater relative speedup discussed in Section 4.4.1.

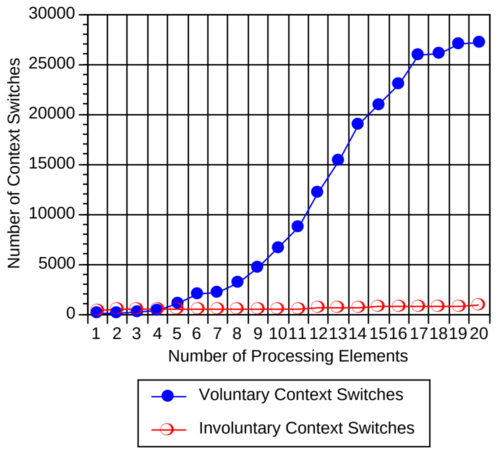

Figure 20: Connectionless Message Parallelism Context Switching

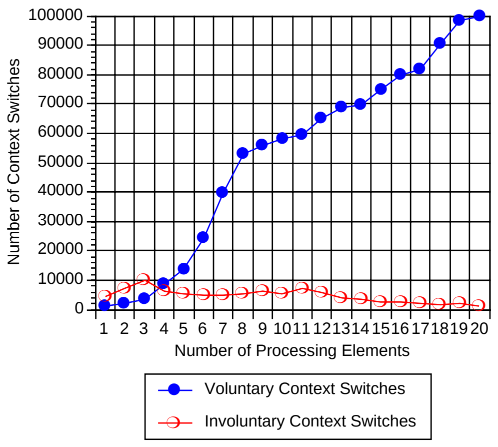

Figure 21: Connection-oriented Layer Parallelism Context Switching

4.4.3 Synchronization Measurements

Measurements of synchronization overhead were collected to determine the amount of time spent acquiring and releasing locks on ASX `Mutex` and `Condition` objects during protocol processing on the 20,000 4 Kbyte messages. Unlike context switches, the SunOS 5.4 `/proc` file system does not maintain accurate metrics on synchronization overhead. Therefore, these measurements were obtained by bracketing the ASX `Mutex` and `Condition` methods with calls to the `gethrtime` system call. This system call uses the SunOS 5.4 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by the `gethrtime` system call is very accurate since it does not drift.

Figures 23 through 27 indicate the total time (measured in msecs) used to acquire and release locks on `Mutex` and `Condition` synchronization objects. These tests were performed using all three threading architectures to implement connection-oriented and connectionless protocol stacks

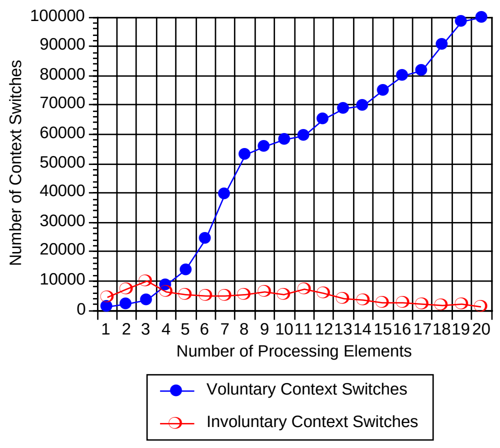

Figure 22: Connectionless Layer Parallelism Context Switching

that contained data-link, network, transport, and presentation layer functionality. The message-based threading architectures (Connectional Parallelism and Message Parallelism, shown in Figures 23, 24, and 26) used `Mutex` synchronization mechanisms that utilize adaptive spin-locks (which rarely trigger a context switch). In contrast, the task-based threading architecture (Layer Parallelism, shown in Figures 25 and 27) utilized both `Mutex` and `Condition` objects (`Condition` objects do trigger context switches).

The following examines the synchronization overhead of each threading architecture in detail:

- *Connectional Parallelism* – Connection-oriented Connectional Parallelism (shown in Figure 23) exhibited the lowest levels of synchronization overhead, which peaked at approximately 700 msecs. This synchronization overhead was approximately 1 order of magnitude lower than the results shown in Figures 24 through 27. Moreover, the amount of synchronization overhead incurred by Connectional Parallelism did not increase significantly as the number of PEs increased from 1 to 20. This behavior occurs since after a message is demultiplexed onto a PE/connection, few additional synchronization operations are required. In addition, since Connectional Parallelism processes messages within a single PE cache, it leverages off of SPARCcenter 2000 multi-processor cache affinity properties [17].
- *Message Parallelism* – The synchronization overhead incurred by connection-oriented Message Parallelism (shown in Figure 24) peaked at just over 6,000 msecs. Moreover, the rate of growth increased fairly steadily as the number of PEs increased from 1 to 20. This behavior occurs from the lock contention caused by `Mutex` objects that serialize access to the `Map_Manager` connection demultiplexer (discussed in Section 3.2). In contrast, the connectionless Message Parallelism protocol stack does not require the use of this connection demultiplexer. Therefore, the amount of synchronization overhead it incurred was much lower, peaking at under 1,800 msecs.
- *Layer Parallelism* – Connection-oriented Layer Parallelism exhibited two types of synchronization overhead (shown in Figure 25). The amount of overhead resulting from `Mutex` objects peaked at just over 2,000 msecs, which was lower than that of connection-oriented Message Parallelism (shown in Figure 24). However, the amount of synchronization overhead from the `Condition` objects was much higher, peaking at approximately 18,000 msecs (shown in Figure 25). In the Layer Parallelism implementation, the `Condition` objects implemented flow control between separate layers executing on different PEs in a protocol stack. The connectionless version of Layer Parallelism also exhibited high levels of synchronization overhead (shown in Figure 27).

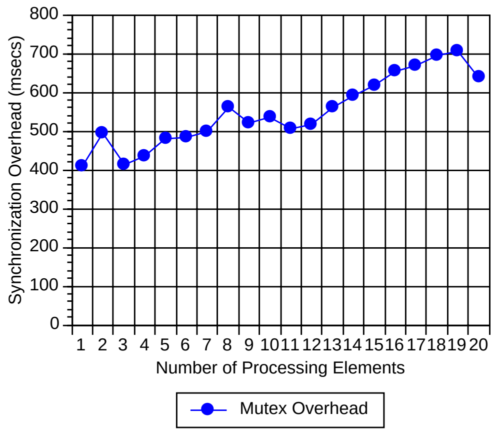

Figure 23: Connection-oriented Connectional Parallelism Synchronization Overhead

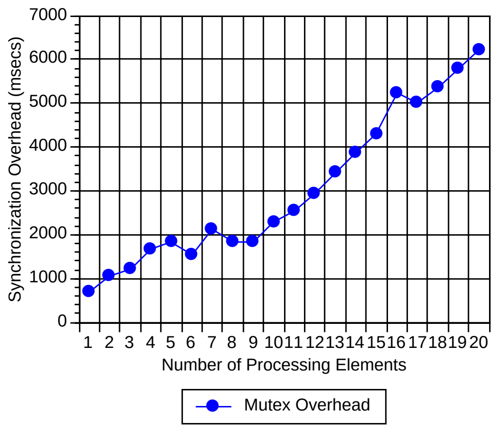

Figure 24: Connection-oriented Message Parallelism Synchronization Overhead

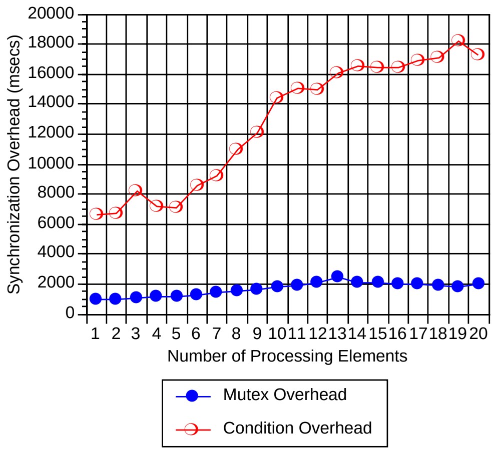

Figure 25: Connection-oriented Layer Parallelism Synchronization Overhead

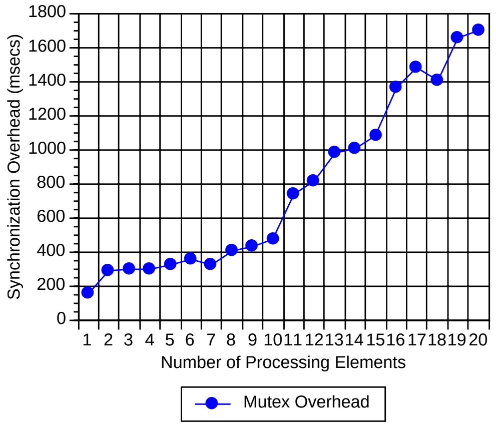

Figure 26: Connectionless Message Parallelism Synchronization Overhead

5 Summary of Observations and Recommendations

This section summarizes our observations and presents recommendations for using threading architectures effectively.

5.1 Summary of Observations

The following is a summary of observations based on the results of our performance experiments on alternative threading architectures for connectionless and connection-oriented protocol stacks.

• **Task-based threading architectures are easier to implement than the message-based process architectures:** The task-based Layer Parallelism threading architecture maps cleanly onto conventional layered communication models using a well-structured "producer/consumer" model [32]. Synchronization *within* a layer is minimal since parallel processing is serialized at service access points (such as the

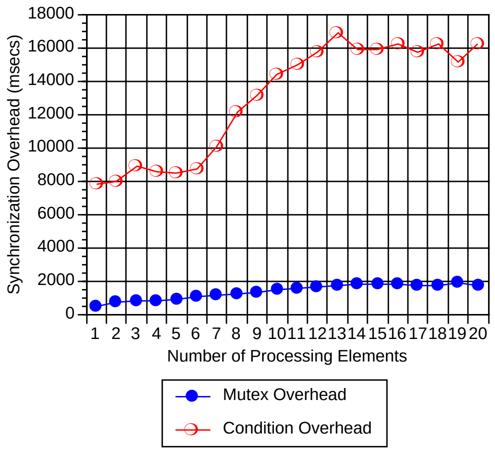

Figure 27: Connectionless Layer Parallelism Synchronization Overhead

service access point defined between the network and transport layers). In contrast, implementing the message-based threading architectures is more complex due to the need for more sophisticated concurrency control, thread pools, and demultiplexing algorithms.

• **Message-based threading architectures are more efficient than task-based threading architectures:** Despite being harder to program, the message-based process architectures use parallelism more effectively than the task-based threading architectures. This is due in part to the fact that message-based threading architecture parallelism is based upon dynamic characteristics (such as messages or connections). As described in Section 4.4.1, the relative speedups gained from parallel message-based threading architectures scaled up to use a relatively high number of PEs. In contrast, the parallelism used by the task-based threading architectures depended on relatively static characteristics (such as the number of layers or protocol tasks), which did not scale up. In addition, the higher rate of growth for context switching and synchronization (discussed in Sections 4.4.2 and 4.4.3) prevented Layer Parallelism from using a large number of PEs effectively.

• **Connectional Parallelism works best when the number of connections is $>=$ the number of PEs:** Connectional Parallelism becomes more effective when the number of connections approaches the number of PEs. Message Parallelism, on the other hand, is more suitable when the number of active connections is significantly less than the number of available PEs. Figure 28 illustrates this point by graphing average throughput as a function of the number of connections. This test held the number of PEs constant at 20, while increasing the number of connections from 1 to 20. Connectional Parallelism consistently out-performs Message Parallelism as the number of connections becomes larger than 10.

• **Connection-oriented Message Parallelism benefits from more PEs when the protocol stack performs presentation layer processing:** As shown in Figure 12, the speedup

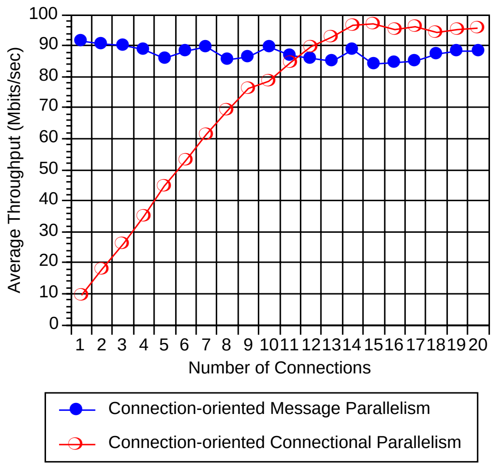

Figure 28: Comparison of Connectional Parallelism and Message Parallelism

curve for connection-oriented Message Parallelism without presentation layer processing flattens out after 8 PEs. In contrast, when presentation layer processing is performed, the speedup continues until 16 PEs, at which point the bandwidth of the shared memory bus becomes the limiting factor. This behavior results from the relatively low amount of synchronization overhead associated with parallel processing at the presentation layer.

- **The relative cost of synchronization operations has a substantial impact on threading architecture performance:** On the SPARCcenter 2000 shared memory multiprocessor running SunOS 5.4, the message-based threading architectures benefit from their use of relatively inexpensive adaptive spin-locks. In contrast, the task-based threading architectures were penalized by the much higher cost of sleep-lock synchronization, (*i.e.,* two orders of magnitude – 4 μsecs vs. 300 μsecs for spin-locks vs. sleep-locks, respectively).

We conjecture that a multi-processor platform possessing different synchronization properties would produce significantly different results. For example, if the experiments reported in this paper were replicated on a non-shared memory, message-passing transputer platform [6], it is likely that the performance of the task-based threading architectures would improve relative to the message-based threading architectures.

5.2 Recommendations

The following recommendations for using threading architectures effectively are based on our performance experiments and our experience gained building and testing the alternative threading architectures:

- *If program simplicity is a more important requirement than performance, we recommend the use of task-based threading architectures.*
- *If performance is a more important requirement than program simplicity, we recommend the use of message-based threading architectures.*
- *If the number of connections is $>=$ than the number of PEs and the connections process data at approximately the same rate, we recommend the use of Connectional Parallelism.*
- *If the number of connections is $<<$ than the number of PEs or the connections process data at highly disproportional rates, we recommend the use of Message Parallelism.*

6 Concluding Remarks

This paper describes communication protocol stack performance measurements obtained using the ASX framework. This framework provides an integrated set of object-oriented components that facilitate controlled experimentation with message-based and task-based threading architectures on multi-processor platforms. The ASX framework contributed to these performance experiments by helping to decouple the protocol-specific functionality from the underlying threading architecture. This decoupling increased component reuse and simplified the development, configuration, and experimentation with alternative parallel protocol stacks.

The experimental results presented in this paper demonstrate that to increase performance significantly, the speed-up obtained from parallelizing a protocol stack must outweight the context switching and synchronization overhead associated with parallel processing. If these sources of overhead are large, parallelizing a protocol stack will not yield substantial benefits. The task-based Layer Parallelism threading architecture exhibited high levels of context switching and synchronization, and did not effectively utilize the available multi-processing resources on a SPARCcenter 2000 shared memory multi-processor platform containing 20 processing elements. In contrast, the message-based threading architectures (Connectional Parallelism and Message Parallelism) incurred significantly less context switching and synchronization overhead, and exhibited much higher levels of performance and multi-processing resource utilization. In general, the results from these experiments underscore the importance of the threading architecture on parallel communication subsystem performance.

Components in the ASX framework described in this paper are freely available via the World Wide Web at URL `http://www.cs.wustl.edu/~schmidt/ACE.html`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ACE object-oriented network programming toolkit [23] developed at the University of California, Irvine and Washington University. The ACE toolkit is currently being used on communication software for production communication systems at Motorola, Siemens, Ericsson, and Kodak, as well as many academic research projects.

References

[1] M. Zitterbart, B. Stiller, and A. Tantawy, “A Model for High-Performance Communication Subsystems,” *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 507–519, May 1993.

[2] D. D. Clark and D. L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[3] D. C. Schmidt and T. Suda, “Transport System Architecture Services for High-Performance Communications Systems,” *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[4] J. C. Mogul and A. Borg, “The Effects of Context Switches on Cache Performance,” in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Santa Clara, CA), ACM, Apr. 1991.

[5] Mats Bjorkman and Per Gunningberg, “Locking Strategies in Multiprocessor Implementations of Protocols,” *Transactions on Networking*, vol. 3, no. 6, 1996.

[6] M. Zitterbart, “High-Speed Transport Components,” *IEEE Network Magazine*, pp. 54–63, January 1991.

[7] M. Goldberg, G. Neufeld, and M. Ito, “A Parallel Approach to OSI Connection-Oriented Protocols,” in *Proceedings of the 3rd IFIP Workshop on Protocols for High-Speed Networks*, (Stockholm, Sweden), May 1992.

[8] T. L. Porta and M. Schwartz, “Performance Analysis of MSP: a Feature-Rich High-Speed Transport Protocol,” in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (San Francisco, California), IEEE, 1993.

[9] C. M. Woodside and R. G. Franks, “Alternative Software Architectures for Parallel Protocol Execution with Synchronous IPC,” *IEEE/ACM Transactions on Networking*, vol. 1, Apr. 1993.

[10] B. Lindgren, B. Krupczak, M. Ammar, and K. Schwan, “Parallelism and Configurability in High Performance Protocol Architectures,” in *Proceedings of the Second Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, (Williamsburg, Virgina), IEEE, September 1993.

[11] T. Braun and M. Zitterbart, “Parallel Transport System Design,” in *Proceedings of the 4th IFIP Conference on High Performance Networking*, (Belgium), IFIP, 1993.

[12] D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. Williamson, “High-Speed Parallel Protocol Implementations,” in *Proceedings of the 1st International Workshop on High-Speed Networks*, pp. 165–180, May 1989.

[13] N. C. Hutchinson and L. L. Peterson, “The *x*-kernel: An Architecture for Implementing Network Protocols,” *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[14] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, “Performance Issues in Parallelized Network Protocols,” in *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, USENIX Association, November 1994.

[15] D. Presotto, “Multiprocessor Streams for Plan 9,” in *Proceedings of the United Kingdom UNIX User Group Summer Proceedings*, (London, England), Jan. 1993.

[16] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, “Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes,” in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[17] J. D. Salehi, J. F. Kurose, and D. Towsley, “The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking,” in *IEEE INFOCOM*, (San Francisco, USA), IEEE Computer Society Press, Mar. 1996.

[18] A. Garg, “Parallel STREAMS: a Multi-Process Implementation,” in *Proceedings of the Winter USENIX Conference*, (Washington, D.C.), Jan. 1990.

[19] S. McCanne and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture,” in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 259–270, Jan. 1993.

[20] D. Ritchie, “A Stream Input–Output System,” *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[21] H. Hueni, R. Johnson, and R. Engel, “A Framework for Network Protocol Software,” in *Proceedings of OOPSLA ’95*, (Austin, Texas), ACM, October 1995.

[22] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, “Language Support for Flexible, Application-Tailored Protocol Configuration,” in *Proceedings of the 18th Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, IEEE, Sept. 1993.

[23] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[24] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[25] D. C. Schmidt, “An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit,” Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

[26] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, “Beyond Multiprocessing... Multithreading the SunOS Kernel,” in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[27] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas, “Tools for Implementing Network Protocols,” *Software Practice and Experience*, vol. 19, pp. 895–916, September 1989.

[28] A. Gokhale and D. C. Schmidt, “Measuring the Performance of Communication Middleware on High-Speed Networks,” in *Proceedings of SIGCOMM ’96*, (Stanford, CA), ACM, August 1996.

[29] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[30] V. Jacobson, “Congestion Avoidance and Control,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Stanford, Calif.), August 1988.

[31] D. D. Clark, “The Structuring of Systems Using Upcalls,” in *Proceedings of the 10th Symposium on Operating System Principles*, (Shark Is., WA), 1985.

[32] M. S. Atkins, “Experiments in SR with Different Upcall Program Structures,” *ACM Transactions on Computer Systems*, vol. 6, pp. 365–392, November 1988.