Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance

Douglas C. Schmidt and Tatsuya Suda
schmidt@ics.uci.edu and suda@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, California 92717[1]

This paper appeared in the proceedings of the 1994 IFIP 4^{th} International Workshop on Protocols for High-Speed Networks in Vancouver, British Columbia, August 10–12, 1994.

Abstract

A communication subsystem consists of protocol functions and operating system mechanisms that support the implementation and execution of protocol stacks. To effectively parallelize a communication subsystem, careful consideration must be given to the process architecture used to structure multiple processing elements. A process architecture binds one or more processing elements with the protocol tasks and messages associated with protocol stacks in a communication subsystem. This paper outlines the two fundamental types of process architectures (task-based and message-based) and describes performance experiments conducted on three representative examples of these two types of process architectures – Layer Parallelism, which is a task-based process architecture, and Message-Parallelism and Connectional Parallelism, which are message-based process architectures. These experiments measure the impact of the process architecture on connectionless and connection-oriented protocol stacks (based upon UDP and TCP) in a shared-memory multi-processor operating system. The results from these experiments indicate that the choice of process architecture significantly affects communication subsystem performance.

1 Introduction

Advances in VLSI and fiber optic technology are shifting performance bottlenecks from the underlying networks to the communication subsystem. A communication subsystem consists of *protocol functions* (such as connection management, end-to-end flow control, remote context management, segmentation/reassembly, demultiplexing, message buffering, error protection, session control, and presentation conversions) and *operating system mechanisms* (such as process management, asynchronous event invocation, message buffering, and layer-to-layer flow control) that support the implementation and execution of communication protocol stacks composed of protocol functions.

Executing protocol functions and OS mechanisms in parallel on multi-processor platforms is a promising technique for increasing protocol processing rates and reducing latency. To significantly increase communication subsystem performance on shared memory multi-processor platforms, however, the speed-up obtained from parallelism must outweight the *context switching* and *synchronization* overhead associated with parallel processing. A context switch is triggered when an executing process relinquishes its associated processing element (PE) voluntarily or involuntarily. Depending on the underlying OS and hardware platform, performing a context switch may involve dozens to hundreds of instructions to flush register windows, memory caches, instruction pipelines, and translation look-aside buffers. Synchronization overhead arises from locking mechanisms that serialize access to shared objects (such as messages, message queues, protocol connection records, and demultiplexing tables) used when processing protocols in parallel.

A number of *process architectures* have been proposed as the basis for parallelizing communication subsystems [1, 2, 3, 4]. There are two fundamental types of process architectures: *task-based* and *message-based*. Task-based process architectures are formed by binding one or more PEs to units of protocol functionality (such as presentation layer formatting or transport layer segmentation/reassembly, acknowledgment processing, end-to-end flow control, and retransmission timer processing). In a task-based process architecture, 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 process architectures are formed by binding the PEs to data messages and control messages received from applications and network interfaces. In a message-based process architecture, parallelism is achieved by escorting multiple data messages and control messages on separate PEs through a stack of protocol tasks.

Protocol suites (such as the Internet and ISO OSI reference models) may be implemented using either task-based or message-based process architectures. However, these two

[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, and Tokyo Electric Power Company.

types of process architectures exhibit significantly different performance characteristics that are affected by the underlying operating system and hardware platform. For instance, on shared memory multi-processor platforms, task-based process architectures often result in high data movement and context switching overhead [5]. Likewise, in a message-passing transputer multi-processor environment, message-based process architectures typically result in high levels of synchronization overhead [2].

Existing research has generally selected a single type of process architecture (either task-based or message-based) and studied it in isolation. Moreover, since different studies have been performed on different OS and hardware platforms, using different protocols and implementation techniques, it is difficult to compare the results obtained from these studies in a controlled manner. This paper describes results obtained from systematic comparisons of the performance impact of task-based and message-based process architectures. These results were obtained using an object-oriented framework that facilitates controlled experiments with alternative process architectures on shared memory multi-processor platforms [6]. The framework controls for a number of key confounding factors (such as protocol functionality, concurrency control schemes, and application traffic characteristics) in order to precisely measure the performance impact of different process architectures for parallelizing communication protocol stacks.

This paper is organized as follows: Section 2 outlines the fundamental types of process architectures and compares related work accordingly; Section 3 describes the design and implementation of the protocol stacks and process architectures used in the experiments reported in Section 4; and Section 5 presents concluding remarks.

2 Alternative Process Architectures

Figure 1 (1) illustrates the basic elements that form the foundation of a process architecture:

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

There are two fundamental types of process architectures (*task-based* and *message-based*) that structure these basic elements differently. Task-based process architectures bind one or more PEs to protocol processing tasks. In this architecture, tasks are the active elements, whereas messages processed by the tasks are the passive elements (shown in Figure 1 (2)). Conversely, message-based process architectures bind the PEs to the control messages and data messages received from applications and network interfaces. In this architecture, messages are the active elements and tasks are the passive elements (shown in Figure 1 (3)).

The remainder of this section briefly examines several alternative process architectures in each category.

2.1 Task-based Process Architectures

Task-based process architectures associate processes[2] with clusters of one or more protocol tasks. Two representative examples of task-based process architectures are the *Layer Parallelism* and *Functional Parallelism* process architectures. The primary difference between these two process architectures involves the granularity of the protocol processing tasks. Layers are more "coarse-grained" than functions since they cluster multiple protocol tasks together to form a composite service (such as the end-to-end transport service provided by the OSI transport layer).

Layer Parallelism associates a separate process with each layer (*e.g.,* the presentation, transport, and network layers) in a protocol stack. Certain protocol header and data fields in the outgoing and incoming messages may be processed in parallel as they flow through a pipeline of protocol stack layers. Buffering and flow control are generally necessary since processing activities in each layer may execute at different rates.

Functional Parallelism associates a separate process with each protocol function (such as header composition, acknowledgement, retransmission, segmentation, reassembly, and routing). These protocol functions execute in parallel and communicate by passing control messages and data messages to each other.

In general, implementing pipelined task-based process architectures is relatively straightforward. Task-based process architectures map directly onto conventional layered communication models using well-structured "producer/consumer" designs. Moreover, minimal synchronization mechanisms are necessary *within* a layer or function since parallel processing is typically serialized at a service access point (such as the transport layer or application layer interface). However, as shown in Section 4, task-based process architectures are susceptible to high context switching overhead on shared memory platforms. This problem is exacerbated when the number of protocol tasks exceeds the number of PEs, due to the context switching performed when transferring messages between protocol tasks.

2.2 Message-based Process Architectures

Message-based process architectures associate processes with messages rather than protocol layers or functions. Two common examples of message-based process architectures

[2] In this paper, the term "process" is used to refer to a series of instructions executing within an address space; this address space may be shared with other processes. Different terminology (such as lightweight processes [6] or threads [7]) has also been used to denote the same basic concepts. Our use of the term process is consistent with the definition adopted in [8].

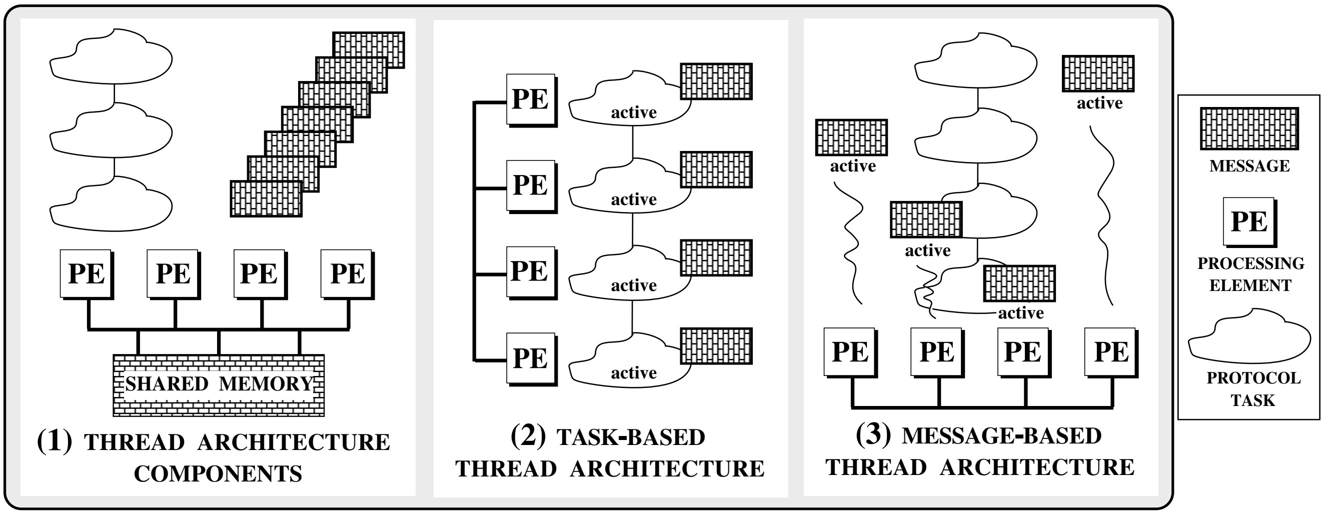

Figure 1: Basic Process Architecture Components and Interrelationships

are *Connectional Parallelism* and *Message Parallelism*. The primary difference between these approaches involves the granularity at which messages are demultiplexed onto processes. Connectional Parallelism demultiplexes all messages bound for the same connection onto the same process, whereas Message Parallelism demultiplexes messages onto any available process.

Connectional Parallelism uses a separate process to handle the messages associated with each open connection. Within a connection, a series of protocol processing tasks are invoked sequentially on each message as it flows through a protocol stack. Outgoing messages generally borrow the thread of control from the application process and use it to escort messages down a protocol stack. For incoming messages, a network interface or packet filter typically performs demultiplexing operations to determine the correct process for each message.

Message Parallelism associates a separate process with every incoming or outgoing message. A process receives a message from an application or network interface and escorts the message through the protocol processing tasks in the protocol stack. As with Connectional Parallelism, outgoing messages generally borrow the thread of control from the application that initiated the message transfer.

In general, a large degree of potential parallelism exists with the message-based process architectures. The degree of parallelism depends on characteristics that change dynamically (such as messages or connections), rather than on the relatively static characteristics (such as the number of layers or protocol functions) that are associated with task-based process architectures. Depending on other communication subsystem characteristics (such as memory and bus bandwidth), this dynamism may enable message-based process architectures to effectively use a larger number of PEs.

2.3 Related Work

A number of studies have investigated the performance characteristics of task-based process architectures developed to run on either message passing or shared memory platforms. [5] measures the impact of several implementations of the transport and session layers in the OSI reference model using an ADA-like rendezvous-style of Layer Parallelism in a nonuniform access shared memory environment. [9] measures the performance of a Functional Parallelism process architecture for presentation layer and transport layer functionality on a shared memory multi-processor. [10] measures the performance of a de-layered, function-oriented transport system [11] using Functional Parallelism on a message passing transputer multi-processor platform. An earlier study [2] measured the performance of the OSI transport layer and network layer in a similar transputer environment. [12] also uses a multi-processor transputer platform to measure the performance of several data-link layer protocols.

Other studies have investigated message-based process architectures. All these studies utilize shared memory platforms. [13] measured the performance of the TCP, UDP, and IP protocols using a Message Parallelism process architecture on a uniprocessor platform running the *x*-kernel. [1] measures the impact of synchronization on Message Parallelism implementations of TCP and UDP transport protocols built within a multi-processor version of the *x*-kernel. [8] measures the performance of the Nonet transport protocol on a multi-processor version of Plan 9 STREAMS developed using Message Parallelism. [3] measures the performance of the OSI protocol stack, focusing primarily on the presentation and transport layers using Message Parallelism. [14] measures the performance of the TCP/IP protocol stack using Connectional Parallelism in a multi-processor version of System V STREAMS.

The work presented in this paper extends existing work by measuring a number of task-based and message-based process architectures in a controlled environment. Our ex-

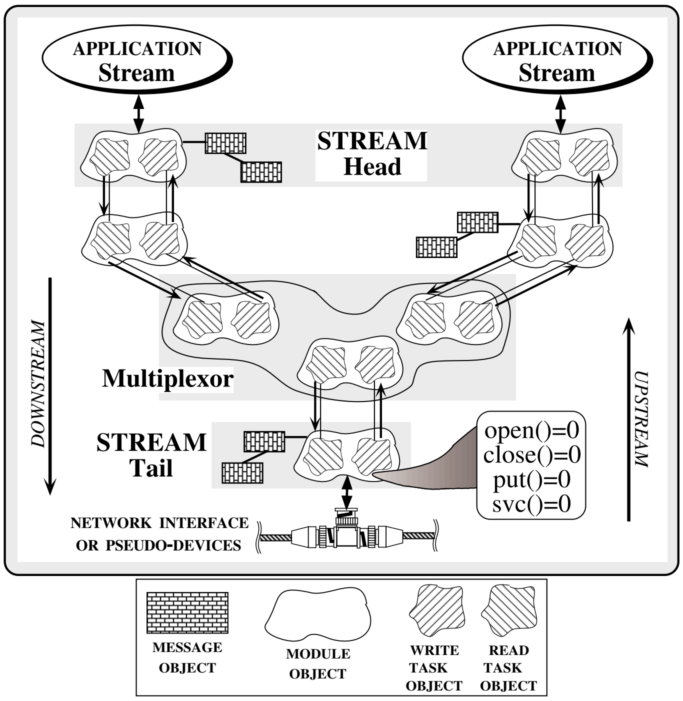

Figure 2: Components in the ADAPTIVE Service eXecutive Framework

periments consider the impact of both synchronization and context switching overhead. In addition to measuring data link, network, and transport layer performance, our experiments also investigate presentation layer performance. The presentation layer is widely considered to be one of the primary bottlenecks in high-performance communication subsystems.

3 Structure of the Experiments

This section describes the object-oriented framework, communication protocols, and process architectures we developed and used in the performance experiments reported in Section 4.

3.1 The ADAPTIVE Service eXecutive Framework

The communication protocols and process architectures in this study were developed using components provided by the ADAPTIVE Server eXecutive (`ASX`) framework [15]. The `ASX` framework contains an integrated set of object-oriented components that facilitate experimentation with task-based and message-based process architectures on shared memory multi-processor platforms.

Components in the `ASX` are responsible for coordinating one or more *Streams*. A Stream is an object used to configure and execute protocol-specific functionality in the `ASX` framework run-time environment. As illustrated in Figure 2, a Stream contains a series of inter-connected `Modules` that may be linked together by developers at installation-time or by applications at run-time. `Modules` are objects that developers use to decompose the architecture of a protocol stack into a series of inter-connected, functionally distinct layers. Each layer implements a cluster of related protocol-specific functions (such as an end-to-end transport service, a presentation layer formatting service, or a real-time PBX signal routing service). Every `Module` contains a pair of `Queue` objects that partition a layer into its constituent read-side and write-side protocol-specific processing functionality.

Any layer that performs multiplexing and demultiplexing of message objects between related Streams may be developed using a `Multiplexor` object. A `Multiplexor` is a C++ template-based container class that provides mechanisms to route messages between `Modules` in a collection of related Streams. A complete Stream is represented as an inter-connected series of `Module` objects that communicate by exchanging messages with adjacent objects. `Modules` and `Multiplexors` may be joined together in essentially arbitrary configurations in order to satisfy application requirements and enhance component reuse.

The `ASX` framework employs a number of object-oriented design techniques (such as design patterns [16] and hierarchical decomposition) and C++ language features (such as inheritance, dynamic binding, and parameterized types). These design techniques and language features enable developers to incorporate protocol-specific functionality into a Stream without modifying the protocol-independent framework components. For example, incorporating a new level of protocol functionality into a Stream at installation-time or at run-time involves the following steps:

1. Inheriting from the `Queue` interface and selectively overriding several methods (described below) in the `Queue` subclass to implement protocol-specific functionality
2. Allocating a new `Module` that contains two instances (one for the read-side and one for the write-side) of the protocol-specific `Queue` subclass
3. Inserting the `Module` into a Stream object at the appropriate level (*e.g.,* the transport layer, network layer, data-link layer, etc.)

The `ASX` framework incorporates concepts from several other modular communication frameworks including System V STREAMS [17], the *x*-kernel [13], and the Conduit [18] (a survey of these and other communication frameworks appears in [19]). These frameworks all contain features that support the flexible configuration of communication subsystems by inter-connecting building-block protocol components. These frameworks encourage the development of standard reusable protocol components by decoupling protocol-specific processing functionality from the surrounding framework infrastructure. In addition to supplying building-block protocol and service components, the `ASX` framework also extends the existing communication frameworks by providing additional components that decouple protocol functionality from the following configuration decisions:

- The type of locking mechanisms used to synchronize access to shared objects
- The use of message-based and task-based process architectures
- The use of kernel-level vs. user-level execution agents

3.2 Communication Protocols

Two types of protocol stacks are used in the experiments. One protocol stack is based on the connectionless UDP transport protocol. The other protocol stack is based on the connection-oriented TCP transport protocol. The protocol stacks contain the data-link, network, transport, and presentation layers. The presentation layer is included in the experiments since it represents a major bottleneck in high-performance communication subsystems, due primarily to the large amount of data movement overhead it often incurs.

Both the connectionless and connection-oriented protocol stacks were developed by specializing reusable components in the `ASX` framework via inheritance and parameterized types. Inheritance and parameterized types are used to hold the protocol functionality constant while systematically varying the process architecture. Each layer in a protocol stack is implemented as a `Module` whose read-side and write-side both inherit interfaces and implementations from the `Queue` class described in [15]. The necessary synchronization and demultiplexing mechanisms are parameterized using C++ template arguments that are instantiated based on the type of process architecture being tested.

Data-link layer processing in each protocol stack is performed by the `DLP Module`. This `Module` transforms network packets received from a network interface into the canonical message format used internally by the Stream components.[3] The network and transport layer components of the protocol stacks are based on the IP, UDP, and TCP implementation in the BSD 4.3 Reno release. The 4.3 Reno TCP implementation contains the TCP header prediction enhancements, as well as the slow start algorithm and congestion avoidance features. The UDP and TCP transport protocols are configured into the `ASX` framework via the `UDP` and `TCP Modules`. Network layer processing is performed by the `IP Module`. This `Module` performs routing and segmentation/reassembly of Internet Protocol (IP) packets.

Presentation layer functionality is implemented in the `XDR Module` using marshalling routines produced by the ONC eXternal Data Representation (XDR) stub generator (`rpcgen`). The ONC XDR stub generator automatically

[3]Preliminary tests using the widely-available `ttcp` benchmarking tool indicated that the PE, bus, and memory performance of the SunOS multiprocessor platform used in the experiments was capable of processing messages through the protocol stack at a much faster rate than the 10 Mbps Ethernet network interface was capable of handling. Therefore, for our process architecture experiments, the network interface was simulated with a single-copy pseudo-device driver operating in loop-back mode.

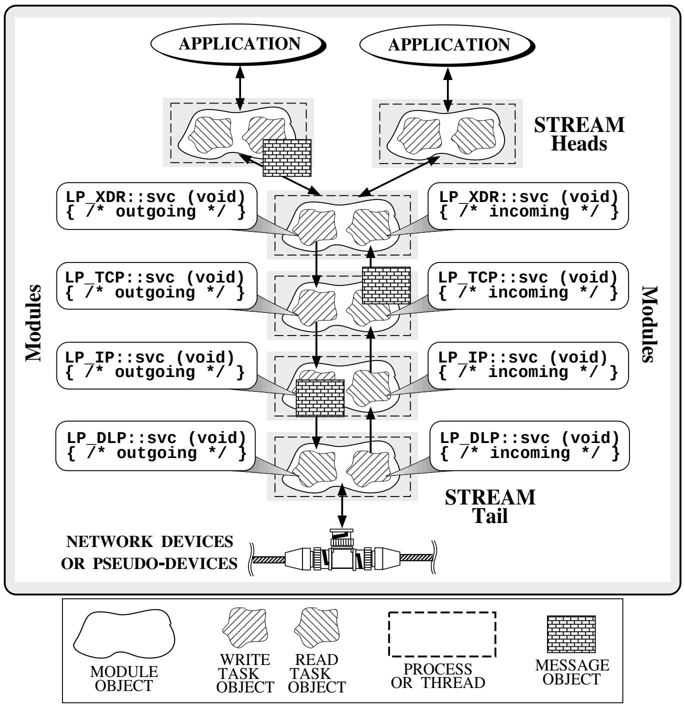

Figure 3: Layer Parallelism

translates a set of type specifications into marshalling routines. These routines encode/decode implicitly-typed messages before/after they are exchanged 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) and real types (*e.g.,* float and double). In addition, these library routines may be combined to produce marshalling routines for arbitrary user-defined composite types (such as record/structures, unions, arrays, and pointers). Messages exchanged via XDR are implicitly-typed, which improves marshalling performance at the expense of run-time flexibility. The `XDR` functions selected for both the connectionless and connection-oriented protocol stacks convert incoming and outgoing messages into and from variable-sized arrays of structures containing both integral and real values. This conversion processing involves byte-order conversions, as well as dynamic memory allocation and deallocation.

3.3 Process Architectures

The remainder of this section outlines the structure of connectionless and connection-oriented protocol stacks developed using task-based and message-based process architectures.

3.3.1 Structure of the Task-based Process Architecture

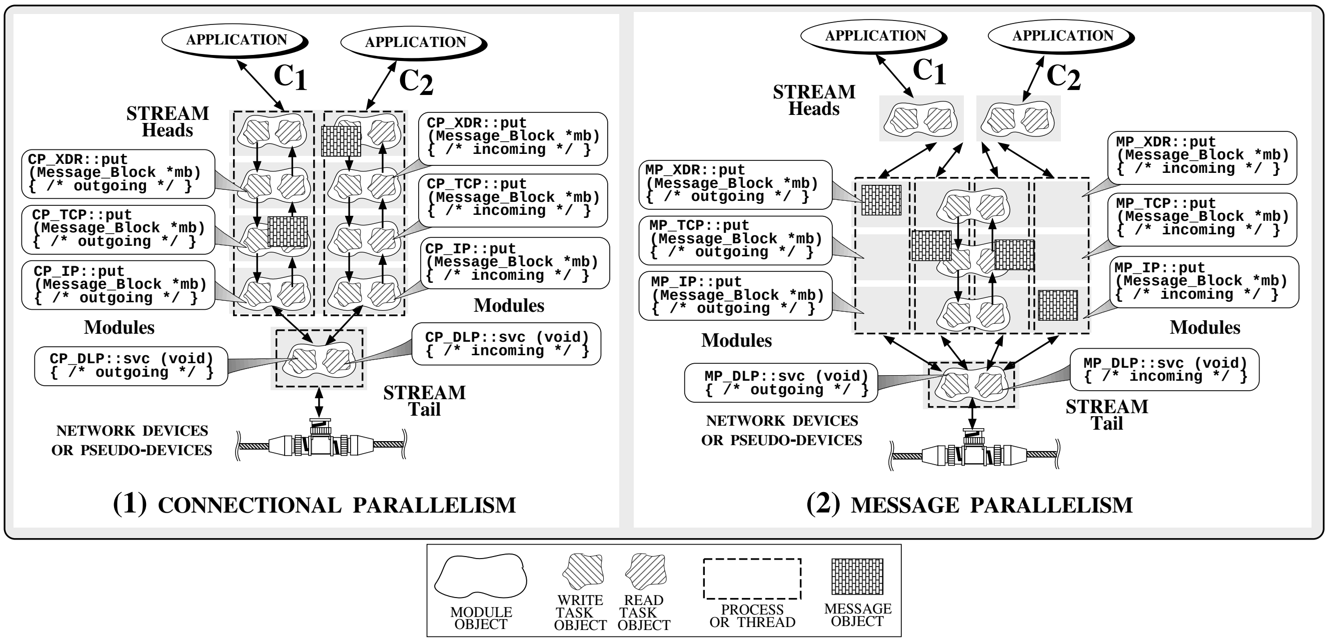

Figure 4: Message-based Process Architectures

• **Layer Parallelism:** Figure 3 illustrates the ASX framework components that implement a Layer Parallelism process architecture for the TCP-based connection-oriented and UDP-based connectionless protocol stacks. Protocol-specific processing at each protocol layer is performed via the Queue::svc method. This method is invoked by a daemon process associated with the Module that implements the protocol layer (*e.g.,* LP_XDR, LP_TCP, LP_IP, and LP_DLP). These daemon processes cooperate in a producer/consumer manner, operating on the header and data fields of messages corresponding to their particular protocol layer in parallel. Each svc method performs its protocol functions before passing the message to an adjacent Module that runs asynchronously in a separate daemon process. Since daemon processes all share a common address space, messages are not copied when passed between adjacent Modules. However, moving messages between processes may invalidate per-PE data caches.

The connectionless and connection-oriented Layer Parallelism process architecture protocol stacks are designed in a similar manner. The primary difference is that the objects in the connectionless transport layer Module implement the simpler UDP functionality. UDP does not generate acknowledgements, keep track of round-trip time estimates, or manage congestion windows, etc.

3.3.2 Structure of the Message-based Process Architectures

• **Connectional Parallelism:** The protocol stack depicted in Figure 4 (1) illustrates an ASX-based implementation of the Connectional Parallelism process architecture. Each connection is associated with a separate process that performs the data-link, network, transport, and presentation layer functionality for that connection. Protocol tasks are divided into four inter-connected Modules, corresponding to the data-link, network, transport, and presentation layers in the ISO OSI communication model. Data-link processing is performed in the CP_DLP Module. This Module uses its read-side svc method to (1) transform network messages into the canonical internal message format that is processed by higher-level components in a Stream and (2) demultiplex incoming messages onto the appropriate transport layer connection.[4] Once a message has been demultiplexed onto a connection, all that connection's context information is directly accessible within the address space of the associated process. This is beneficial since (1) pointers to messages may be passed between protocol layers via simple procedure calls (rather than using more complicated and costly interprocess communication mechanisms used for Layer Parallelism process architecture), (2) cache affinity properties may be preserved since messages are processed largely within a single PE cache, and (3) minimal internal locking is required within a connection. Therefore, a process may operate on its connection's messages without incurring additional demultiplexing, synchronization, and context switching overhead. The CP_IP, CP_TCP, and CP_XDR Modules all perform their processing synchronously in their respective put methods.

• **Message Parallelism:** Figure 4 (2) illustrates a message-based process architecture for the connection-oriented protocol stack. When an incoming message arrives, it is handled by the MP_DLP::svc method, which manages a pool of pre-spawned threads. Each message is associated with a separate thread that escorts the message synchronously through a series of inter-connected Queues in a Stream. Each layer of the protocol stack performs its protocol functions and then makes

[4]The connection-oriented implementation of Connectional Parallelism performs "eager demultiplexing" via a packet filter at the data-link layer.

an upcall to the next adjacent layer in the protocol stack by invoking the `Queue::put` method in that layer. The `put` method executes the protocol tasks associated with its layer. For instance, the `MP_TCP::put` method utilizes mutual exclusion (mutex) objects that serialize access to per-connection control blocks as separate messages from the same connection ascend the protocol stack in parallel.

The connectionless message-based protocol stack is structured in a similar manner, though it performs the simpler set of UDP functionality. Unlike the `MP_TCP::put` method, the `MP_UDP::put` method handles each message concurrently and independently, without explicitly preserving inter-message ordering. This reduces the number of synchronization operations required to locate and update shared resources, which improves performance.

4 Communication Subsystem Performance Experiment Results

This section describes experiments that measure the performance impact of different combinations of the protocol stacks and process architectures described above. The multi-processor platform and the measurement tools used in the experiments are also discussed.

4.1 Multi-processor Platform

All experiments were conducted on an otherwise idle Sun 690MP SPARCserver, which contains 4 SPARC 40 MHz processing elements (PEs), each capable of performing at 28 MIPs. The memory bandwidth of the SPARCserver platform was measured at approximately 150 Mbits/sec, which represents an upper limit on protocol processing throughput. Protocol processing throughput is also significantly affected by context switching and synchronization overhead exhibited by the different task-based and message-based process architectures. The costs of context switching and synchronization overhead in the SPARCserver platform are described below.

The operating system used for the experiments is release 5.3 of SunOS, which provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel [6]. All the process architectures in these experiments execute protocol tasks in separate *unbound* threads multiplexed over 1, 2, 3, or 4 SunOS *lightweight processes* (LWPs) within a process. SunOS 5.3 maps each LWP directly onto a separate kernel thread. Since kernel threads are the units of PE scheduling and execution in SunOS, this mapping enables multiple LWPs (each executing protocol processing tasks in an unbound thread) to run in parallel on the SPARCserver's PEs.

Rescheduling and synchronizing a SunOS LWP involves a kernel-level context switch. The time required to perform a context switch between two LWPs was measured to be approximately 30 *u*secs. During this time, the OS performs system-related overhead (such as flushing register

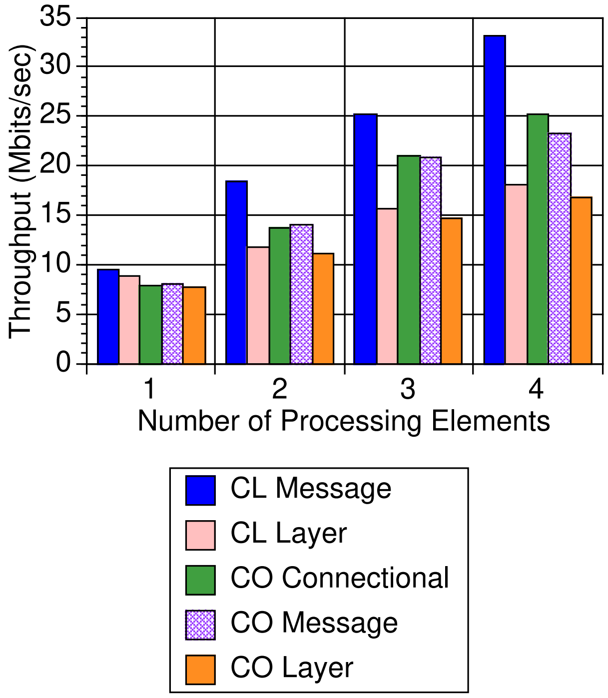

Figure 5: Process Architecture Throughput

windows, instruction and data caches, instruction pipelines, and translation lookaside buffers) on the PE and therefore does not process protocol tasks. Measurements also revealed that it requires approximately 2 *u*secs to acquire or release a `Mutex` object implemented using a SunOS spin-lock. Likewise, measurements indicated that approximately 90 *u*secs are required to synchronize two LWPs using `Condition` objects implemented using SunOS sleep-locks. The larger amount of overhead for the `Condition` object operations compared with the `Mutex` object operations occurs from the more complex locking algorithms involved, as well as the additional context switching incurred by SunOS sleep-locks.

4.2 Measurement Results

This section presents results obtained by measuring the data reception portion of the connection-oriented and connectionless protocol stacks implemented using the Layer Parallelism task-based process architecture and the Connectional Parallelism and Message Parallelism message-based process architectures. Three types of measurements were obtained for each combination of process architecture and protocol stack: *total throughput*, *context switching overhead*, and *synchronization overhead*.

Total throughput was measured by holding the protocol functionality, application traffic patterns, and network interfaces constant and systematically varying the process architecture to determine the resulting performance impact. Each benchmarking session consisted of transmitting 10,000 4 Kbyte messages through an extended version of the widely available `ttcp` protocol benchmarking tool. The original `ttcp` tool measures the processing resources and overall user and system time required to transfer data between a transmitter process and a receiver process communicating via TCP or UDP. 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 enhanced to allow a user-specified number of communicating applications to be measured simultaneously. This feature measured the impact of multiple connections on the performance of process architectures (the connection-oriented process architecture tests were run using 4 connections). The `ttcp` tool was also modified to use the `ASX`-based protocol stacks configured according to the process architectures described in Section 4.2. To measure the impact of parallelism on throughput, each test was run using 1, 2, 3, and 4 PEs. Furthermore, each test was performed multiple times to detect the amount of spurious interference incurred from other internal OS tasks (the variance between test runs proved to be insignificant).

Context switching and synchronization measurements were obtained to help explain differences in the throughput results. These metrics were obtained from the SunOS 5.3 `/proc` file system, which records the number of voluntary and involuntary context switches incurred by threads in a process, as well as the amount of time spent waiting to obtain and release locks on `Mutex` and `Condition` objects.

Figure 5 illustrates throughput (measured in Mbits/sec) as a function of the number of PEs for the task-based and message-based process architectures used to implement the connection-oriented (CO) and connectionless (CL) protocol stacks.[5] The results in this figure indicate that increasing the number of PEs improves throughput for all the process architectures. However, the message-based process architectures significantly outperformed their task-based counterparts as the number of PEs increased from 1 to 4. For example, the performance of the connection-oriented task-based process architecture was only slightly better using 4 PEs (approximately 16 Mbits/sec, or 1.92 milliseconds per-message processing time) than the message-based process architecture was using 2 PEs (14 Mbits/sec, or 2.3 milliseconds per-message processing time). Moreover, if a larger number of PEs had been available, it appears likely that the performance improvement gained from parallel processing in the task-based process architectures would have leveled off sooner than the message-based tests due to the higher rate of growth for context switching and synchronization shown in Figure 6 and Figure 7.

The Connection Parallelism process architecture exhibited the highest levels of throughput for the connection-oriented protocol stacks when the number of PEs equaled the number of connections. The major limitation with Connectional

[5] The Connectional Parallelism process architecture does not support the connectionless protocol stack.

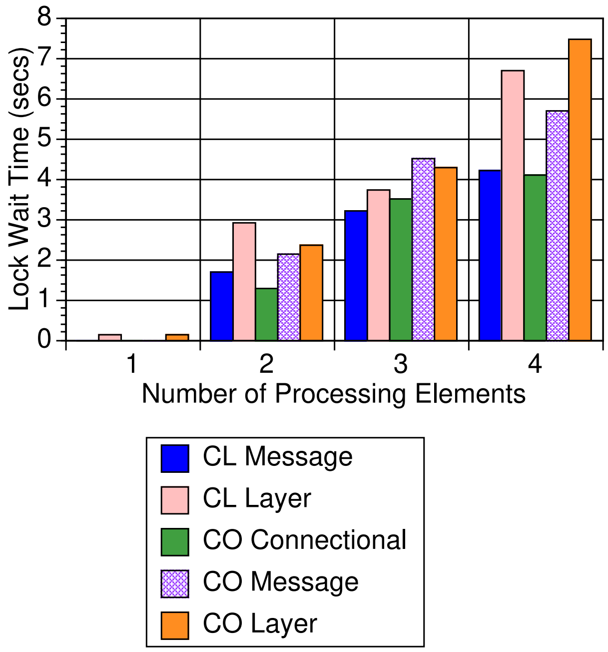

Figure 7: Process Architecture Locking Overhead

Parallelism, however, is that it only utilizes parallelism to improve *aggregate* end-system performance since each individual connection still executes sequentially. In contrast, Message Parallelism also utilizes multiple PEs effectively for a single connection.

Figure 6 illustrates the number of *involuntary* and *voluntary* context switches incurred by the process architectures measured in this study. An involuntary context switch occurs when the OS kernel preempts a running thread. For example, the OS preempts running threads periodically when their LWP time-slice expires in order to schedule other threads to execute. A voluntary context switch is triggered when a thread puts itself to sleep until certain resources (such as I/O devices or synchronization locks) become available. For example, when a protocol task attempts to acquire a resource that may not become available immediately (such as obtaining a message from an empty list of messages in a `Queue`), the protocol task puts itself to sleep by invoking the `wait` method of a `Condition` object. This action causes the OS kernel to preempt the current thread and perform a context switch to another thread that is capable of executing protocol tasks immediately.

As shown in Figure 6, The Layer Parallelism task-based process architectures exhibited slightly higher levels of involuntary context switching than the message-based process architectures. This is due mostly to the fact that the Layer Parallelism tests required more time to process the 10,000 messages and were therefore pre-empted a greater number of times. Furthermore, the task-based process architectures also incurred significantly more voluntary context switches, which accounts for the substantial improvement in overall throughput exhibited by the message-based process architec-

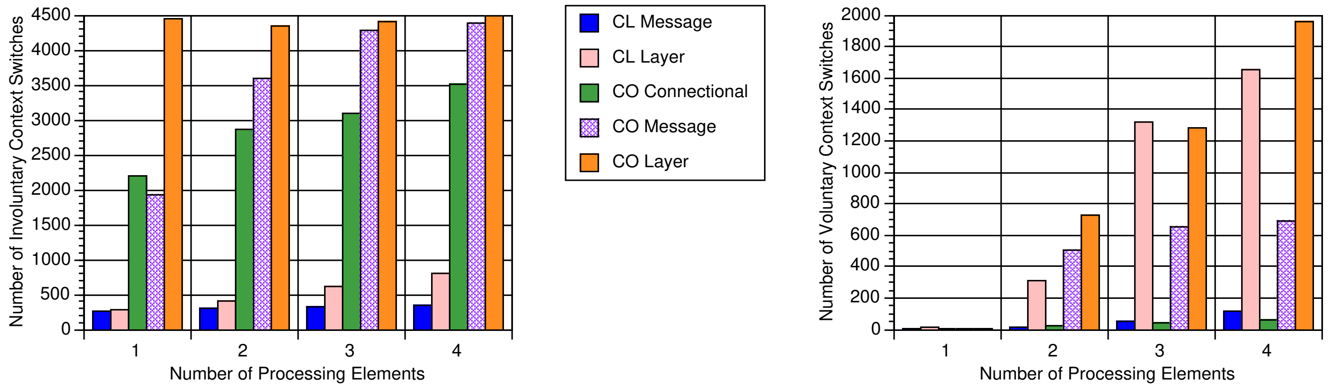

Figure 6: Process Architecture Context Switching Overhead

tures. The primary reason for the increased context switching is that the locking mechanisms used by the message-based process architectures utilize adaptive spin-locks (which rarely trigger a context switch), rather than the sleep-locks used by task-based process architectures (which *do* trigger a context switch). Note that the Connectional Parallelism process architecture incurred the least amount of context switching for the connection-oriented protocol stacks.

Figure 7 indicates the amount of execution time that the `/proc` metrics reported as being devoted to waiting to acquire and release locks in the connectionless and connection-oriented benchmark programs. As with context switching benchmarks, the message-oriented process architectures incurred considerably less synchronization overhead, particularly when 4 PEs were used. As with context switching, the spin-locks used by message-based process architecture reduce the amount of time spent synchronizing, in comparison with the sleep-locks used by the task-based process architectures.

5 Concluding Remarks

Despite an increase in the availability of operating system and hardware platforms that support networking and parallel processing, developing communication subsystems that effectively utilize parallel processing remains a complex and challenging task. A key aspect of communication subsystem performance involves the type of process architecture selected to structure parallel processing of protocol tasks. Measurement results reported in this paper indicate that task-based process architectures incur much higher levels of context switching and synchronization overhead on a shared memory platform, which significantly reduces performance. Conversely, the message-based process architectures (particularly Connectional Parallelism) incur much less context switching and synchronization, and therefore exhibit higher performance.

The `ASX` framework contributed to these performance experiments by helping to decouple the protocol-specific functionality from the underlying of process architecture. This decoupling increased reuse and simplified development, configuration, and experimentation with parallel protocol stacks. Components in the `ASX` framework are freely available via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z`. This distribution contains complete source code, documentation, and example test drivers for the C++ components. Components in the `ASX` framework have been ported to both UNIX and Windows NT. The `ASX` framework is currently being used in a number of commercial products including the AT&T Q.port ATM signaling software product, the Ericsson EOS family of PBX monitoring applications, and the network management portion of the Motorola Iridium mobile communications system.

References

[1] Mats Bjorkman and Per Gunningberg, "Locking Strategies in Multiprocessor Implementations of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (San Francisco, California), ACM, 1993.

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

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

[4] J. Jain, M. Schwartz, and T. Bashkow, "Transport Protocol Processing at GBPS Rates," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 188–199, ACM, Sept. 1990.

[5] 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.

[6] 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.

[7] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young, "Mach Threads and the Unix Kernel: The Battle for Control," in *Proceedings of the USENIX Summer Conference*, USENIX Association, August 1987.

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

[9] 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.

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

[11] 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.

[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] 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.

[15] D. C. Schmidt and T. Suda, "The ADAPTIVE Service eXecutive: an Object-Oriented Architecture for Configuring Concurrent Distributed Applications," in *Proceedings of the* 8th *International Working Conference on Upper Layer Protocols, Architectures, and Applications*, Barcelona, Spain: North-Holland, June 1994.

[16] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

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

[18] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in *Proceedings of the* 2nd *USENIX C++ Conference*, pp. 191–203, USENIX Association, April 1990.

[19] 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.