Measuring the Performance of Parallel Message-based Process Architectures

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]

An earlier version of this paper appeared at the IEEE INFOCOM Conference in Boston, MA, April, 1995.

Abstract

Message-based process architectures are widely regarded as an effective method for structuring parallel protocol processing on shared memory multi-processor platforms. A message-based process architectures is formed by binding one or more processing elements with the data messages and control messages received from applications and network interfaces. In this architecture, parallelism is achieved by simultaneously escorting multiple messages on separate processing elements through a stack of protocol tasks. This paper reports performance results from an empirical comparison of a connection-oriented protocol stack implemented using two different message-based process architectures. These performance experiments measure the throughput, context switching, and synchronization exhibited by the two parallel process architectures on a shared memory multi-processor platform. The experimental results demonstrate the extent to which the selection of a parallel process architecture affects protocol stack performance.

Keywords: Parallelizing Protocols for High-Speed Networks, Testbeds and Measurements

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 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 management, timer-based and I/O-based event invocation, 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 [1].

Executing protocol stacks in parallel on multi-processor platforms is a promising technique for increasing protocol processing performance [2]. Significant increases in performance are possible, however, only if the speed-up obtained from parallelism outweights 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, a context switch may require dozens to hundreds of instructions to flush register windows, memory caches, instruction pipelines, and translation look-aside buffers [3]. Synchronization overhead arises from locking mechanisms that serialize access to shared objects (such as message buffers, message queues, protocol connection records, and demultiplexing maps) used during protocol processing [4].

Message-based process architectures are widely regarded as an effective method for structuring parallel protocol processing on shared memory multi-processor platforms [5, 6, 7, 4, 8]. A message-based process architecture is formed by binding one or more PEs with data messages and control messages received from applications and network interfaces. In this architecture, parallelism is achieved by simultaneously escorting multiple 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 different types of message-based process architectures. Existing studies have generally selected a single message-based process architecture and studied it in isolation. Moreover, these studies have been conducted on different OS and hardware platforms, using different protocol stacks and implementation techniques, which makes it difficult to meaningfully compare results. This paper reports the performance results of systematic, empirical comparisons of two message-based process architectures implemented on a widely-available shared memory multi-processor plat-

[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, and Hewlett Packard (HP).

form. The performance experiments were conducted using an object-oriented framework [9] that supports controlled experiments with alternative types of parallel process architectures. The framework controls for a number of relevant confounding factors (such as protocol functionality, concurrency control strategies, application traffic characteristics, and network interfaces). This enables precise measurement of the performance impact of using different process architectures to parallelize communication protocol stacks.

The paper is organized as follows: Section 2 outlines the two types of message-based process architectures and classifies related work accordingly; Section 3 examines empirical results from experiments performed using the framework; and Section 4 presents concluding remarks.

2 Message-based Process Architecture Examples

Message-based process architectures associate processes[2] with messages, rather than with protocol layers or protocol tasks. Two common examples of message-based process architectures are *Connectional Parallelism* and *Message Parallelism*. The primary difference between these process architectures involves the point at which messages are demultiplexed onto a process. 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. As shown in Figure 1 (1), connections C_1, C_2, C_3, and C_4 reside in separate processes 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 the application process and use it to escort messages down a protocol stack [11]. For incoming messages, a network interface or packet filter typically performs demultiplexing operations to determine the correct process to associate with each message.

Connectional Parallelism is particularly useful for server applications that handle many active connections simultaneously. The advantages of Connectional Parallelism are (1) inter-layer communication overhead is reduced (since moving messages between protocol layers does not require a context switch), (2) synchronization and communication overhead is relatively low within a given connection (since synchronous *intra-process* downcalls and upcalls may be used to communicate between the protocol layers), and (3) the amount of available parallelism is determined dynamically since it is a function of the number of active connections (rather than a function of the number of layers or tasks, which are determined statically). One disadvantage with Connectional Parallelism is the difficulty of PE load balancing. For example, a highly active connection may swamp its PE with messages, leaving other PEs tied up at less active or idle connections.

Message Parallelism associates a separate process with every incoming or outgoing message. As illustrated in Figure 1 (2), a process 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.

The advantages of Message Parallelism are similar to those for Connectional Parallelism. In addition, the amount of available parallelism may be higher since it depends on the number of messages exchanged, rather than the number of connections. Likewise, processing loads may be balanced more evenly between PEs since each incoming message may be dispatched to an available PE. The primary disadvantage of Message Parallelism is the overhead resulting from synchronization and mutual exclusion primitives required to serialize access to shared resources (such as memory buffers and control blocks that reassemble protocol segments addressed to the same higher-layer connection).

A number of studies have investigated the performance characteristics of message-based process architectures. All these studies utilized shared memory platforms. [6] measured the performance of the TCP, UDP, and IP protocols using Message Parallelism on a uniprocessor platform running the *x*-kernel. [4] 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, [8] examined performance issues in parallelizing TCP-based and UDP-based protocol stacks using a different multi-processor version of the *x*-kernel. [10] 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. [12] measured the performance of the TCP/IP protocol stack using Connectional Parallelism in a multi-processor version of System V STREAMS.

The research presented in this paper extends existing work by measuring the performance of two message-based process architectures in a controlled environment. Furthermore, our experiments report the impact of both context switching and synchronization overhead on communication subsystem performance. 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 [13].

[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 or threads) has also been used to denote the same basic concepts. Our use of the term process is consistent with definitions presented in [10, 11].

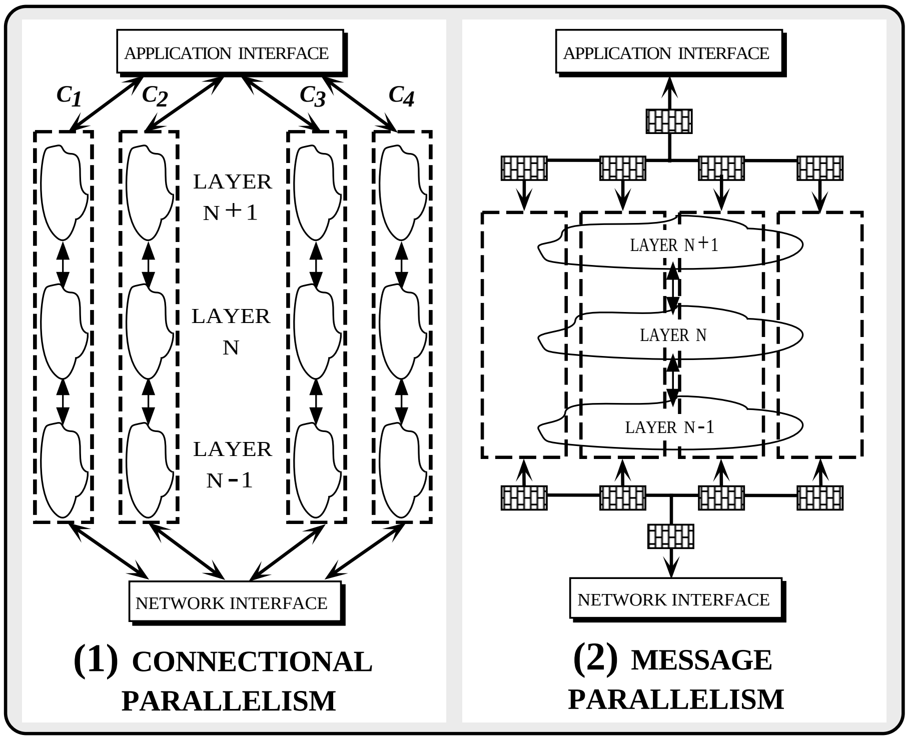


Figure 1: Message-based Process Architecture Examples

3 Communication Subsystem Performance Experiments

This section presents performance results obtained by measuring the data reception portion of a connection-oriented protocol stack implemented using the *Message-Parallelism* and *Connectional Parallelism* process architectures. In this section, the multi-processor platform and the measurement tools used in the experiments, communication protocol stack and process architectures, and performance results are presented.

3.1 Multi-processor Platform

All experiments were conducted on an otherwise idle Sun SPARCcenter 2000 shared memory symmetric multiprocessor. This SPARCcenter platform contained 640 Mbytes of RAM and 20 superscalar SPARC 40 MHz processing elements (PEs), each rated at approximately 135 MIPs. The operating system used for the experiments was release 5.3 of SunOS. SunOS 5.3 provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel on the SPARCcenter platform [14]. Both the process 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 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 SPARCserver's 20 PEs.

The memory bandwidth of the SPARCserver platform is 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 [3]. These activities take 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.

Synchronization operations were implemented using SunOS adaptive spin-locks (called `mutex` objects [14]). Adaptive spin-locks ensure mutual exclusion by using an atomic hardware instruction that polls a designated memory location until one of the following conditions occur:

- The value at this location is changed by the process that currently owns the lock. This signifies that the lock has been released and may now be acquired by the spinning process.
- The process that is holding the lock goes to sleep. At this point, the spinning process also puts itself to sleep to avoid unnecessary polling [14]. When contention for a `mutex` is low, acquiring an adaptive spin-lock rarely triggers a context switch.

On a multi-processor, the overhead incurred by a spin-lock is relatively minor. Hardware-based polling does not cause contention on the system bus since it only affects the local PE caches of processes that are spinning on a `mutex` object. 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.

3.2 The Structure and Functionality of the Communication Protocol Stack

The protocol stack investigated in the experiments was based on the connection-oriented TCP transport protocol. This protocol stack 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 [7, 13].

The connection-oriented communication protocol stack in this study was developed using components provided by the ADAPTIVE Server eXecutive (`ASX`) framework [9]. The `ASX` framework contains an integrated set of object-oriented components that facilitate experimentation with process architectures on shared memory multi-processor platforms.

Components in the `ASX` framework 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's 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 distinct layers of functionality. Each layer implements a cluster of related protocol-specific tasks (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 tasks.

The `ASX` framework employs a number of object-oriented design techniques (such as design patterns [15] and hierarchical decomposition [16]) and object-oriented language features (such as abstract classes, 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 layer 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 the `put` and `svc` methods[3] 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 layer (*e.g.,* the transport layer, network layer, data-link layer, etc.)

[3]A `Queue`'s `put` method borrows the thread of control from an adjacent `Queue` that invoked it. In contrast, a `Queue`'s `svc` method is invoked by a separate process associated with the `Queue`. The `put` method synchronously performs protocol tasks that require immediate processing, whereas the `svc` method performs protocol tasks that may be deferred and run asynchronously.

The `ASX` framework incorporates concepts from several other modular communication frameworks such as System V STREAMS [17], the *x*-kernel [6], and the Conduit [18] (a survey of these and other communication frameworks appears in [1]). These frameworks contain tools that support the flexible configuration of communication subsystems. These tools support the interconnection of building-block protocol components (such as message managers, timer-based event dispatchers, and connection demultiplexers [6] and other reusable protocol mechanisms [19]) to form protocol stacks.

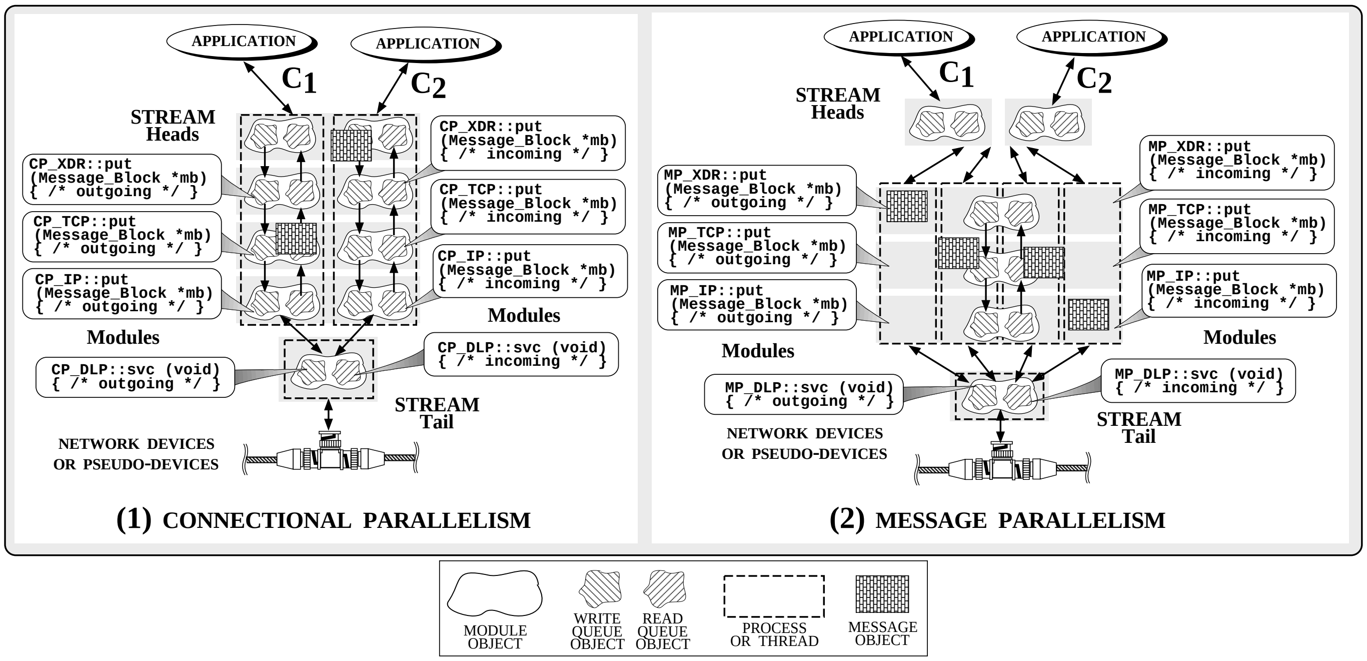
In addition to supplying building-block protocol components, the `ASX` framework also extends the features provided by existing communication frameworks. In particular, `ASX` provides components that decouple protocol-specific functionality from the following structural and behavioral characteristics of a communication subsystem:

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

In this study, inheritance and parameterized types were used to hold protocol stack functionality constant, while the process 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 `Queue` abstract class. The synchronization and demultiplexing mechanisms required to implement different process architectures were parameterized using C++ template class arguments. These templates were instantiated based upon the type of process architecture being tested.

Data-link layer processing in the 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 `Queue` 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 process architecture experiments was simulated with a single-copy pseudo-device driver operating in loop-back mode. This approach is consistent with those used in [4, 7, 8].

The transport and network layers of the protocol stack was based on the TCP/IP implementations in the BSD 4.3 Reno release. The 4.3 Reno TCP implementation contains



```
CP_XDR::put
(Message_Block *mb)
{ /* outgoing */ }
CP_TCP::put
(Message_Block *mb)
{ /* outgoing */ }
CP_IP::put
(Message_Block *mb)
{ /* outgoing */ }
CP_DLP::svc (void)
{ /* outgoing */ }
CP_XDR::put
(Message_Block *mb)
{ /* incoming */ }
CP_TCP::put
(Message_Block *mb)
{ /* incoming */ }
CP_IP::put
(Message_Block *mb)
{ /* incoming */ }
CP_DLP::svc (void)
{ /* incoming */ }
MP_XDR::put
(Message_Block *mb)
{ /* outgoing */ }
MP_TCP::put
(Message_Block *mb)
{ /* outgoing */ }
MP_IP::put
(Message_Block *mb)
{ /* outgoing */ }
MP_DLP::svc (void)
{ /* 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)
{ /* incoming */ }
```

Figure 2: ASX-based Message-based Process Architectures

the TCP header prediction enhancements, as well as the TCP slow start algorithm and congestion avoidance features. The TCP transport protocol was configured into the ASX framework via the 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.

3.3 Structure of the Process Architectures

This section outlines the structure of the message-based process architectures used to parallelize the connection-oriented protocol stack described above.

3.3.1 Connectional Parallelism

The protocol stack depicted in Figure 2 (1) illustrates an ASX-based implementation of the Connectional Parallelism (CP) process architecture outlined in Section 2. Each process 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 at the data-link layer. Thus, the CP_DLP Module uses its read-side svc method to 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, a pointer to a message is passed between protocol layers.

3.3.2 Message Parallelism

Figure 2 (2) depicts the Message Parallelism (MP) process 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 `Queues` that form a protocol stack. Each layer of the protocol stack performs the protocol tasks defined by its `Queue`. When these tasks are complete, an upcall may be 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 `Queue`. This `put` method borrows the thread of control from its caller and executes the protocol tasks associated with its layer.

The connection-oriented `MP_TCP::put` method utilizes several `mutex` synchronization objects. 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.

The ASX-based Connectional Parallelism and Message Parallelism implementations optimize message management by using SunOS thread-specific storage [14] to buffer messages as they flow through a protocol stack. This optimization leverages off the cache affinity properties of the SunOS shared memory multi-processor. In addition, it minimizes the cost of synchronization operations used to manage the global dynamic memory heap.

3.4 Measurement Results

This section presents results obtained by measuring the data reception portion of the connection-oriented protocol stack developed using the process architectures described in Section 3.3. Three types of measurements were obtained for each combination of process architecture and protocol stack: *average throughput*, *context switching overhead*, and *synchronization overhead*. Average throughput measured the impact of parallelism on protocol stack performance. Context switching and synchronization measurements were obtained to help explain the variation in the average throughput measurements.

Average throughput was measured by holding the protocol functionality, application traffic, and network interfaces constant, while systematically varying the process 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 throughput performance). 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 internal OS process management tasks did not perturb the results.

Various statistics were collected using an extended version

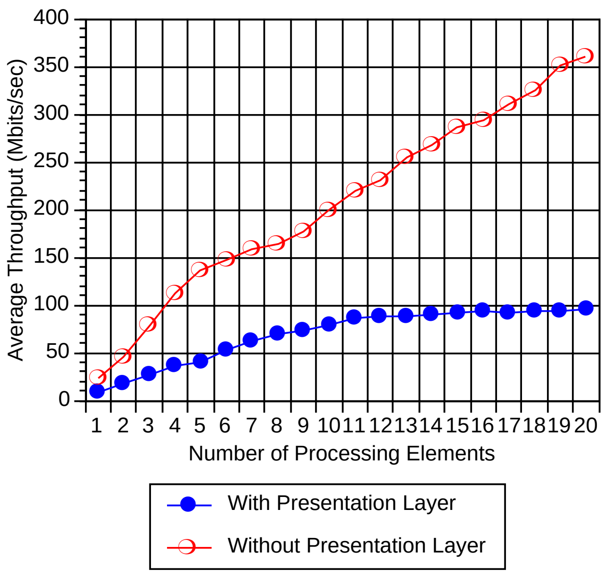


Figure 3: Connection-oriented Connectional Parallelism Throughput

of the widely available `ttcp` protocol benchmarking tool. The `ttcp` tool measures the amount of OS processing resources, user-time, and system-time required to transfer data between a transmitter process and a receiver process. 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 process architectures described in Section 3.3. 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 process architectures.

3.4.1 Throughput Measurements

Figures 3 and 4 depict the average throughput for the Connectional Parallelism and Message Parallelism process architectures used to implement the connection-oriented protocol stack. Each test run for these connection-oriented process 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 5 and 6 indicate the relative speedup that resulted from successively adding another PE to each process architecture. Relative speedup is computed by dividing the average aggregated throughput for n PEs (shown in Figures 3 and 4, where $1 \leq n \leq 20$) by the average

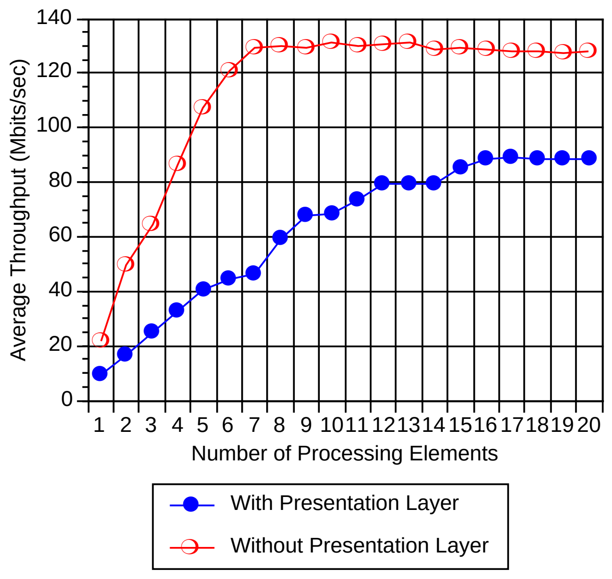


Figure 4: Connection-oriented Message Parallelism Throughput

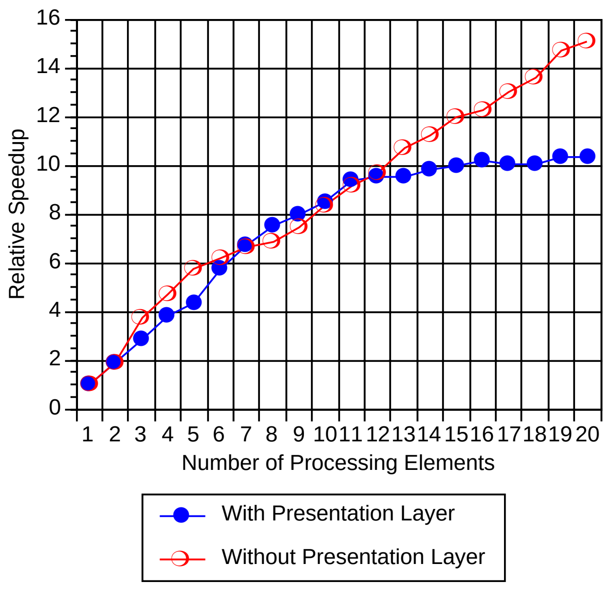


Figure 5: Relative Speedup for Connection-oriented Connectional Parallelism

throughput for 1 PE.

The results from Figures 3, 4, 5, and 6 indicate that increasing the number of PEs generally improves the average throughput in the message-based process architectures. Connection-oriented Connectional Parallelism exhibited the highest performance, both in terms of average throughput and in terms of relative speedup. The average throughput of Connectional parallelism with presentation layer processing peaks at approximately 100 Mbits/sec (shown in Figure 3). 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 5, 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.

The average throughput achieved by connection-oriented Message Parallelism without presentation layer processing peaks at just under 130 Mbits/sec (shown in Figure 4). 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 6) flattens out after 8 CPUs. This speedup curve flattens out when presentation layer processing is omitted due to increased contention for shared synchronization objects at the transport layer. This synchronization overhead is discussed further in Section 3.4.3. In contrast, the relative speedup of connection-oriented Message Parallelism with presentation layer processing grows steadily from 1 to 20 PEs. This behavior suggests that connection-oriented Message Parallelism benefits more from parallelism when the protocol stack contains presentation layer processing. This finding is consistent with those reported in [8], and is related to the minimal synchronization requirements of the presentation layer (compared with the higher relative amounts of synchronization in the connection-oriented transport layer).

A limitation with Connectional Parallelism is that each individual connection executes sequentially. Therefore, Connectional Parallelism becomes most effective as the number of connections approaches the number of PEs. In contrast, Message Parallelism utilizes multiple PEs more effectively when the number of connections is much less than the number of PEs. Figure 7 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 greater than 10.

3.4.2 Context Switching Measurements

Measurements of context switching overhead were obtained by modifying the `ttcp` benchmarking tool to use the SunOS 5.3 `/proc` process file system. The `/proc` file system provides access to the executing image of each process in the system. It reports the number of voluntary and involuntary context switches incurred by SunOS LWPs within a process. Figures 8 and 10 illustrate the number of voluntary and involuntary context switches incurred by transmitting the 20,000 4 kbyte messages through the process 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

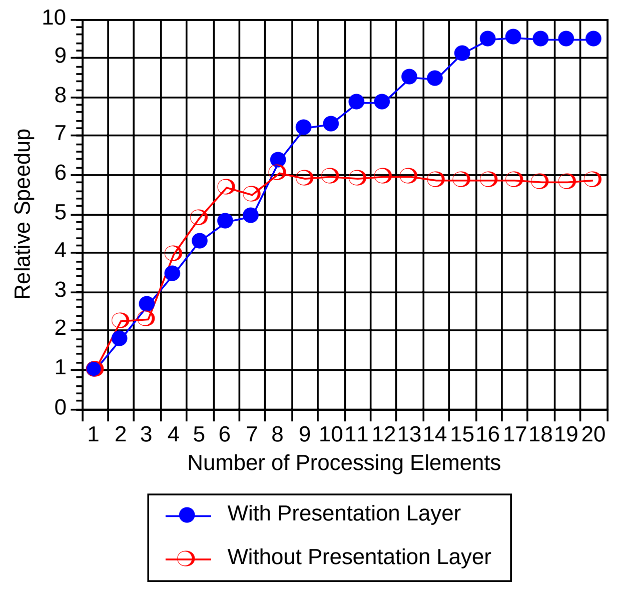


Figure 6: Relative Speedup for Connection-oriented Message Parallelism

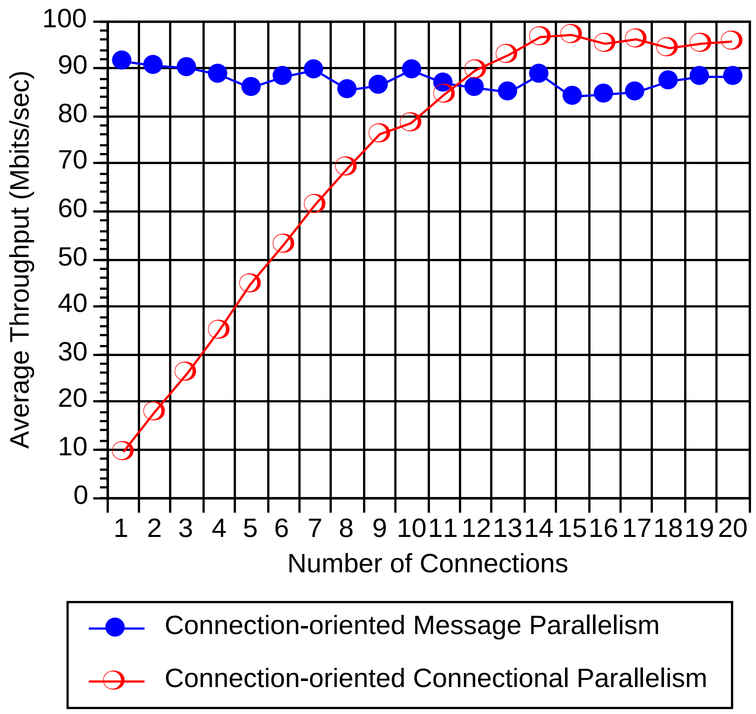


Figure 7: Comparison of Connectional Parallelism and Message Parallelism

I/O devices or synchronization locks) to become available. This 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 process architecture and protocol stack, voluntary context switching increases fairly steadily as the number of PEs increase from 1 through 20 (shown in Figures 8 and 10).

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 8 and 10 remains fairly consistent as the number of PEs increase.

Connectional Parallelism incurred the lowest levels of context switching for the connection-oriented protocol stack, compared with Message Parallelism. In the Connectional Parallelism process 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. Thus, a thread of control in Connectional Parallelism processes its connection's messages without incurring additional context switching overhead.

3.4.3 Synchronization Measurements

Measurements of synchronization overhead were collected to determine the amount of time spent acquiring and releasing locks on `mutex` objects during protocol processing on the 20,000 4 kbyte messages. Unlike context switches, the SunOS 5.3 `/proc` file system does not maintain accurate metrics on synchronization overhead. Therefore, these measurements were obtained by bracketing the `mutex` operations with calls to the `gethrtime` system call. This system call uses the SunOS 5.3 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by the `gethrtime` system call is not subject to resetting or drifting since it is not correlated with the current time of day.

Figures 9 and 11 indicate the total time (measured in msecs) used to acquire and release locks on `mutex` objects. These tests were performed using Connectional Parallelism and Message Parallelism to implement a connection-oriented protocol stack that contained data-link, network, transport, and presentation layer functionality. Connection-oriented Connectional Parallelism (shown in Figure 9) 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 11. 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.

The synchronization overhead incurred by connection-oriented Message Parallelism (shown in Figure 11) peaked at just over 6,000 msecs. Moreover, the rate of growth increased fairly steadily as the number of PEs increased from 1

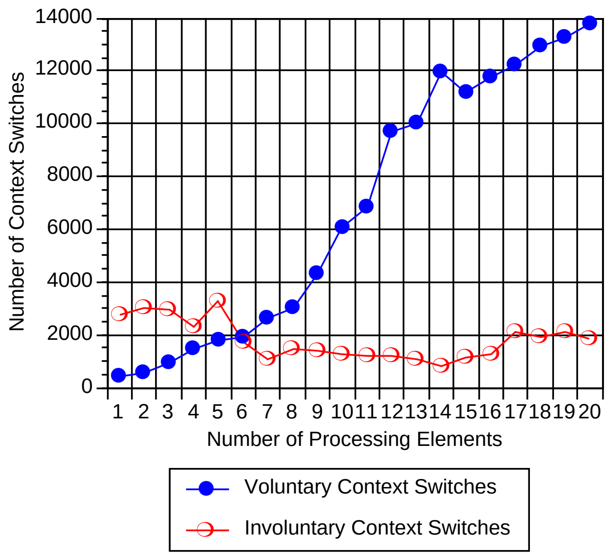


Figure 8: Connection-oriented Connectional Parallelism Context Switching

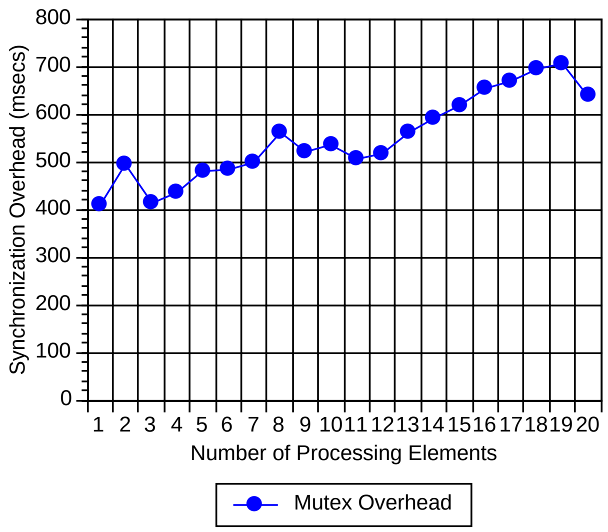


Figure 9: Connection-oriented Connectional Parallelism Synchronization Overhead

to 20. This behavior occurs from the lock contention caused by `mutex` objects that serialize access to connection demultiplexing mechanisms at the transport layer.

3.4.4 Summary of Observations

The following observations resulted from conducting performance experiments on message-based process architectures for a connection-oriented protocol stack:

- Connectional Parallelism is more suitable than Message Parallelism as 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. In addition, unlike Connectional Parallelism, Message Parallelism is suitable for connectionless applications.
- Connection-oriented Message Parallelism benefits more from parallelism when the protocol stack contains presentation layer processing. As shown in Figure 6, the speedup 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. This behavior results from the relatively low amount of synchronization overhead associated with parallel processing at the presentation layer. In contrast, the relative speedup for Connectional Parallelism without presentation layer processing continues to increase steadily up to 20 PEs (shown in Figure 5). Connectional Parallelism performs well in this case due to its low levels of synchronization and context switching overhead.
- The relative cost of synchronization operations has a substantial impact on process architecture performance. On the SPARCcenter 2000 shared memory multi-processor running SunOS 5.3, the Connectional Parallelism and Message Parallelism process architectures benefit from their use of inexpensive adaptive spin-locks. 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, it is likely that the performance of the message-based process architectures would have decreased significantly.

4 Concluding Remarks

This paper describes performance measurements obtained by using the ASX framework to parallelize a connection-oriented protocol stack implemented with Connectional Parallelism and Message Parallelism process architectures. The ASX framework provides an integrated set of object-oriented components that facilitate experimentation with different types of process architectures on multi-processor platforms. By decoupling the protocol-specific functionality from the underlying process architecture, the ASX framework increased component reuse and simplified the development, configuration, and experimentation with 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. Compared with Connectional Parallelism, the Message Parallelism process architecture exhibited rela-

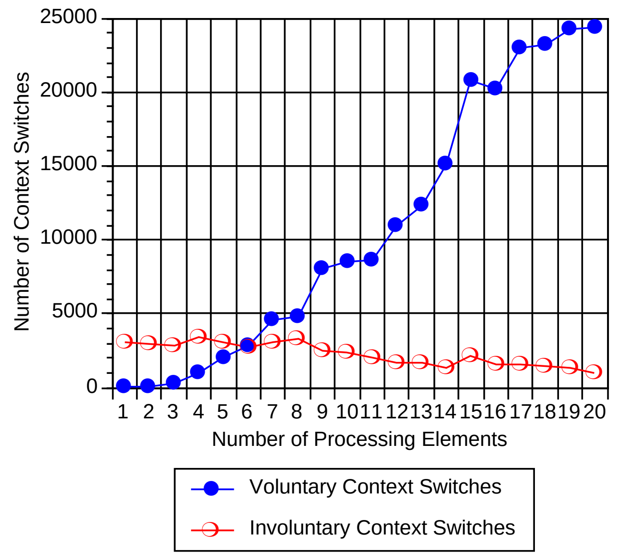


Figure 10: Connection-oriented Message Parallelism Context Switching

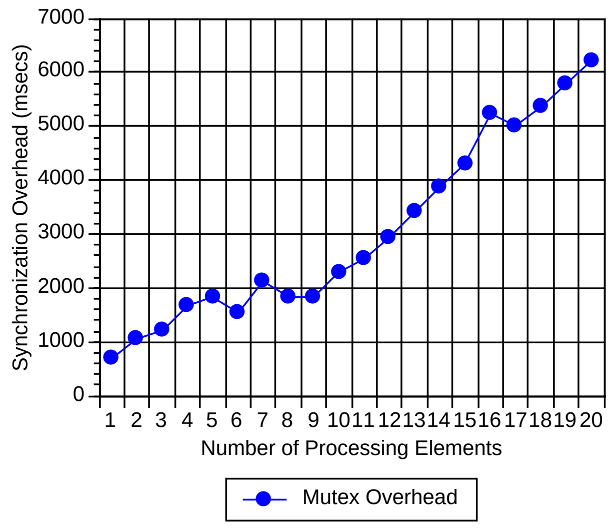


Figure 11: Connection-oriented Message Parallelism Synchronization Overhead

tively high levels of context switching and synchronization overhead. This overhead helps to account for the higher performance exhibited by Connectional Parallelism. In general, the results from these experiments underscore the importance of the process architecture on parallel communication subsystem performance.

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 ASX 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 [15], as well as the network management subsystems in the Ericsson EOS project [9] and the Motorola Iridium global personal communications system.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

[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] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, "Performance Issues in Parallelized Network Protocols," in *The Operating Systems Design and Implementation conference*, USENIX Association, November 1994.

[9] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, December 1994.

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

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

[12] 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.

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

[14] 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.

[15] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching,” in *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, August 1994.

[16] D. Batory and S. W. O’Malley, “The Design and Implementation of Hierarchical Software Systems Using Reusable Components,” *ACM Transactions on Software Engineering and Methodology*, vol. 1, pp. 355–398, Oct. 1992.

[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* 2^{nd} *USENIX C++ Conference*, pp. 191–203, USENIX Association, April 1990.

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