Fine-Grain Protocol Execution Mechanisms & Scheduling Policies on SMP Clusters

by

Babak Falsafi

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1998

Abstract

Symmetric multiprocessor (SMP) clusters are emerging as the cost-effective medium- to large-scale parallel computers of choice, exploiting the superior cost-performance of SMP desktops and servers. These machines implement communication among SMP nodes by sending/receiving messages through an interconnection network. Many applications and systems use a variety of software protocols to coordinate this communication. As such, protocol performance can significantly impact communication time and overall system performance.

This thesis proposes and evaluates techniques to improve fine-grain software protocol performance. Rather than provide embedded network interface processors, some systems schedule and execute the protocol code on the SMP processors to reduce hardware complexity and cost. This thesis evaluates *when* it is beneficial to dedicate one or more processors in every SMP to always execute the protocol code. Results from simulating a fine-grain software distributed shared memory (DSM) indicate that a dedicated protocol processor:

- benefits light-weight protocols much more than heavy-weight protocols;
- benefits systems with four or more processors per node;
- will also result in the best cost-performance when scheduling overheads are much higher than protocol weight.

Much like ordinary application software, the protocol code can execute either sequentially or in parallel. The central contribution of this thesis is a novel set of mechanisms, *parallel dispatch queue* (*PDQ*), for efficient parallel execution of fine-grain protocols. PDQ is based on the observation that by partitioning system resources among protocol

threads, multiple threads can execute in parallel each accessing an exclusive set of resources thereby obviating the need for synchronization.

This thesis proposes two fine-grain DSM systems—*Hurricane* and *Hurricane-1*—which execute software coherence protocols in parallel using PDQ. Hurricane achieves high performance by integrating embedded protocol processors into a network interface device. Hurricane-1 reduces cost by using SMP processors to execute the software protocol. Simulation results comparing the Hurricane systems to an all-hardware DSM implementation indicate that:

- PDQ helps significantly improve software protocol performance;
- Hurricane with four embedded processors performs as well as an all-hardware implementation;
- Hurricane-1 performs within 75% of an all-hardware implementation on average.

Acknowledgments

I dedicate this thesis to my parents Houshang Falsafi and Meimanat Lotfi. Without their guidance, moral, and financial support, I would not have made it this far. I am forever indebted to them for who I am. Thanks to my sisters Fariba and Rebecca who were always supportive of me in setting high academic and education standards. Thanks to Babak Farsaii and Mohammad Moein, my brother in laws, for being my role models in pursuing a Ph.D. Thanks to Babak for teaching me first about computers.

Many thanks to my advisor, David Wood, for teaching me how to write, how to speak, and how to think, for the many years of financial support in graduate school, and for introducing me to some of the finest microbrews in Wisconsin.

Thanks to Mark Hill for teaching me how to be strategic both in research and in my graduate career. Many thanks to Guri Sohi for teaching me how to objectively evaluate the goals and means of my research. Thanks to both Mark and Guri for advising me on how to prepare for an academic career. Thanks to Jim Goodman and Jim Smith for sitting on my thesis committee and their valuable feedback on my research.

Thanks to Scott Breach, Doug Burger, Alain Kägi, and T. N. Vijaykumar for all the time we spent together in Madison from working late night hours on Project Jihad, Friday night dinners, Atlas Pasta and Eureka Joe's lunches and coffee breaks, and all the wonderful social gatherings. Special thanks to Scott for accompanying me most to Anchora and Eureka Joe's for an espresso drink, Alain for all the wonderful ski trips, Doug for chatting with me about the game, and last but not least, Vijay for keeping me company with all the J.R. Ultimos, the Glenlivets, and the Pierre Ferrands.

Thanks to Jignesh Patel for being an excellent friend, house-mate, and cook. Thanks to Jignesh for all the Boney M. music, the Chicago trip that turned my life around, and much more.

Many thanks to Stefanos Kaxiras, Angeliki Baltoyianni, Andreas Moshovos, Vasiliki Draganigou, Ioannis Schoinas, and Leah Parks for all the wonderful time in Madison, Chicago, and The Great Dane. Thanks to Subbarao Palacharla for telling me about everything that was going on around me and I was not aware of. Thanks to John Watrous for the Chocolate Porter we brewed together.

Alvy Lebeck, Dionisios Pnevmatikatos, and Rajesh Mansharamani provided me with all the guidance I needed from senior graduate students. Thanks to Steve Reinhardt for teaching me how to write systems code and for not complaining the least bit about all the nagging I did while reading/rewriting his code. Steve provided me with the Wisconsin Wind Tunnel II which served as the experimentation testbed for all my dissertation work.

Thanks to the folks in the computer sciences department in general and the computer architecture community in particular at Wisconsin. I have never come across a group of people who can get along so well both professionally and socially.

Most of all, thanks to Anastassia Ailamaki, for all her love, support, and encouragement throughout the time I was writing and completing this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Fine-Grain Software Protocols in SMP Clusters

Advances in semiconductors fabrication processes have led to a tremendous increase in clock speeds and transistor counts in today's single-chip devices. Higher clock speeds along with novel architectural techniques exploiting the abundance of transistors in single-chip devices have improved microprocessor performance by orders of magnitude in the last decade [BG97]. In spite of the dramatic improvements in a single chip's performance, computer customers continue to demand higher performance to solve large classes of important scientific and commercial problems [AG89].

To increase performance beyond a single chip, computer designers have studied techniques to package two or more microprocessors in a single computer system. The most common form of these *multiprocessors* organizes two or more microprocessor chips on a single board and interconnects them through a system bus to a single memory system [Bel85]. Such a system is called a symmetric multiprocessor (SMP) because each processor has equal access to memory. An SMP is a cost-effective computer organization because it amortizes the cost of memory and peripheral I/O subsystems over multiple processors.

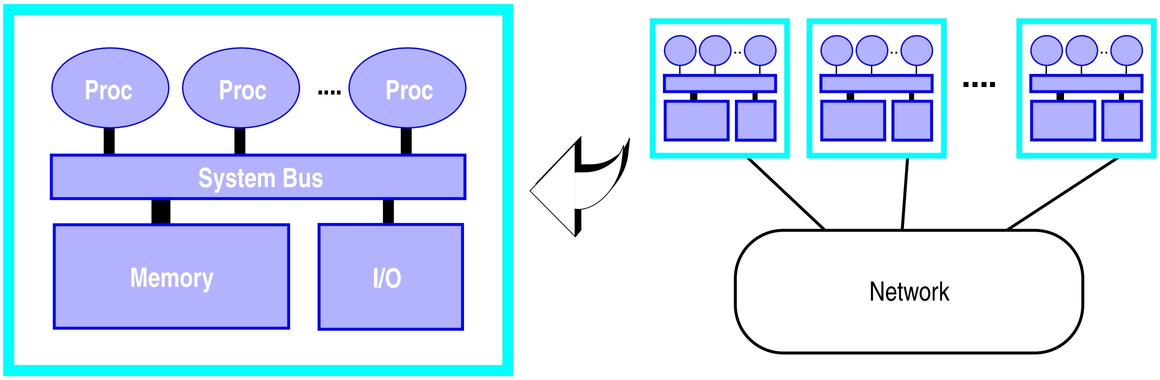


Figure 1-1. Architecture of a cluster of SMPs.

Electrical characteristics of wires in a shared system bus, however, limit the scalability of SMPs to a few tens of processors [AB86]. Rather than engineer a new computer organization with processors repackaged into custom boards, computer designers are using SMPs as building blocks to further scale the system size. The high sales volumes and economies of scales of commodity SMPs allows computer designers to build cost-effective medium- to large-scale multiprocessors. Clustering several SMPs through a high-bandwidth low-latency network (Figure 1-1) also eliminates the single bus bottleneck offering performance scalability with a large number of processors.

To program these multiprocessors, computer designers provide a variety of programming abstractions and languages. These (parallel) programming abstractions allow a processor to *compute*—i.e., produce—a data item in memory and subsequently *communicate* it—i.e., make it available—to other processors. Multiprocessors provide a variety of mechanisms for processors to communicate data among each other. Whereas processors *within* an SMP use a common (physically) shared memory to produce and communicate data, processors *across* SMPs communicate by sending messages through the interconnection network.

To schedule and coordinate communication, both applications and systems employ a variety of protocols. The SMP system bus, for instance, implements a shared-memory protocol in hardware to provide a consistent image of a single shared memory among the SMP processors. The protocols implementing communication across SMPs provide a variety of functions. At the highest level, the protocols implement application-level programming abstractions such as a client-server model in databases. Application-level protocols in turn are often programmed in lower-level abstractions such as distributed shared memory (DSM)—providing a single global address space over an SMP cluster—or simple point-to-point message passing. At the lowest level, the protocols provide message delivery services—such as checksums, reliable delivery, fragmentation and re-assembly, and flow control—to guarantee that messages are safely transferred between two SMPs.

Multiprocessor systems can implement communication protocols in either hardware or software. Hardware implementations often offer superior performance over software. Hardware protocols also implement communication transparently without involving the application or system programmer. Hardware DSM, for instance, can mimic the fine-grain shared-memory communication mechanisms of an SMP, allowing SMP programs to transparently run on a larger-scale SMP cluster.

A hardware protocol typically implements a fixed policy for coordinating communication. Although such a policy may be well-suited for some applications, it fails to meet the communication requirements of all classes of applications. Conversely, software is advantageous because it provides flexibility allowing programmers to tailor software protocols to fit the communication patterns of an application [QB97,FLR+94]. Software also serves as a suitable substrate for experimenting with complex protocols. By reducing the frequency of network messages, customized software protocols can dramatically improve an application's performance.

Vendors may also use software protocols simply because of their reduced manufacturing cost [Mei93] and shorter design times [LC96]. Communication protocols are typically implemented in the form of finite-state machines. Large and complex finite-state machines are difficult to debug [Cha97] and require enormous amounts of computing resources to even partially verify [DDHY92]. Rectifying hardware bugs also requires repeating the design and manufacturing process which increases cost.

Software protocols can also increase portability [SFH+97,JKW95,SFL+94] by providing a common high-level programming interface which systems in turn implement on various systems using platform-specific lower-level communication mechanisms. Portable interfaces allow programmers to develop and debug applications on low-cost small-scale systems and subsequently execute them on expensive large-scale machines.

Rapid improvements in networking technology, however, are dramatically reducing point-to-point message latencies between two SMP nodes [KC94]. As a result, software protocol execution is beginning to dominate the communication time. To address this problem, computer designers are studying various techniques to improve software protocol performance. Much like an ordinary application, software protocol performance can be improved in two basic ways: (i) accelerating the sequential execution, and (ii) parallelizing the execution of the protocol code.

The primary contributions of this thesis are to develop and evaluate several techniques to both accelerate sequential execution and parallelize the execution of software protocols. The thesis focuses on software protocols that implement *fine-grain* communication where two processors exchange data at small granularities of a few tens or hundreds of bytes. An example of such a protocol is a fine-grain DSM coherence protocol which transfers data among SMPs within a cluster at a cache block (e.g., 32-128 bytes) granularity. Fine-grain communication is characteristic of many important classes of scientific applications [BH86,CSBS95]. Commercial applications such as web servers, file servers, and database engines run on SMP servers using the SMP fine-grain shared-memory mechanisms. These applications can transparently run on an SMP cluster with the help of fine-grain DSM.

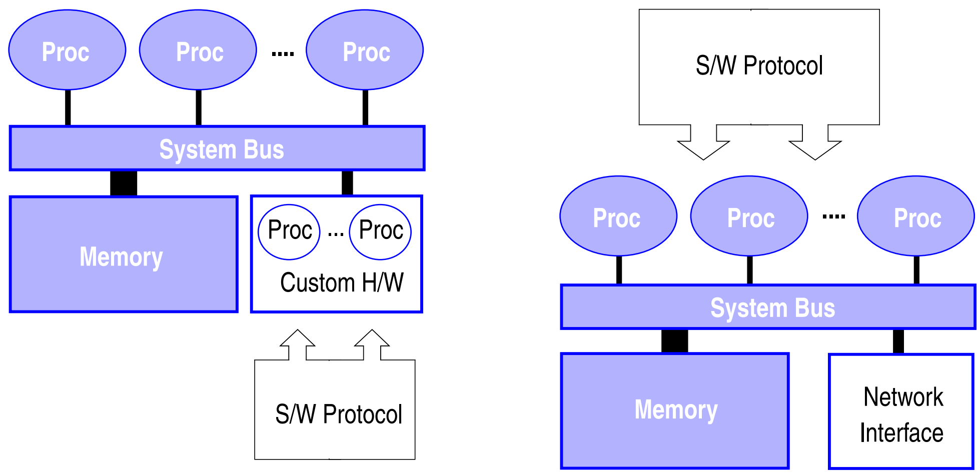
The rest of this section motivates and describes the contributions of this thesis. Section 1.1 describes the software model and explains how software protocols are invoked and executed on SMP multiprocessors. Section 1.2 briefly describes the mechanisms for parallel execution of fine-grain software protocols which is a key contribution of this thesis. Section 1.3 briefly presents the experimental results of this thesis. Section 1.4 describes the overall thesis organization.

1.1 Protocol Execution Semantics & Scheduling Policy

Fine-grain applications produce and communicate data in small granularities. As such, communication in these applications often involves frequent protocol invocations—e.g., to access remote data in distributed shared memory. Fine-grain applications also rely on quick protocol turn-around time because communication often lies on the critical path of execution. To boost a fine-grain protocol's performance, systems employ various hardware and software techniques to accelerate both protocol invocation and execution. Protocol invocation consists of detecting when communication is required and subsequently initiating the execution. Protocol execution involves running the software protocol on a processor.

At one extreme, a hardware-centric design may provide all the resources necessary to invoke and execute the software protocol in a single custom device (Figure 1-2 left). Such a device minimizes protocol invocation overhead upon arrival of network messages by executing the protocol on a device that directly interfaces to the network. This approach may significantly improve performance but also increases cost by requiring the design and manufacturing of a custom device.

At the other extreme, the system uses a commodity network interface device with little or no hardware support for protocol execution. The system software schedules and executes the protocol code on the SMP processors (Figure 1-2 right). Such a decoupling of hardware resources—e.g., processors from the network interface—increases both protocol invocation and execution overhead. Software (rather than hardware) schedules the protocol code, increasing the invocation overhead. Network accesses from processors must also



Custom H/W　　　　Commodity H/W

Figure 1-2. Software protocol execution models. Executing software protocol on (left) custom hardware, (right) SMP processors

traverse the system bus, increasing the protocol execution time. Such a system, however, can reduce cost by exploiting commodity network hardware.

Traditionally, both hardware-centric and software-centric designs for executing software protocols execute the protocol code sequentially. Hardware-centric systems provide a single embedded *protocol processor* on the network interface card [Mei93]. Software-centric systems, instead, execute the software protocol on either a uniprocessor node [HT93] or one of the SMP processors [Int93]. To reduce a software protocol's execution time, either design may parallelize the protocol's execution over multiple processors.

This thesis proposes a taxonomy for a software protocol's execution semantics: *single-threaded* and *multi-threaded* execution correspond to serial and parallel execution of protocols respectively. Multi-threaded protocol execution has been extensively studied in the context of coarse-grain networking software (e.g., TCP/IP) [Kai93]. Fine-grain protocols typically have short running times due to the fine granularity of communication. Conventional locking techniques used to parallelize coarse-grain protocols may result in excessive overheads in a fine-grain protocol's execution.

This thesis is the first to investigate mechanisms for implementing parallel fine-grain software protocols. A key contribution of this thesis is a novel set of mechanisms, *parallel dispatch queue* (PDQ), that allows efficient parallel execution of fine-grain protocols. PDQ is based on the observation that protocols can be synchronization-free if the system resources can be partitioned among multiple protocol processors, providing each processor access to an exclusive set of resources.

This thesis also investigates techniques for reducing protocol invocation overhead in the software-centric systems—i.e., systems executing the software protocol on SMP processors (Figure 1-2 right). The thesis proposes a taxonomy for two classes of software protocol scheduling policies: *dedicated* and *multiplexed* policies. A dedicated policy eliminates scheduling overhead and maximizes communication throughput by dedicating one or more SMP processors to always execute the protocol software. A multiplexed policy, however, maximizes processor utilization by allowing all processors to contribute to computation and dynamically schedules the protocol code on one or more processors when the application needs to communicate.

Much like protocol execution semantics, protocol scheduling has been extensively studied in the context of coarse-grain networking protocols [SKT96]. Because of short protocol running times, fine-grain protocol performance is also more sensitive to scheduling overhead. This thesis is the first to evaluate protocol scheduling policies for fine-grain software protocols.

1.2 PDQ: Parallel Execution of Synchronization-Free Protocols

A key contribution of this thesis is parallel dispatch queue (PDQ), a novel set of mechanisms for programming synchronization-free protocols—i.e., protocols that do not require explicit synchronization mechanisms such as locks. PDQ is based on the observation that system resources can be partitioned among the protocol processors so that each processor is provided access to an exclusive set of resources. In such a manner, the protocol code can execute free of explicit synchronization mechanisms.

PDQ requires minimal system support—e.g., network interface hardware—to classify and dispatch a protocol message or request (e.g., from the application to access remote data) to the protocol processor in the appropriate resource partition. Packet classifiers in parallelized networking protocols (such as TCP/IP [BGP+94]) and multi-snoop memory buses in high-performance SMP servers (such as the Gigaplane-XB [CPWG97]) use hardware mechanisms analogous to PDQ to dispatch and handle packets/transactions in parallel.

PDQ gives rise to a range of implementations varying in cost and performance. High-performance PDQ designs dynamically balance the load among the protocol processors in hardware by dispatching protocol messages or requests to idle protocol processors. Less hardware-intensive designs statically select the protocol messages and requests that dispatch to a distinct protocol processor. Similarly, both hardware-centric and software-centric models for executing software protocols (Figure 1-2) can take advantage of PDQ mechanisms to parallelize the protocol execution.

1.3 Experimenting with Software Fine-Grain DSM

This thesis evaluates protocol execution semantics and scheduling policies for software protocols in the context of fine-grain DSM on an SMP cluster. DSM implements a (virtually) global image of a single address space over (physically) distributed SMP memories. Conventional software implementations of DSM allocate and maintain coherence at the page granularity (or larger). Transparent page-level coherence, however, often results in frequent movement of (large) data pages among SMP memories and poor performance in fine-grain applications. To mitigate this problem, most page-based software DSMs require programmers to carefully annotate applications with system-specific synchronization primitives thereby sacrificing transparency.

Fine-grain DSM allocates shared memory at the page granularity, but maintains coherence at cache block granularity (e.g., 32-128 bytes). Fine-grain DSM is a particularly attractive implementation of DSM on SMP clusters because it transparently—i.e., without the involvement of the application programmer—extends an SMP's fine-grain shared-memory abstraction across a cluster. Such a shared-memory system enables the portability of SMP server-based applications across a cluster of SMPs.

The following presents the simulation methodology for the systems evaluated in this thesis. The rest of the section presents the goals and the results from the two experimental studies in the thesis. Section 1.4 concludes the chapter by describing the thesis organization.

1.3.1 A Model for the System Architecture

The experimental methodology in this thesis is a simulation model for fine-grain DSM implemented on an SMP cluster. I use the Wisconsin Wind Tunnel II (WWT-II) [MRF+97]—a parallel simulator of SMP clusters—as the simulation test-bed. Figure 1-1 (on page 2) illustrates the general organization of an SMP cluster multiprocessor. Each machine node consists of one or more 400-MHz dual-issue statically-scheduled processors (modeled after the Ross hyperSPARC), each with a 256-entry direct-mapped TLB and a 1-Mbyte one-level data cache. WWT-II assumes instruction cache references are all single-cycle hits. Such an assumption does not invalidate the results and conclusions of this thesis, because the software protocols studied have very short running times.

Processor caches are kept coherent within each SMP using a 100-MHz split-transaction 256-bit wide memory bus. WWT-II also assumes a constant point-to-point network latency of 100 processor cycles, but accurately models contention at the network interfaces.

WWT-II assumes an operating system both provides local services and manages the nodes collectively as a single parallel machine [ACP95,HT93]. Parallel applications follow the SPMD programming model. This thesis assumes space sharing—where the nodes are logically allocated to separate parallel tasks. More general time sharing is of course possible, but is beyond the scope of this work.

1.3.2 Scheduling Policies for Single-Threaded Protocols

Cost-effective (rather than high-performance) multiprocessor designs may interconnect low-cost small-scale SMPs with commodity networking hardware. Many such systems provide little support to accelerate protocol execution and run a single software protocol thread on every SMP. To reduce protocol invocation and execution overhead, these systems sometimes dedicate one of the SMP processors to always run the protocol thread. A dedicated protocol processor, however, may waste processor cycles (e.g., when the application is compute-bound) which could have contributed to computation.

The first experimental study asks the question "*when does it make sense to dedicate one processor in each SMP node specifically for protocol processing?"* The central issue is when do the overheads eliminated by a dedicated protocol processor offset its lost contribution to computation? The study addresses this question by examining the performance and cost-performance trade-offs of two scheduling policies for a single-threaded protocol:

- *Fixed*, a dedicated policy where one processor in an SMP is dedicated to execute the protocol thread, and
- *Floating,* a multiplexed policy where all processors compute and alternate acting as protocol processor.

Results from running shared-memory applications on a simulation model for fine-grain DSM using a software coherence protocol indicate that:

- Fixed benefits fine-grain protocols (e.g., fine-grain DSM) much more than coarse-grain protocols (e.g., page-based DSM).
- Fixed generally offers superior performance for systems with four or more processors per SMP.
- Floating's performance is not very sensitive to protocol invocation overhead.
- Fixed always results in the most cost-effective design for systems with high protocol invocation overheads (e.g., systems with no OS support for fast exception handling) running fine-grain protocols.

1.3.3 Executing a Fine-Grain DSM Protocol in Parallel

Computer designers can also build higher-performance multiprocessors from large-scale SMPs. Large-scale SMPs increase the demand on software protocol execution within every SMP and may significantly benefit from parallel protocol execution. This thesis proposes *PTempest* (Parallel Tempest), a parallel programming abstraction for building fine-grain DSM protocols based on Tempest [Rei95] and PDQ.

Tempest defines a set of mechanisms for implementing user-level fine-grain DSM protocols in software. Tempest's mechanisms allow a protocol to map/unmap shared-memory pages in an application's address space. Using Tempest, a protocol can also manipulate shared-memory access semantics to fine-grain (e.g., 32-128 bytes) memory blocks on the mapped pages. Protocol handlers in Tempest, however, are based on Active Messages [vECGS92] and have single-threaded execution semantics. PTempest relaxes the single-threaded execution semantics of Tempest using the PDQ parallel protocol execution mechanisms.

The second experimental study evaluates fine-grain DSM systems based on two implementations of PTempest—*Hurricane* and *Hurricane-1*—with varying degrees of hardware support. Both systems integrate PDQ with fine-grain shared-memory access control logic and networking hardware in a custom network interface device. Hurricane also tightly integrates one or more embedded protocol processors with the custom device representing a high-performance PTempest implementation. In contrast, Hurricane-1 is representative of a lower-cost PTempest implementation and uses the SMP processors to execute the software protocol.

To gauge the impact of parallel protocol execution on software fine-grain DSM's performance, the experiment compares the Hurricane systems against S-COMA, a hardware implementation of fine-grain DSM [HSL94]. Results from running shared-memory applications on simulation models for the Hurricane systems and S-COMA indicate that:

- PDQ helps significantly improve the performance of software protocol implementations, alleviating the software protocol execution bottleneck.
- A Hurricane system with four embedded processors either outperforms or performs as well as S-COMA.
- A cost-effective Hurricane-1 system with no extra dedicated protocol processors (i.e., using SMP processors for both computation and executing protocols) performs within 75% of S-COMA on average.

1.4 Thesis Organization

Chapter 2 and Chapter 3 describe the taxonomy for software protocol execution semantics and scheduling policies respectively. These chapters qualitatively discuss the performance and cost-performance trade-offs between the classes within each taxonomy. Chapter 2 also describes PDQ and discusses its design and implementation spectrum with respect to performance and cost-performance. Chapter 4 presents the first experimental study evaluating the scheduling policies for single-threaded protocols. Chapter 5 describes the second experimental study evaluating how PDQ can help improve software fine-grain DSM's performance by parallelizing the protocol execution. Finally, Chapter 6 concludes the thesis with a summary and future directions for this work.

Chapter 2

Protocol Execution Mechanisms & Semantics

This thesis investigates techniques for improving fine-grain software protocol performance in SMP clusters. This chapter studies alternative execution semantics for fine-grain software protocols. Distributed-memory parallel computers traditionally used uniprocessor nodes and executed the software protocols on either a node's single commodity processor along with computation [SGT96,KDCZ93,HT93,CBZ91,Int90,AS88], or an embedded processor on the network interface [K+94,RLW94,Mei93]. As a result, fine-grain software protocols in these machines had sequential execution semantics.

To take advantage of the superior cost-performance of SMPs, computer designers are also constructing parallel computers using SMPs as building blocks [LC96,WGH+97,CA96]. SMP nodes, however, increase the demand on software protocol execution because multiple SMP processors simultaneously generate protocol requests—e.g., to fetch remote memory blocks. Grouping processors into SMPs also reduces the number of nodes, and as a consequence, the number of network interface cards in the system. Decreasing the number of network interface cards also increases the protocol traffic into a node. The combined effect of a higher protocol request rate and a faster incoming protocol traffic quickly makes software protocol execution a communication bottleneck as SMP nodes become larger [MNLS97,LC96].

One approach to mitigate the software protocol bottleneck is to parallelize the protocol execution. Software protocols may run on either multiple embedded processors on the network interface card, or several SMP-node commodity processors. Parallel protocol execution using SMP processors is particularly attractive for clusters of large-scale SMPs. Large-scale SMPs increase the likelihood of many (computation) processors being idle (e.g., waiting for synchronization) enabling them to contribute to software protocol execution. Such a design both improves performance by increasing the parallelism in protocol execution and reduces cost by obviating the need for extra dedicated protocol processors [FW97c,FW96].

In this chapter, I present a taxonomy of software protocol execution semantics:

- *Single-threaded* protocol execution allows for only a single protocol thread to run on an SMP node at any given time.
- *Multi-threaded* protocol execution allows multiple protocol threads to simultaneously execute on an SMP node.

I propose a novel set of mechanisms called *parallel dispatch queue* (PDQ) for implementing synchronization-free parallel protocols. Because of the short running time of protocol handlers in fine-grain protocols, parallelizing protocol execution using conventional locking schemes (e.g., software spin-locks) would result in prohibitively high synchronization overheads. PDQ helps obviate the need for explicit synchronization mechanisms by partitioning the protocol resources and requiring protocols to label protocol events with the appropriate partition id. PDQ efficiently parallelizes protocol execution by dispatching and executing protocol handlers in parallel for protocol events with distinct partition ids.

The following (Section 2.1) first describes the mechanisms and resources typically required to execute fine-grain software protocols. Section 2.2 describes the taxonomy and

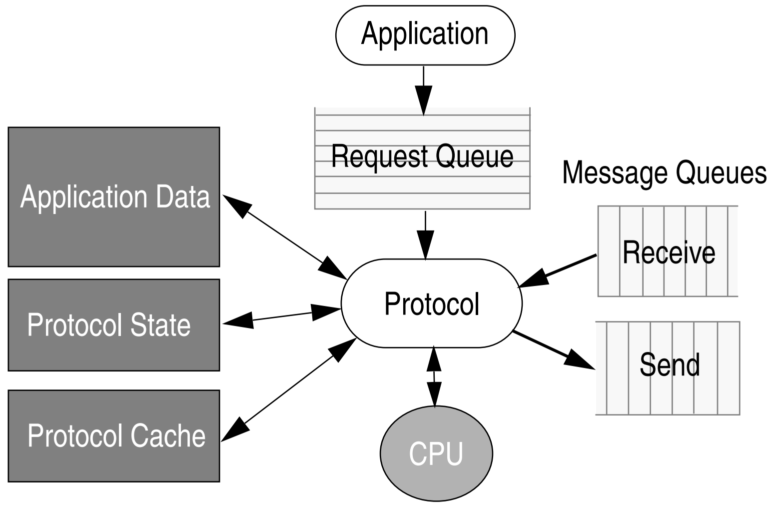


Figure 2-1. Resources for a simple fine-grain software protocol.

compares and contrasts the advantages of each protocol execution semantics. Section 2.3 presents the PDQ mechanisms for parallel execution of synchronization-free protocols, and qualitatively evaluates the design and implementation space for PDQ. Section 2.4 and Section 2.5 discuss the related work and summarize the chapter's contributions respectively.

2.1 Protocol Execution Mechanisms & Resources

Many high-level parallel programming abstractions are implemented using software communication protocols [JKW95,RLW94,CDG+93]. These protocols typically implement simple finite-state machines which read as input a *protocol event* and the corresponding state, make a state transition, and write as output a subsequent protocol event and the new state. A protocol event is typically either a network message or an application's request for protocol invocation on a remote node. Protocol state often corresponds to a data entity in an application, and protocol events implement communication by moving data between the network and memory.

Protocols require a set of mechanisms and resources to generate events and implement the corresponding state transitions. Figure 2-1 illustrates an example of protocol resources required by a simple fine-grain software communication protocol (like Split-C [CDG+93]). Protocol messages communicate an application's data among the machine nodes. Simple software protocols may cache remote data temporarily into a protocol cache to eliminate multiple network traversals to access remote memory. A set of protocol states maintains the access semantics to data in main memory and the protocol cache. Applications submit requests—e.g., to fetch remote data—to the protocol through a request queue. A protocol also moves data among machine nodes using a pair of send/receive message queues.

Cost and performance are the critical factors in choosing whether protocol resources should be implemented in hardware or software. At one extreme, low-cost implementations maintain the protocol cache, protocol state, and/or the request queue in main memory using software [JKW95,SFL+94,CDG+93]. These systems also use send/receive message queues on commodity network cards, and execute the software protocol on the node's commodity processor(s). At the other extreme, high-performance designs (as in tightly-coupled custom hardware support for fine-grain DSM) use SRAM to implement the protocol cache and state, and tightly integrate them with the request and message queues and one or more embedded processors on a custom board [K+94,RLW94]. Less-integrated designs with hardware support for only performance-critical resources are also feasible based on the desired cost and performance trade-offs [RPW96,BLA+94].

2.2 Protocol Execution Semantics

There are two types of protocol execution semantics: single-threaded and multi-threaded. This section describes the two execution semantics and discusses their advantages/disadvantages.

2.2.1 Single-Threaded Protocol Execution

Single-threaded protocol execution is the conventional way of executing software protocols; the protocol simply runs in a single thread on every node. Software protocols implementing parallel programming abstractions on uniprocessor-node machines—such as fine-grain [SFL+94] and page-based software DSM [CBZ91,KDCZ93], messaging abstractions based on Active Messages [WHJ+95,CDG+93], and object-based distributed systems [BTK90,BJK+95]—all use single-threaded protocol execution semantics.

In systems with single-threaded execution semantics, protocol resources (such as the protocol cache, protocol state, and the messaging queues) are only accessible to a single protocol thread. As such, there is no need for synchronizing and coordinating accesses to the resources. Moreover, a single thread precludes contention and queueing of accesses at the resources. Eliminating synchronization and queueing reduces *protocol occupancy*—i.e., the time to handle a single protocol event [HHS+95]—decreasing overall communication time.

Single-threaded protocol execution is advantageous for applications that primarily benefit from low protocol latency. In latency-bound applications, communication is generally asynchronous, sporadic, and mainly relies on quick protocol round-trip time. Due to the lack of protocol event queueing at the protocol processor, these applications can not take advantage of parallel (i.e., multi-threaded) protocol event handling. Such applications, however, may benefit much from a low protocol occupancy characteristic of a single-threaded protocol execution.

Single-threaded protocol execution also favors clusters of small-scale (“narrow”) rather than large-scale (“fat”) SMPs. SMPs with only a small number of processors may not generate enough demand for software protocol execution to justify parallelizing it. Grouping processors into small-scale SMPs also increases the number of nodes in the system. Because every node executes a single protocol thread, a larger number of nodes reduces the demand on protocol execution on a single protocol thread. On the contrary, large-scale SMPs place a large demand on protocol execution and may make a single protocol thread the communication bottleneck.

Hardware support can help accelerate single-threaded protocol execution, alleviating the protocol execution bottleneck. Pipelined execution of protocol events (e.g., performing protocol state update and memory access in separate pipeline stages) can help increase single-threaded protocol execution bandwidth. Tight integration of protocol resources with embedded protocol processors on a custom device [K+94,RLW94] lowers protocol occupancy. Latency-hiding techniques such as speculative execution of protocol handlers [Muk98] and speculative accesses to protocol resources [HKO+94] also lower the protocol occupancy. A lower protocol occupancy improves protocol execution bandwidth without requiring multi-threaded protocol execution.

2.2.2 Multi-Threaded Protocol Execution

Multi-threaded protocol execution allows simultaneous invocation of several protocol threads in parallel. Executing a protocol in parallel increases communication bandwidth by reducing queueing delays at the protocol processor. Higher communication bandwidth benefits bandwidth-bound applications—i.e., applications experiencing large queueing delays at the protocol processor. Parallel protocol execution also favors clusters of large-scale (rather than small-scale) SMPs because large SMPs place high demands on protocol execution.

Multi-threaded protocol execution also relaxes restrictions on protocol complexity. Many systems require protocols to quickly drain the network to avoid contention. Protocols are forced to have short handlers that simply integrate data from the network into an application's data structures and optionally send a reply message. Allowing multiple handlers to execute simultaneously somewhat relaxes the restrictions on handler running times. A higher handler running time allows for more aggressive protocols that optimize communication by making complex state transitions to reduce message frequencies [GJ91], vectorizing messages [FLR+94], performing simple computations [BALL90], and creating/managing light-weight threads [WHJ+95]. Simultaneously executing handlers also helps reduce network contention and backward pressure by increasing the message reception rate.

Many systems also require protocols to have non-blocking handlers to avoid network deadlocks. Non-blocking handlers, however, restrict arbitrary synchronization between the computation and the protocol. Some systems simply avoid synchronization and implement intricate handler invocation schemes to make handlers non-blocking [BCL+95]. Others allow synchronizing handlers but significantly increase handler execution time in the case where synchronization actually occurs [WHJ+95]. These systems use complex mechanisms to detect synchronization, and subsequently buffer the protocol event and create a separate computation thread to execute the protocol handler.

With the help of multi-threaded protocol execution, the system can avoid network deadlocks while allowing for blocking protocol handlers. The system can specialize one or more protocol processors to exclusively run non-blocking handlers. By requiring protocols to label protocol events with the handler execution type (i.e., blocking or non-blocking), the system can distinguish and execute protocol handlers on the appropriate protocol processors. The specialized protocol processors will always be available to drain the network thereby avoiding deadlocks.

Executing protocols in parallel also comes at a cost. Parallel execution requires accesses to (shared) protocol resources to be mutually exclusive. Parallelized legacy network protocols (such as TCP/IP) often implement mutual exclusion and synchronization using spin-locks [SKT96,BG93,Kai93,HP91]. Legacy protocols, however, typically have long running times and a coarse granularity of data structure accesses. As such, the overhead of acquiring/releasing spin-locks does not impact the overall performance in these protocols.

In fine-grain communication protocols, handlers have very short running times and typically move a fine-grain (e.g., 32-128 byte) data block between memory and message queues, update the corresponding protocol state, and/or remove an entry from the request queue. Because of the handlers' short running time, the overhead of acquiring/releasing locks around message queues and protocol data structures could prohibitively increase handler occupancy [BG93]. Furthermore, moving data between memory and message queues often dominates handler running time and synchronizing around message queues would effectively serialize handler execution. As such, parallel execution of fine-grain protocols requires efficient support for handler synchronization.

Parallel handler execution may also result in contention for protocol resources. Multiple handlers may simultaneously contend for common protocol resources. Parallel execution may also increase the access frequency to protocol resources—e.g., due to protocol data migration among multiple protocol processor caches [SKT96,TG89]—further exacerbating the resource contention. In the absence of efficient support for high-bandwidth access paths to resources, resource contention may become a bottleneck rendering parallel execution less beneficial.

High-bandwidth memory systems, however, are only characteristic of high-cost medium- to large-scale SMP servers. These systems are typically equipped with interleaved memory banks and out-of-order wide memory buses allowing multiple protocol handlers to simultaneously access memory. Because accessing memory often dominates handler execution time in fine-grain protocols, low-performance memory systems may effectively serialize the execution of parallel protocol handlers.

Providing high-bandwidth access paths to protocol resources in tightly-integrated custom devices also may significantly increase cost. Rather than provide high-bandwidth access paths, some systems opt to specialize resource accesses to specific embedded processors on the custom device [MNLS97,CAA+95]. Specializing resource accesses, however, may result in load imbalance in protocol execution and offset the advantages of parallel protocol execution.

Resource synchronization and contention in multi-threaded protocol execution may also significantly increase protocol occupancy resulting in poor single-thread performance. Single-thread performance plays a major role for latency-bound applications which primarily rely on quick protocol round-trip times and exhibit little or no queueing at the protocol processor. Lower single-thread performance may also increase critical-path execution time and decrease overall performance in applications with large load imbalance.

Parallel handler execution may also lead to out-of-order processing of protocol events. Users often specify protocols in terms of finite-state machines. The larger the state space, the more complex the process of debugging, verifying and maintaining of the protocol code. Out-of-order message delivery and request processing forces protocol writers to introduce intermediate machine states, resulting in a larger and more complex protocol [Cha97,GJ91].

2.3 Synchronization-Free Parallel Protocols

Access synchronization is only necessary if the handlers indeed simultaneously access common resources. Careful partitioning of resources among the protocol threads would guarantee mutual exclusion in accessing resources and obviates the need for synchronization. The protocol programming abstraction can provide mechanisms for labelling protocol events. The system can then dispatch a protocol event based on its partition label to the corresponding protocol thread with exclusive access to its own set of protocol resources. This approach allows parallel execution of protocol handlers for events with distinct partition labels.

Protocol resources can be partitioned based on application, programming abstraction, or system characteristics. In many parallel programming abstractions such as fine-grain DSM [K+94,RLW94], high-level languages [HKT92,BCF+93,CDG+93] or run-time systems [JKW95] implementing a global address space, and object-based distributed systems [BK93,BTK90,BST89], data entities exist in the form of immutable memory objects. A memory coherence block in fine-grain DSM is an example of an immutable object. Partitioning the protocol resources at an object granularity in these systems allows protocols to access resources without requiring explicit synchronization mechanisms.

Protocol resources for an object consist of several components (e.g., the object data in memory or protocol cache, or the object protocol state). The protocol writer and the system designer can make resource accesses corresponding to distinct objects mutually exclusive by careful implementation and storage of the object components. For instance, if a protocol maintains the object state in the form of bit vectors—as in the directory state in a typical DSM—then the machine must provide mechanisms for updating one object's state bits without accessing other bits in the vector. Instead, if protocol state is maintained in main memory, the protocol may have to store the state in a form that is in accord with the machine's native memory access granularity—e.g., a byte or word—to guarantee mutual exclusion.

To identify which object partition a protocol event—i.e., a protocol message or a request—is associated with, a protocol is required to label protocol events with partition ids. Upon generating an event, a protocol may simply provide an id along with the event [BCL+95]. To dispatch a protocol event, the system must first *demultiplex* the event into a partition based on the partition id. The system must also *bind* an object partition to a protocol thread. Protocol event demultiplexing and binding mechanisms are very similar in nature to those in parallelized network stack protocols where message packets are demultiplexed into protocol stacks [BGP+94,DPD94] and protocol stacks are assigned to processors [SKT96].

In some parallel programming environments, handler execution may occasionally involve accessing multiple object partitions. In fine-grain DSM, for example, the majority of protocol handlers manipulate protocol resources associated with a single memory coherence block. Occasionally, however, a protocol handler may map/unmap a page of data including multiple memory blocks (e.g., to migrate a page between two nodes). To maintain consistency of protocol data structures and application data, the handler execution must appear atomic. Because such handlers execute infrequently, the system can simply execute the handlers sequentially providing them access to the entire set of protocol resources. To indicate a sequential handler execution semantics, the protocol can label protocol events with special (reserved) partition ids. Upon receiving a protocol event with the special partition id, the dispatching mechanisms must first allow all protocol handlers to complete executing, dispatch the protocol event with sequential execution semantics, and wait for the sequential handler to complete executing before dispatching other protocol events.

Similarly, a software fine-grain protocol may have handlers that do not require access synchronization. Accessing read-only remote data structures, for instance, in Split-C [CDG+93] does not require handler synchronization because neither the protocol nor the application modify the (read-only) data. By labeling protocol events with special partition ids, multiple instances of such protocol handlers can simultaneously execute in parallel without any dispatch and execution restrictions.

This section proposes a novel set of protocol dispatching mechanisms called *parallel dispatch queue* (PDQ) for parallel execution of synchronization-free protocols. It qualitatively evaluates the PDQ design spectrum in terms of both cost and performance. Chapter 5 evaluates a high-performance implementation of PDQ in the context of fine-grain distributed shared memory.

2.3.1 Parallel Dispatch Queue (PDQ)

Parallel dispatch queue is a set of mechanisms that allows programmers to write synchronization-free parallel protocols. PDQ requires a protocol to group its data structures into object partitions and label protocol events with the corresponding partition id. By simultaneously dispatching protocol handlers only from distinct object partitions, PDQ allows parallel execution of protocols without requiring synchronization of accesses to protocol resources. PDQ also provides special partition ids for sequential handler execu-

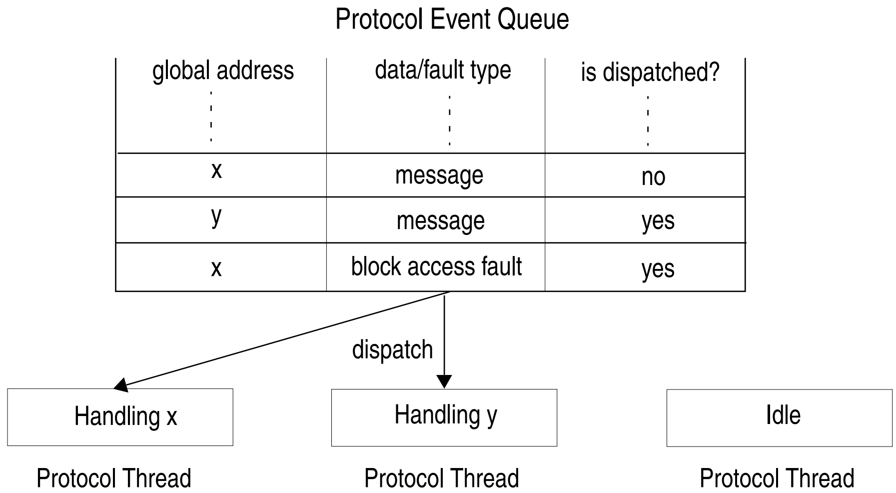


Figure 2-2. Parallel protocol event dispatch in fine-grain DSM.

tion semantics and for handlers with restriction-free dispatch semantics (as discussed in the previous section).

Figure 2-2 illustrates an example of how a PDQ implementation may dispatch protocol events in parallel in a fine-grain DSM. There are two types of protocol events. A *message* typically carries either a memory block's data or coherence information such as an invalidation. A *block access fault* corresponds to a request for a remote memory block (generated locally). A memory block constitutes an object partition and therefore both protocol event types are labeled with a memory block's global address. Protocol data structures are grouped so that handler accesses to distinct memory blocks are mutually exclusive. Protocol events corresponding to (distinct) global addresses *x* and *y* may dispatch in parallel. Multiple protocol events for a global address *x* must be handled serially even though there is an idle protocol thread ready to dispatch.

To dispatch an event to a protocol thread, a PDQ implementation must first demultiplex the event into a dispatch queue representing an object partition. A dispatch queue must also bind to a protocol thread before the thread can remove and handle an event off of the queue. Demultiplexing events and binding dispatch queues can be performed either statically or dynamically based on the desired system cost and performance.

The simplest form of a PDQ implementation statically demultiplexes (i.e., inserts) an event into protocol dispatch queues solely based on its object partition id. To allow a protocol thread to remove and handle protocol events from a dispatch queue, such a system also binds a protocol thread (statically) to a dispatch queue for the duration of an application's execution. Because protocol events in such a system are always handled by the same protocol thread, a skewed distribution of protocol events can lead to a load imbalance among the protocol threads.

To mitigate the load imbalance, a PDQ implementation can (statically) demultiplex protocol events into a large number of dispatch queues and provide mechanisms for a protocol thread to (dynamically) bind to a non-empty dispatch queue. Although such a system helps distribute the event execution load among the protocol threads, dynamic binding to dispatch queues may incur high overheads offsetting the advantages of protocol load distribution.

The highest-performance PDQ implementation statically binds a (single-entry) dispatch queue to each protocol thread and dynamically demultiplexes a protocol event into a dispatch queue upon demand. A dynamic demultiplexing PDQ only inserts an event into an empty dispatch queue if there are no events with the same partition id in other dispatch queues. Dynamic demultiplexing offers superior performance by allowing events to demultiplex into any dispatch queue thereby eliminating load imbalance. Dynamic demultiplexing, however, increases hardware complexity and cost by requiring an associative search of dispatch queues upon demultiplexing. Because event distribution among dispatch queues only occurs upon demand, dynamic demultiplexing obviates the need for protocol load distribution through dynamic binding.

The following section describes static and dynamic demultiplexing in detail and qualitatively evaluates the cost-performance trade-offs between the two schemes.

2.3.2 Statically Demultiplexing Protocol Events

Static demultiplexing divides the object partitions among a set of protocol dispatch queues based on their ids. For example, with two dispatch queues, even objects go to one queue and odd objects go to the other queue. Such a scheme is advantageous because protocol events can be demultiplexed quickly and early entirely based on partition ids. Because an object with a given id is always demultiplexed to the same dispatch queue, dispatch queues need not be searched for already-dispatched instances of an event associated with the object. By not requiring an associative search, static demultiplexing also reduces the hardware complexity and cost of a PDQ implementation.

Early demultiplexing allows dispatch queues to be implemented as simple hardware fifos on the network interface card. Dispatch queues can take advantage of hardware caching techniques to increase event (e.g., message) buffering and overflow their contents to memory (as in cachable queues [MFHW96]). A large influx of protocol events can quickly demultiplex into the appropriate queues and spill to memory removing backward pressure from the network.

The simplest form of a PDQ implementation statically binds a single dispatch queue to a protocol thread. Such a design is analogous to conventional hardware message queues in parallel machines [Int93,HT93] where a protocol thread dispatches events off of a single message queue. Figure 2-3 (left) illustrates protocol event dispatch for statically demultiplexed events and statically bound dispatch queues. An application or the network generate protocol events labeled with object (partition) ids. There is a dispatch queue corresponding to every partition id. There are also as many dispatch queues as protocol threads; each protocol thread polls on its designated dispatch queue.

Static binding may be advantageous because the protocol code and data structures can exploit locality of references in the protocol processor's cache hierarchy. Moreover, from a design standpoint, it may improve both cost and performance to make certain hardware protocol resources (e.g., such as the directory in fine-grain DSM) available only to a single

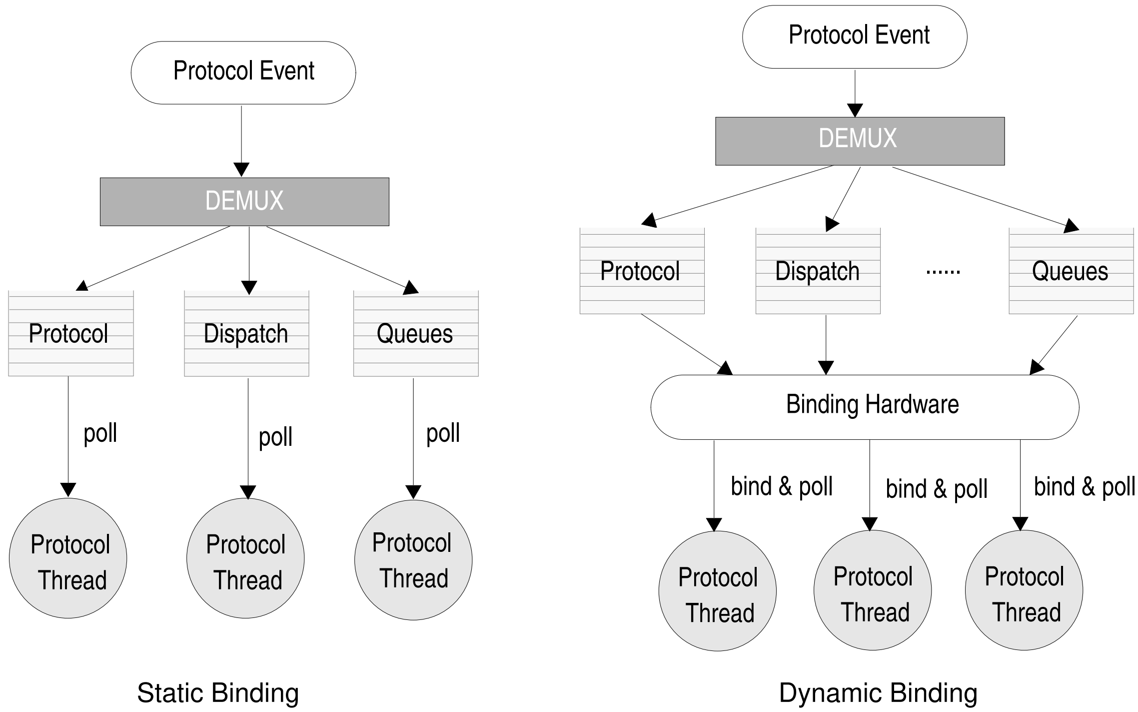


Figure 2-3. Static protocol event demultiplexing: protocol events can be both statically demultiplexed and bound (left) or statically demultiplexed and dynamically bound (right) to protocol threads.

processor [MNLS97,CAA+95]. Such a processor would be responsible for handling protocol events that require access to the specific resource.

Static partitioning of objects into a small number of dispatch queues, however, may result in load imbalance. Alternatively, the protocol events may be demultiplexed into a larger number of dispatch queues, where each queue is responsible for a smaller object partition. Reducing the size of object partitions allows for a better event load distribution among the dispatch queues. Because there are more dispatch queues than protocol threads, dispatch queues must dynamically bind to protocol threads. Dynamic binding, therefore, alleviates the load imbalance among the protocol threads by allowing them to choose among a large group of non-empty dispatch queues.

Dynamic binding may also be advantageous in systems which preempt the protocol thread execution. There are protocol scheduling policies which alternate scheduling the execution of an application and the protocol on the same processor (Chapter 3). As a result, these scheduling policies may preempt a protocol thread's execution. Dispatch queues bound to preempted protocol threads are not accessible and can not be serviced by other protocol threads potentially resulting in a load imbalance. To mitigate load imbalance in the presence of a preempting protocol scheduling policy, the system must allow a protocol thread to dynamically bind to a dispatch queue when the thread is scheduled to execute.

Figure 2-3 (right) illustrates protocol event dispatch for statically demultiplexed events and dynamically bound dispatch queues. When idle, a protocol thread requests to bind to a non-empty dispatch queue. Upon binding, the protocol thread polls and dispatches events from the queue until the queue is empty. The protocol thread subsequently relinquishes the queue and requests to bind to another non-empty queue.

Dynamic binding is common in systems with multiple protocol event queues. An example of such a system is the Typhoon-0 fine-grain DSM, that uses separate block access fault and message receive queues (as in Typhoon-0 [Rei96]). Other examples are messaging systems which use backup message buffer space in memory due to limited message buffering on the network interface card [Sch97,MFHW96].

Dynamic binding also allows for the use of cachable queues. Cachable queues typically maintain information as to whether a queue is full or empty in the network interface hardware even though the queue itself may mostly reside in memory [MFHW96]. A PDQ implementation can simply keep track of which cachable queues are non-empty and bind queues to protocol threads accordingly.

Dynamic binding also has its disadvantages. Dynamic binding requires mechanisms for checking which dispatch queues are non-empty and maintaining a record of whether a dispatch queue has been bound to a protocol thread. Binding also incurs overhead which

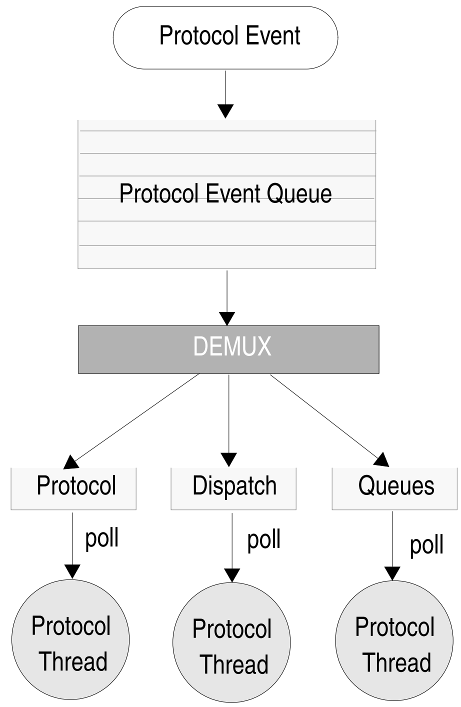


Figure 2-4. Dynamic protocol event demultiplexing.

increases the protocol event handling time and reduces performance in the absence of queueing.

2.3.3 Dynamically Demultiplexing Protocol Events

Static demultiplexing reduces a PDQ implementation's hardware complexity but may result in a load imbalance and lower performance. Dynamic binding helps mitigate load imbalance but may incur high dispatch overhead and decrease a protocol's single-thread performance if dispatching events frequently require binding. Alternatively, a PDQ implementation can entirely eliminate load imbalance by demultiplexing protocol events dynamically upon demand.

Figure 2-4 depicts protocol event dispatch for dynamically demultiplexed events. Every protocol thread is assigned a single-entry protocol dispatch queue. A protocol dispatch queue temporarily holds the event being handled by a protocol thread. A protocol event is inserted into a protocol event queue upon arrival. When a protocol thread polls on an empty protocol dispatch queue, a protocol event is demultiplexed from the protocol event queue and is subsequently placed in the protocol dispatch queue. Demultiplexing a protocol event consists of selecting an event from the protocol event queue and searching in the protocol dispatch queues to guarantee that no other protocol events with the same partition id are dispatched. A protocol thread clears the entry in its dispatch queue when it completes handling a protocol event.

Dynamic demultiplexing increases a PDQ implementation's hardware complexity by requiring an associative search through the protocol event queue entries and the protocol dispatch queues. Furthermore, associative search may be slow when the search space is large. To accelerate the search, the search space can be limited to a small number of entries at the head of the protocol event queue. Limiting the search space, however, may limit the parallelism in event dispatch and lower performance. Alternatively, to alleviate the search speed limitation the search can be initiated early (e.g., as soon as an event is dispatched).

Late demultiplexing also complicates the use of cachable queues. Protocol event queue entries may spill to memory requiring event search to access memory. Applications benefitting from parallel protocol execution typically exhibit bursty communication phases. Memory spilling of queue entries will be frequent in these bandwidth-bound applications, increasing the likelihood that the event search will involve accessing memory. Searching in memory to dispatch available protocol events would be prohibitively slow and may offset the gains from parallel protocol execution.

Customizing the PDQ design, however, may allow for the use of cachable queues. Buffering several entries at the head of the protocol event queue in hardware (on the network interface card) allows the search engine to proceed without frequently accessing the memory. Such a scheme allows the entire protocol event queue to spill to memory much like a cachable queue. Buffer entries can be prefetched from memory upon an event dispatch to hide the latency of the memory access.

Protocol event search also only requires the use of object partition ids. To reduce hardware storage requirement or to increase the search space, a PDQ implementation could maintain just the partition ids of several entries at the head of protocol event queue. Upon event dispatch, the system must fetch the event from memory and place it in the appropriate protocol dispatch queue. Alternatively, the system can place the queue entry index in the protocol dispatch queue and let the protocol thread fetch the queue entry from memory.

To further reduce the hardware storage requirements for partition ids, a PDQ implementation using a cachable event queue can instead store a small number of bits from each partition id. Such an optimization, however, may decrease the parallelism in event dispatch because the system may falsely identify independent object partitions to be identical, thereby dispatching them serially.

2.4 Related Work

There is a myriad of literature on parallel implementations of stack protocols—such as TCP/IP—in the networking community [SKT96,BG93,Kai93,HP91]. These protocols typically use spin-locks to guarantee mutually exclusive accesses to protocol resources from parallel protocol threads. Because stack protocols have long handler running times, the overhead of acquiring/releasing spin-locks can be amortized over the long execution of a handler. In contrast, fine-grain communication protocols—such as fine-grain DSM—have protocol handlers with short running times and require more efficient synchronization mechanisms to provide mutual exclusion for accessing protocol resources in parallel.

Many researchers have studied early packet demultiplexing in stack protocols using classifiers either directly in hardware [BGP+94] or in software running on an embedded network processor [DPD94]. Packet classifiers (statically) demultiplex network packets into the corresponding protocol stacks. This work builds upon previous research on event demultiplexing by proposing *dynamic* demultiplexing mechanisms for dispatching events (such as network messages) to protocol threads.

Several studies on clusters of SMPs implementing fine-grain DSM conclude that the high demand on software protocol execution due to multiple SMP-node processors can make single-threaded software protocol execution the bottleneck [MNLS97,LC96]. One such study evaluates a limited form of parallel protocol execution which statically demultiplexes protocol events into two parallel protocol threads [MNLS97]. The study reports a high load imbalance between protocol threads and a utilization gap of up to 65%.

2.5 Summary

This chapter proposes a taxonomy for software protocol execution on a node of a parallel system:

- Single-threaded protocol execution allows for a single protocol thread to execute at any given time.
- Multi-threaded protocol execution allows multiple protocol threads to execute simultaneously.

The chapter presents application and system characteristics that impact the cost-performance trade-offs between the two classes of protocol execution semantics. Both single-threaded and multi-threaded protocol execution have been previously studied in the context of coarse-grain network stack protocols. Fine-grain software protocols, however, have traditionally executed in a single thread on every node of a distributed-memory parallel machine. Because of the short handler running times, fine-grain protocols can not employ conventional high-overhead locking techniques (such as spin-locks) to synchronize and coordinate parallel protocol threads.

The central contribution of this chapter is a novel set of mechanisms, PDQ, for efficient multi-threaded execution of fine-grain software protocols. PDQ is based on the key observation that careful partitioning of protocol resources—such as protocol cache and state—among protocol threads allows multiple handlers to simultaneously access the resources synchronization-free. The chapter presents a qualitative evaluation of the design and implementation spectrum for PDQ.

Chapter 3

Protocol Scheduling Policies

Chapter 2 discussed how protocol execution semantics can impact the communication bandwidth and thereby the system performance in SMP clusters. This chapter examines the impact of protocol invocation overhead in systems that execute software protocols on the SMP processors. Because both the application and the software protocol execute on the SMP processors, scheduling software protocol execution may also impact the system performance.

Early distributed-memory parallel machines contained uniprocessor nodes [HT93,Int90,AS88] and used a simple scheduling policy of alternating execution of computation and the software protocol. An SMP node enables the opportunity to dedicate one or more processors to only execute protocol threads. Therefore, SMP nodes give rise to two basic classes of protocol scheduling policies: *dedicated* and *multiplexed*.

A dedicated policy (statically) schedules one or more SMP-node processors to only execute protocol threads. A multiplexed policy allows all processors to perform computation and (dynamically) schedules one or more protocol threads to execute when processors become idle (e.g., due to waiting for a remote request or synchronization operation) or upon arrival of a protocol message.

This chapter describes and qualitatively evaluates the two classes of scheduling policies. Section 3.1 presents a description of the two classes. Section 3.2 enumerates mechanisms required to implement the policies. Section 3.3 discusses the choice of policy based on design and functionality requirements of the target system. Section 3.4 identifies application and system characteristics that have a significant impact on the performance of policies. Section 3.5 presents the policy trade-offs from a cost-performance perspective. Section 3.6 presents a summary of related work, and finally Section 3.7 summarizes the chapter.

3.1 Dedicated vs. Multiplexed Protocol Scheduling

A dedicated policy allocates one or more SMP processors to only execute protocol threads. By always polling the protocol event queues when otherwise idle, dedicated protocol processors eliminate the need for message interrupts or polling by compute processors. In addition to decreasing the total overhead, a dedicated policy invokes protocol handlers more quickly, reducing the protocol occupancy and round-trip latency.

The disadvantage of a dedicated policy is that protocol processors may waste cycles that could have productively contributed to computation. Multiplexed policies address this dilemma by using all processors to perform computation; however, when a processor becomes idle it becomes a protocol processor. Since all processors may be computing, either interrupts or (instrumented) periodic polling is still required to ensure timely protocol event handling. On the other hand, once a processor becomes a protocol processor, handler dispatch may be as efficient as in a dedicated policy.

Figure 3-1 illustrates scheduling a simple request/reply protocol with single-threaded execution semantics (i.e., one protocol thread executing on every node) on a machine with two dual-processor nodes. The figure illustrates protocol scheduling under a dedicated

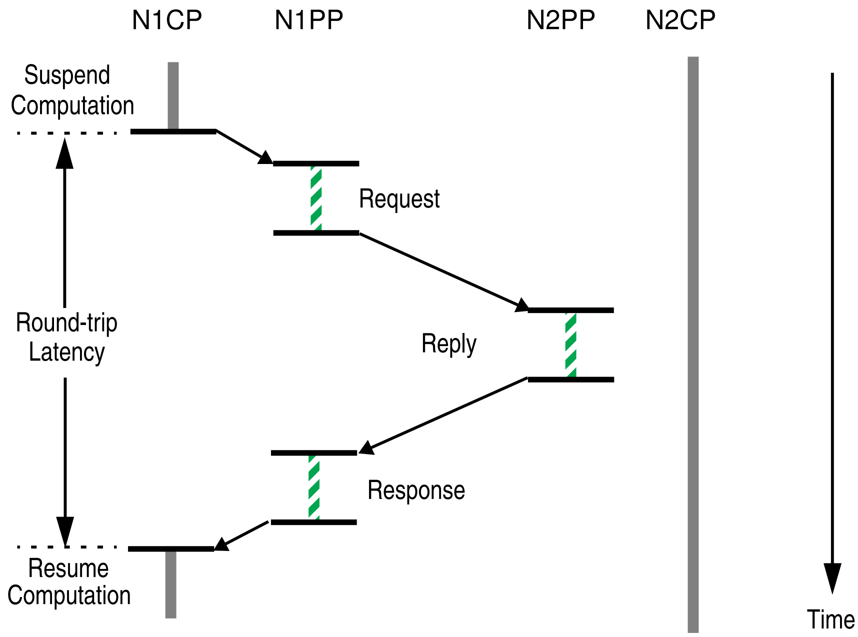


Figure 3-1. Protocol scheduling under a dedicated policy.

policy. The compute processor N1CP submits a request to the protocol processor N1PP, which in turn sends a message. At the destination node, protocol processor N2PP immediately invokes the protocol handler and sends the appropriate reply. Because of the dedicated protocol processor, compute processor N2CP proceeds uninterrupted. Finally, the reply arrives and the handler runs on N1PP, which then resumes the computation thread.

Figure 3-2 illustrates scheduling the same request/reply protocol, but under a multiplexed policy. The (compute) processor N1CP2 submits a request, becomes the protocol processor and sends a message. When the message arrives at node 2, all processors are busy computing. Thus, an interrupt is generated causing processor N2CP1 to act as protocol processor. The requesting processor incurs the overhead of two context switches (to and from the protocol thread) and the resulting cache pollution. The replying processor additionally incurs the overhead of delivering (and returning from) the interrupt. An idle processor acting as protocol processor (N1CP2) can immediately handle a request by another processor on the node (N1CP1), thereby eliminating the interrupt overhead.

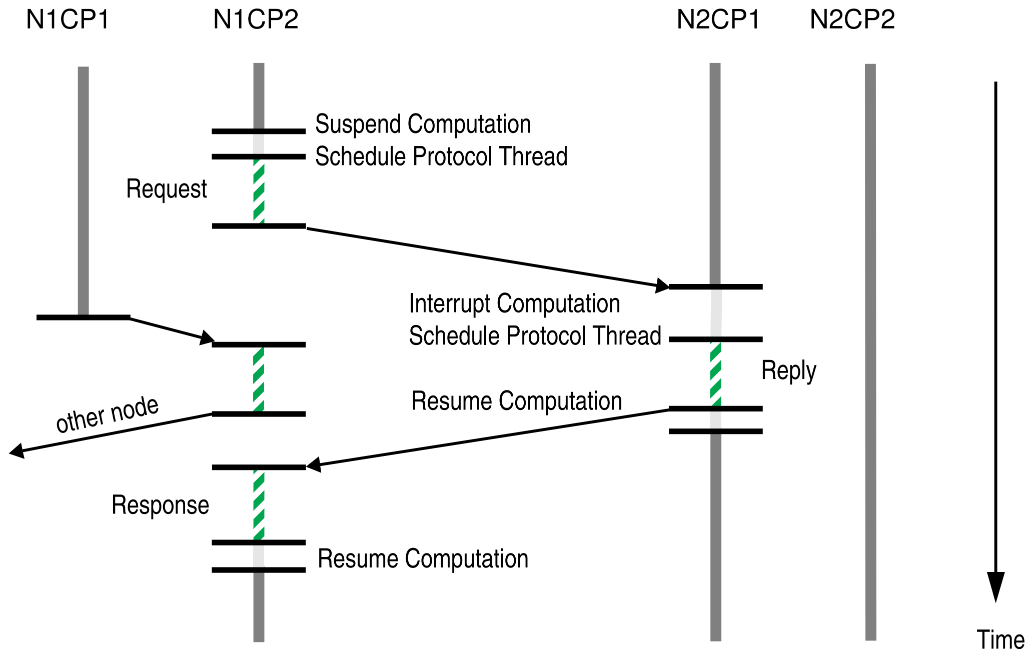


Figure 3-2. Protocol scheduling under a multiplexed policy.

Multi-threaded execution semantics gives rise to a variety of multiplexed scheduling policies. While most multiplexed policies schedule a protocol thread when a processor becomes idle, the policies may differ as to whether to interrupt the computation on one processor when there are protocol threads running on others. At one extreme, to eliminate excessive scheduling overhead, a policy may only interrupt the computation on one processor when there are no protocol threads running on others. At the other extreme, to minimize protocol invocation latency, a policy may interrupt the computation on all the processors to schedule protocol threads as long as there are outstanding protocol events.

3.2 Protocol Scheduling Mechanisms

Protocol invocation and scheduling requires system mechanisms to detect the arrival of a protocol event—e.g., a message—invoke the protocol, and to suspend/resume the computation (Figure 3-1 and Figure 3-2). To invoke the protocol, the system must provide mechanisms to schedule a protocol thread's execution. This section describes in detail the protocol invocation and scheduling mechanisms required by the two classes of scheduling policies.

3.2.1 Detecting Protocol Events

To detect a protocol event's arrival, processors must either poll the appropriate queues—e.g., a protocol request queue or a message queue (Figure 2-1 on page 15)—or the system must deliver an interrupt to a processor. Previous work has extensively studied the trade-offs between polling and interrupts [BCL+95,vECGS92]. This section briefly describes the trade-offs and their implications on the mechanisms required to efficiently implement either technique.

Polling is the natural approach to detecting protocol events under a dedicated policy. Dedicated protocol processors always poll on the protocol event queues and minimize protocol invocation overhead by immediately detecting a protocol event's arrival. Polling, however, may introduce high overheads without the appropriate system support. Message queues, for instance, typically reside on the network interface card which is typically placed on either the memory bus (e.g., in a tightly-coupled parallel machine [HT93,Int93]) or a peripheral bus (e.g., in a machine with commodity desktop nodes [BCF+95]). To check for message arrivals, a protocol thread typically uses an uncached memory operation to read the status of the message queue. Frequent polling using uncached accesses may generate excessive memory traffic. Peripheral device accesses must also cross a memory-to-I/O bus bridge which further increases the memory bus utilization.

Hardware support can provide efficient polling mechanisms. Cachable control registers [MFHW96,Pfi95] allow device registers to be cached in processor caches much like memory. A device is responsible for invalidating a cachable control register from processor caches when the content of the register changes (e.g., as in a message arrival at the head of the queue). In the common case of no messages, polling a cachable control register results in a cache hit, eliminates the memory bus transaction, and incurs minimal overhead.

To make (compute) processors poll the protocol event queues under a multiplexed policy, the system either uses executable editing [LS95] or a compiler [vECGS92] to *instrument* an application's executable with instructions to periodically poll the protocol event queues. Frequent polling through instrumentation, however, may introduce significant delays in the computation. This effect is exacerbated in the absence of system support for efficient polling—such as cachable control registers. System designers often opt to reduce polling overhead by decreasing polling frequency thereby sacrificing protocol invocation speed.

In the absence of support for efficient polling, a system may invoke protocols by delivering an interrupt to a processor. Network interface cards typically provide mechanisms for delivering an interrupt signal to either a processor or an interrupt arbiter. SMPs are typically equipped with programmable interrupt arbitration circuitry [int97]. These devices can be programmed to implement basic interrupt distribution policies. A multiplexed policy may program an interrupt arbiter to distribute interrupts round-robin among the processors. A dedicated policy may program the interrupt arbiter to only distribute the interrupts among the dedicated protocol processors.

If the protocol is executing at the user level, the operating system must also provide mechanisms for delivering the interrupt to the user protocol code. Fast user-deliverable interrupts, however, are only typical of specialized operating systems on parallel computers [RFW93]. User-deliverable interrupts on stock operating systems are extremely slow [TL94] and would be prohibitive for fine-grain parallel applications. Masking/unmasking interrupts may also require invoking system calls which incur high overheads. The operating system, however, can be customized to provide low-overhead user-level interrupt masking schemes [SFL+94,SCB93].

Systems may also provide a hybrid of polling and interrupt mechanisms. To eliminate polling across the memory bus into a peripheral device, a processor can poll on a user-accessible cachable memory location instead. By customizing the network interface device driver, a low-level interrupt routine can signal a message arrival through a user-accessible (cachable) memory location, effectively emulating a cachable control register [Sch97].

3.2.2 Scheduling a Protocol Thread

Both dedicated and multiplexed policies require system mechanisms to schedule a protocol thread's execution. A dedicated policy requires a mechanism to bind a protocol thread to a processor. To guarantee the protocol thread is always executing, the system must also prevent other (computation or protocol) threads from executing on the dedicated protocol processor. Variations of such mechanisms are commonly available in stock operating systems running on commodity SMP desktops and servers. Solaris, for example, provides a system call by which a user can bind a (protocol) thread to a specific SMP processor [SS92].

A multiplexed policy typically schedules a protocol thread for execution when a processor becomes idle—e.g., waiting for synchronization—or when a message arrival interrupts the computation. To schedule and execute a protocol thread, the system must provide mechanisms to swap the processor state corresponding to the computation with that of the protocol thread. Commodity operating systems running on SMPs also typically provide thread packages that allow (dynamic) scheduling of protocol thread execution [SS92].

In many systems employing fine-grain software protocols, a user-level protocol invocation can be as simple as a procedure call. Such systems require the protocol handlers to have short running times and execute to completion upon servicing a protocol event (Chapter 2). As a result, protocol handler execution in these system does not leave stack state (corresponding to nested procedure calls) behind. Rather than use a separate thread stack, the software protocol simply executes on a computation thread's stack much like trap/interrupt service routines in some customized operating systems [RFW93]. Such systems directly invoke a user-level protocol through a procedure call within the computation thread.

3.2.3 Suspending & Resuming the Computation

Both classes of policies require mechanisms to suspend (and subsequently resume) the computation when an application submits a protocol request (e.g., to access remote data). Under a multiplexed policy, a message arrival may also interrupt and suspend the computation. The mechanisms provided to suspend and resume the computation may affect the choice of policy in a given system.

In many systems employing software protocols, the application submits a protocol request through a software handshake [JKW95,SFL+94,CDG+93,BTK90]. Upon a protocol request, the system can simply suspend the computation by making a processor spin on a memory flag waiting for the request to be satisfied. The protocol signals request completion and resumes the computation by writing to the memory flag. A multiplexed policy may also schedule a protocol thread while the computation is waiting for a protocol request to complete. To detect a protocol request completion and resume the computation, the system must provide mechanisms for a protocol thread to poll on the request completion (memory) flag, as well as, the protocol event queues.

Some fine-grain DSM systems use hardware support to detect when a processor is accessing remote data [Pfi95,SFL+94]. In many such systems, hardware detects accesses to remote data by inspecting transactions on the memory bus. Upon detecting a remote data access, some systems generate a bus error exception in response to the memory bus transaction [Rei96]. Upon a bus error, the system saves the processor state—e.g., the condition code registers and the program counter—necessary to resume the computation exactly at the point of the exception. These systems can also use a simple software handshake for resuming the computation by simply making the processor spin on a memory flag inside the bus error exception routine waiting for the remote data to arrive.

Modern commodity microprocessors do not always support precise (or even restartable) bus error exceptions [SP88]. In the absence of precise bus errors or to reduce communication overhead, some fine-grain DSM systems suspend and resume the faulting (compute) processor directly in hardware [Rei96]. These systems suspend/resume the computation by masking/unmasking the faulting processor from memory bus arbitration. Such a mechanism would preclude using a multiplexed policy because the faulting processor can no longer access memory until the access violation is satisfied (i.e., the remote data is fetched).

Under a multiplexed policy, a message arrival may require suspending the computation to invoke a protocol. Message arrivals invoke a protocol either through interrupts or instrumented polling. An interrupt suspends the computation using system exception mechanisms similar to a bus error. Much like systems with a software handshake for protocol request submission, instrumented polling suspends the computation by simply invoking the protocol scheduling mechanisms directly in software. In both the interrupt- and polling-based systems, the scheduling mechanisms can immediately resume the computation (i.e., schedule the computation thread for execution) once the protocol thread finishes draining the event queues and becomes idle.

3.2.4 Mechanisms for Choosing a Policy

An operating system may choose one protocol scheduling policy statically at boot time or allow users to choose a policy at runtime. Many parallel systems provide mechanisms to change and invoke protocols that are optimized for specific communication patterns during an application's execution [KDCZ93,CBZ91]. Likewise, systems can provide application-level mechanisms to change and invoke a scheduling policy optimized for a given communication phase. During a computation-intensive phase, an application may choose to use a multiplexed policy allowing all processors to contribute to computation, while in a communication-intensive synchronous phase a dedicated policy may be preferable allowing the system to pin a protocol thread to a specific processor eliminating protocol thread migration overhead.

3.3 Design & Functionality Requirements

Systems may choose a policy over another because of several system design and functionality requirements. A dedicated policy may be preferred over a multiplexed policy when specialized hardware resources (e.g., hardware support for fine-grain shared memory) are only accessible to specific SMP-node processors [CAA+95]; multiplexing such processors may create a load imbalance in the computation thereby lowering performance.

Some parallel environments require protected messaging between individual processors across machine nodes [LHPS97,Int93]. To provide protected communication, such systems allow access to protocol resources (e.g., such as the network interface board) through the operating system. A multiplexed policy in such an environment would require a system call upon every invocation of the protocol thread and may incur prohibitively high overheads. A dedicated policy, however, would eliminate the system call overhead by always executing a protocol thread on a dedicated processor in system mode.

Statically scheduling processors to execute protocol threads may also be preferable from both design and manufacturing standpoint. Dedicated protocol processors eliminate the hardware and software requirements for preempting the computation and invoking the protocol threads thereby reducing design complexity and turn-around time.

3.4 Policy Performance Trade-Off

There are several application and system characteristics that impact performance under a specific scheduling policy. A multiplexed policy favors applications that are computation-intensive and systems in which the overhead of (dynamically) scheduling and executing a protocol thread accounts for a negligible fraction of overall execution time. Conversely, a dedicated policy favors applications in which communication accounts for a significant fraction of overall execution time. Systems with high protocol scheduling overheads also benefit from a dedicated policy. This section enumerates factors that have a significant impact on system performance under the policies.

3.4.1 Application Characteristics

Applications with low communication-to-computation ratios (e.g., such as dense matrix codes) can exhaust all the computational resources available on a node. These applications exhibit good speedups even with heavy-weight page-based DSM protocols [KDCZ93,CBZ91]. Such applications would underutilize dedicated protocol processors, wasting processor cycles that could have otherwise contributed to computation.

Applications with bursty communication patterns are also likely to benefit from a multiplexed policy. Examples of such applications are those in which communication and computation proceed in synchronous phases. Bursty communication is also characteristic of shared-memory applications using software prefetching, relaxed memory consistency models, and customized application-specific protocols. Bursty communication allows the processors to schedule protocol thread(s) once for the duration of the communication, thereby eliminating overhead.

The choice of policy also depends on the communication and computation granularity in an application. Several classes of important applications exhibit fine-grain and asynchronous communication. Examples of such applications are gravitational N-body simulation [BH86], cholesky factorization [WOT+95], and fine-grained sparse-matrix methods [CSBS95]. These applications exploit parallelism by overlapping fine-grain communication and computation among machine nodes. By balancing the load between the computation and protocol processors, fine-grain applications with asynchronous communication favor a dedicated policy. Dedicated protocol processors also eliminate the protocol thread scheduling overhead which is frequent in such applications.

3.4.2 Overhead in a Multiplexed Policy

There are two types of overhead in a multiplexed policy: scheduling overhead to invoke a protocol thread upon arrival of a protocol event, and cache interference overhead between the computation and protocol thread. Processors either voluntarily schedule a protocol thread when they become idle (e.g., while waiting for a protocol request or at a synchronization), or involuntarily through a scheduling invocation mechanism such as interrupts or instrumented polling.

Naive implementations of user-level interrupts and instrumented polling may incur high overheads (see Section 3.2) and result in poor performance under a multiplexed policy. Large-scale SMPs increase the likelihood of one or more processors being idle, significantly reducing the frequency of interrupts [FW97c,KS96]. Instrumented polling, however, always incurs a minimum overhead of checking for protocol events.

Executing protocol threads on a compute processor may pollute the processor's instruction [MPO95] and data [PC94] cache hierarchy. A dedicated policy has the advantage of providing a separate set of instruction and data caches for the protocol thread to use, avoiding cache interference with computation. Cache interference under a multiplexed policy, however, may be minimal depending on the number of cache references made the protocol thread. Fine-grain communication protocols (e.g., a coherence protocol in fine-grain DSM) have handlers with very short running times and typically only access a fine-grain (e.g., 32-256 bytes) memory block and update the corresponding protocol state. As such, these protocols are likely to incur minimal cache interference under a multiplexed policy.

Network interfaces equipped with data caches (such as Typhoon's block buffer [RPW96] or CNI's cachable queues [MFHW96]) allow protocols to leave the protocol data in the network interface cache. A requesting (computation) processor may directly load the data from the network interface cache reducing the protocol thread's cache interference with computation. Protocols may also induce a positive cache interference [FW97a] under a multiplexed policy by leaving the requested data in the requestor's data cache if the requesting processor is also the processor running the protocol thread [FW97c,SFH+97].

3.4.3 Multiprocessing & Multithreading

Both the degree of multiprocessing (i.e., the number of processors per node) and multi-threading (i.e., the number of threads running on a single compute processor) have several effects on the policy trade-off. More processors increase the likelihood that at least one processor is idle (e.g., waiting for a protocol response). Under a multiplexed policy, such a processor voluntarily schedules and runs a protocol thread, eliminating the scheduling overhead upon arrival of a protocol event. With parallel protocol dispatch support (e.g., using the PDQ mechanisms), multiple processors can contribute to protocol execution, significantly increasing the communication bandwidth. Dedicated protocol processors, however, also save the cache interference overhead which may improve performance in the presence of high bus utilization.

By parallelizing the computation *within* a node, multiple compute processors also increase the *apparent* communication-to-computation ratio. Multi-threading the compute processors also increases the apparent communication-to-computation ratio by overlapping communication with computation among multiple threads [Aga92,GHG+91]; a processor can schedule and execute one computation thread while waiting for a protocol request for another. A dedicated policy increases the communication bandwidth by eliminating the protocol scheduling and execution overhead favoring a higher apparent communication-to-computation ratio. Large-scale SMP nodes also make a dedicated policy advantageous by reducing the opportunity cost (in lost computation) of the dedicated protocol processors.

3.4.4 Protocol Weight

Protocol weight is a qualitative measure of the protocol's execution time. It is a function of the protocol complexity, the architecture of the network and the network interface device. Protocol weight affects the policy trade-off because for heavy-weight protocols (i.e., protocols with long running times) the overheads saved by a dedicated policy become an insignificant fraction of the overall communication time. Thus, a dedicated policy should be more beneficial for light-weight protocols (e.g., active-message-based protocols [JKW95,CDG+93]) than for heavy-weight protocols (e.g., page-based DSM [KDCZ93,CBZ91,LH89]). This runs counter to the common intuition that dedicating a protocol processor helps off-load heavy-weight protocols from the compute processor.

Protocol complexity is a function of the high-level abstractions required by an application. An application may require simple point-to-point messaging as in Active Messages, a shared global address space as in DSM, or more general models in which the protocol performs arbitrary computation as in RPC [BN84]. Many applications and systems also opt for higher protocol complexity to reduce the frequency of messaging. Adaptive cache coherence shared-memory protocols, for instance, minimize communication by monitoring the sharing behavior of data at runtime and selecting one out of many protocols suitable for enforcing coherence on a given data item [FW97b,CF93,SBS93,BCZ90].

Network interface cards may provide hardware support for high-level abstractions. For instance, CNI's [MFHW96] hardware moves data in cache block granularities between the network interface card and processor caches. Typhoon-1's [RPW96] shared-memory hardware atomically moves a fine-grain memory block between memory (or processor caches) and the network and updates the corresponding protocol state. Similarly, DMA engines [BDFL96] decouple constructing a message (performed by the protocol) from the actual data transfer (performed by the engine). The above mechanisms all reduce protocol complexity and thereby protocol weight.

Low-level messaging services also contribute to protocol complexity. Parallel applications typically require low-level messaging services such as checksumming, reliable delivery, in-order delivery, flow control, and fragmentation and reassembly. Tightly-coupled distributed-memory machines usually provide such services in hardware both at the level of the network switch and the interface card [HT93]. Commodity networks, however, typically do not implement these services entirely in hardware and require software protocols to provide some of the functionality [KC94]. These low-level protocols either run on a high-speed embedded processor on the network interface card [BVvE95] or must execute on a host (node) processor along with higher-level abstractions [Sch97].

The network interface card's location with respect to a processor also affects the protocol weight. Tightly-coupled parallel machines have traditionally positioned the network interface card on the memory bus to provide a low-latency access path from the processors to the network. Current commodity network interface cards are placed on peripheral buses which require crossing the memory bus through an I/O bridge to access a device. Peripheral addresses are also typically protected. As such, accessing the network through a user-level protocol requires copying message data multiple times among protection boundaries [Sch97]. Network interface cards placed on peripheral buses can significantly increase the protocol weight.

3.5 Policy Cost-Performance Trade-Off

System designers often use cost-performance rather than performance as the primary metric for evaluating a design. Cost-performance is important when comparing the policy trade-off because adding one or more dedicated protocol processors can always improve performance given large enough SMP nodes. Cost-performance, however, only improves if the performance improvement is large enough to offset the additional cost of the dedicated processors [WH95,FW94].

Whereas the manufacturing cost of computer products is typically related to the cost of components, cost from a customer's perspective is related to price which is also dictated by market forces [HP90]. High-performance products, for instance, tend to target smaller markets and therefore carry larger margins and higher price premiums. Parallel machines can either use small-scale desktop SMPs or medium- to large-scale SMP servers as building blocks. Depending on the degree of multiprocessing, SMP products can belong to either a low-margin desktop or high-margin server market. Adding (dedicated) protocol processors to high-premium SMPs may considerably improve performance while not significantly increase the cost, resulting in a cost-effective system.

3.6 Related Work

Process and thread scheduling has been a topic of considerable interest in parallel systems research community [ABLL92,VZ91,DBRD91,SL90,TG89]. Similarly, researchers in the networking community have extensively studied scheduling policies for parallelized network communication protocols [SKT96,BG93,Kai93,HP91]. Other researchers have studied scheduling VM-based software DSM protocols [KS96,ENCH96] on SMP clusters.

This thesis primarily focuses on scheduling fine-grain communication protocols. In the past, several systems providing software fine-grain communication protocols on SMP clusters have employed a dedicated scheduling policy because of special hardware resources available only to a particular SMP-node processor [CAA+95], or to provide efficient protected communication by always running the protocol thread in system mode [Int93,LHPS97], or to simply eliminate the context switch overhead between computation and the protocol thread [RPW96].

This thesis is the first to propose a taxonomy for protocol scheduling policies. The thesis also evaluates system and application characteristics that affect performance and cost-performance trade-offs between the two classes of taxonomy.

3.7 Summary

Rather than provide embedded processors on a custom device to execute software protocols, some cost-effective parallel computers execute the protocols on the node's commodity processor. This chapter evaluates scheduling policies for software protocols in a cluster of SMPs. A taxonomy of scheduling policies defines two classes of policies:

- A dedicated policy (statically) schedules one or more SMP-node processors to only execute protocol threads,
- A multiplexed policy allows all processors to perform computation and (dynamically) schedules one or more protocol threads to execute when processors become idle (e.g.,

due to waiting for a remote request or synchronization operation) or upon arrival of a protocol message.

This chapter describes in detail the two classes of scheduling policies. Choice of policy may depend on the required mechanisms to schedule and invoke protocols, the system design and functionality requirements, or the performance and cost-performance trade-offs between the scheduling policies. The chapter identifies the key application and system characteristics that affect the performance trade-offs between the two policies:

- Application's characteristics such as communication-to-computation ratio and burstiness in communication,
- Scheduler invocation and thread migration overhead in a multiplexed policy,
- Degree of multiprocessing or multi-threading,
- Protocol weight which is a function of protocol's complexity and network architecture and speed.

Chapter 4

Scheduling Policies for a Single-Threaded Protocol

Chapter 2 and Chapter 3 presented a taxonomy for software protocol execution semantics and scheduling policies respectively. This chapter presents an evaluation of protocol scheduling policies for single-threaded execution of fine-grain software protocols on a cluster of small-scale SMPs.

Small SMP systems—such as the Intel Pentium-Pro-based servers—are becoming widely available, making them attractive building blocks for parallel computers [LC96,WGH+97,CA96]. Some cost-effective multiprocessor designs implement communication protocols in software and use little or no custom hardware support for protocol execution [SGA97,SFH+97]. These machines interconnect SMP nodes using relatively simple commodity network interfaces [BCF+95] and perform most protocol processing in a single protocol thread running on a regular SMP-node processor.

Much like other OS services, communication protocols employ a scheduling policy on SMP processors. One SMP processor may either be statically scheduled (i.e., dedicated) to run only the protocol thread for the duration of an application's execution, or any SMP-node processor performing computation may be dynamically scheduled (i.e., multiplexed) to instead run the protocol thread for a while.

Several proposed and/or implemented systems dedicate one processor specifically for protocol processing because of special hardware resources available only to a particular SMP-node processor [CAA+95], or to provide efficient protected communication by always running the protocol thread in system mode [Int93,LHPS97], or to simply eliminate the context switch overhead between computation and the protocol thread [RPW96].

While a dedicated protocol processor can improve communications performance, it provides little benefit for compute-bound programs. These applications would rather use the dedicated processor for computation. In a recent experiment, Womble, et al., demonstrated that using the Paragon's protocol processor for computation (via a low-level cross-call mechanism under SUNMOS) improved performance on LINPACK by more than 50% [WG94]. Similarly, others have shown that a dedicated protocol processor provides little benefit for systems with large communication latencies and overheads as in ATM [KS96] or HIPPI [ENCH96] networks.

In this chapter, I ask the question: *"when does it make sense to dedicate one processor in each SMP node specifically for protocol processing?"* The central issue is when do the overheads eliminated by a dedicated protocol processor offset its lost contribution to computation? I address this question by examining the performance and cost-performance trade-offs of two scheduling policies for a single-threaded protocol:

- *Fixed*, a dedicated policy where one processor in an SMP node is dedicated for protocol processing, and
- *Floating,* a multiplexed policy where all processors compute and alternate acting as protocol processor.

To evaluate the two scheduling policies, this study models a fine-grain DSM on an SMP cluster. The system implements intra-node communication through the MOESI coherence protocol on the SMP bus, and uses a software DSM coherence protocol to extend the fine-grain shared-memory mechanisms of an SMP bus across a cluster. The coherence protocol used, *Stache* [RLW94], is an invalidation-based full-map directory protocol which uses a portion of each node's main memory to cache remote data much like S-COMA [HSL94]. Stache implements caching by allocating data at page granularity but maintaining coherence (within the page) at cache block (e.g., 32-128 bytes) granularity. Although the experiments in this study are in the context of Stache, the results are applicable to more general software communication protocols.

Much like other simple request/reply protocols [CDG+93,JKW95], Stache implements protocol actions using Active Messages [CDG+93]. Active messages require the execution of the handlers to appear atomic. Stache simply guarantees this requirement by executing the protocol in a single thread on every node.

The next section describes in detail the protocol processing mechanisms and the network interface architecture. Section 4.2 describes the two protocol processing policies in more detail. Section 4.3 present performance results from a microbenchmark and a macrobenchmark experiment, respectively and uses a simple cost model to evaluate policy cost/performance. Section 4.4 presents a discussion of related work. Finally, Section 4.5 concludes the chapter.

4.1 Protocol Execution & Scheduling Mechanisms

Fine-grain software coherence protocols require mechanisms for detecting a remote block miss and a subsequent dispatch of a protocol handler. Protocol actions are also invoked through messages across machine nodes and require mechanisms for sending a message and dispatching the corresponding protocol handler upon receiving the message. Systems may provide mechanisms for remote block miss detection and protocol handler dispatch either in software or hardware [SFL+94]. While the results of this study are largely independent of whether these mechanisms are implemented in hardware or software, we assume a hardware implementation via a Typhoon-1 board [RPW96].

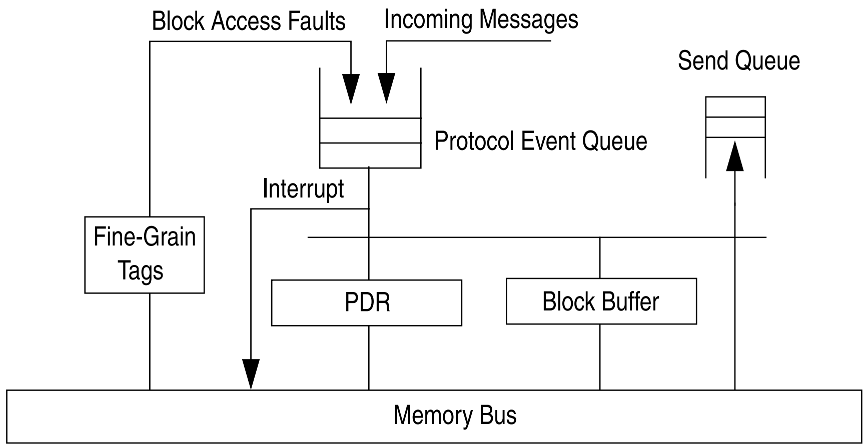


Figure 4-1. The Typhoon-1 network interface. The network interface contains a protocol dispatch register, a block buffer, and a message send queue. There is a single SRAM device maintaining the fine-grain tags for cached remote data, and a protocol dispatch queue recording block access faults and incoming messages on the node.

Figure 4-1 illustrates the architecture of a Typhoon-1 network interface. An SRAM device maintains the fine-grain tags used to enforce access control semantics on shared-memory loads and stores that miss in the cache. The board snoops on cache fills appearing in the form of memory transactions on the bus and performs a tag lookup. Upon access violation, the board enters the faulting address, the access type, the tag value, and the address of the protocol handler to be dispatched into a *protocol event queue*. The protocol event queue also maintains the incoming (active) messages from other nodes.

The protocol thread polls on a *protocol dispatch register* (PDR) which is the head entry in the protocol dispatch queue. The protocol dispatch register is cachable control register [RPW96,MFHW96] located on the Typhoon-1 board. Cachable control registers eliminate excessive poll traffic on the memory bus by allowing a processor to poll on a device register directly in its cache. Typhoon-1 notifies a processor when an entry has been inserted in the protocol event queue by invalidating the cached copy of the protocol dispatch register.

Typhoon-1 is also equipped with a block buffer; a small direct-mapped cache that serves as an intermediary storage device for moving memory blocks between the node's memory hierarchy (e.g., processor caches or main memory) and the message queues. Removing/

placing remote data from/into a node also typically involves modifying the state of the block maintained by the fine-grain tags. Shared-memory access semantics dictates that moving data and updating the corresponding tag value should appear to execute atomically. The block buffer implements this atomic operation directly in hardware.

To allow dynamic scheduling of the protocol thread upon message arrival, Typhoon-1 also allows invoking interrupts on the memory bus. A memory-mapped interrupt arbiter device distributes interrupts among the processors in a round-robin fashion.

To avoid invoking high-overhead system calls for masking/unmasking interrupts in a critical section the system uses a low-overhead software interrupt masking scheme [SCB93]. Each computation thread is assigned a user- and system-accessible flag in memory to set and reset upon entering and exiting a critical section. When an interrupt arrives in the middle of a critical section (determined by checking the flag), the system sets an interrupt-pending flag and masks further interrupts. Upon exiting the critical section, a computation thread checks for pending interrupts. If an interrupt is pending, the computation thread clears the pending flag, invokes the protocol code to handle messages, and unmasks interrupts. This interrupt masking scheme optimizes the overhead in the common case of no interrupts in a critical section to a small number of memory accesses.

4.2 Protocol Scheduling Policies

In this study, we examine two scheduling policies for protocol processing: *Fixed* and *Floating*. Because Stache is single-threaded, each node is limited to one processor executing protocol events at any one time. Regardless of the policy we say that this processor is *acting* as protocol processor.

The Fixed policy dedicates one processor of a multiprocessor node to perform only protocol processing. The dedicated protocol processor executes all the remote miss and active message handlers. By always polling the network when otherwise idle, the protocol processor eliminates the need for message interrupts or polling by the compute processor(s).

The disadvantage of dedicating a protocol processor is that it may waste cycles that could have productively contributed to computation. The Floating policy addresses this dilemma by using all processors to perform computation; however, when one becomes idle (e.g., due to waiting for a remote request or synchronization operation) it assumes the role of protocol processor. Since all processors may be computing, either interrupts or periodic polling are required to ensure timely handling of active messages. On the other hand, once a processor assumes the role of protocol processor, it acts much like a dedicated protocol processor. We use the term *Single* to refer to the special case of a single processor (per node) performing all protocol processing as well as all computation.

4.3 When does dedicated protocol processing make sense?

In this study, I pose the question: “when does dedicated protocol processing make sense?” I address this question by evaluating when one of our two protocol processing policies performs better or is more cost-effective than the other. While there are many factors—including system software complexity, and protection [LHPS97]—I believe that performance and cost-performance are important.

To quantify cost-effectiveness, I use the simple cost model from Wood and Hill [WH95]. A change, e.g., adding a second processor, is cost-effective if and only if the increase in cost (or costup) is less than the increase in performance (or speedup). In this paper, we say a system is *cost-effective* if its cost-performance ratio is less than a uniprocessor node’s. A system is *most cost-effective* if it achieves the lowest cost-performance ratio. This simple cost model assumes that a processor represents 30% of the cost of a uniprocessor node.[1] Thus, a two-processor node and a five-processor node have costups of 1.3 and 2.2, respectively.

1. The incremental cost of an additional processor varies greatly depending on the processor, memory hierarchy, peripherals, and the overall system cost per node. In many cases the incremental cost may be less than 30% which will shift cost-performance in favor of Fixed.

To answer “when” one policy is better than another, I use results from microbenchmark and macrobenchmark experiments to compare policy performance. The experiments vary application and system characteristics that have a first-order impact on policy performance (as discussed in Chapter 3).

4.3.1 Methodology

I use the simulation methodology and system parameters described in Section 1.3.1 to simulate an SMP cluster interconnected with a Typhoon-1 network interface. This study models a memory system characteristic of small-scale SMP desktops and therefore assumes a single memory bank with a four-entry write-buffer in the memory controller. Unless specified otherwise, the simulator assumes an interrupt overhead and a bus error trap overhead of 200 cycles, characteristic of carefully tuned parallel computers [RFW93].

4.3.2 Microbenchmark Experiment

In this section I evaluate the Fixed and Floating policies using two simple synthetic benchmarks. We base our benchmarks on a simple request/reply protocol, similar to that employed by many parallel computing paradigms [CDG+93,JKW95,CBZ91,RLW94]. The benchmark times the execution of a tight loop running on a two-node machine. Each iteration alternates between computing and issuing a remote request using a simple request/reply protocol. To induce cache effects, computation is interleaved with uniformly random accesses to a (private) processor-specific segment of the address-space. The size of the segment is equal to the size of the processor cache. The compute processor caches warm up before the start of measurements.

The experiment uses two request/reply protocols with different protocol weights. A *null-handler* protocol represents the lightest-weight protocol achievable in the simulated system. The protocol handlers do nothing but send the appropriate active message, i.e., the reply handler simply sends a null message back to the requester.

A *fetch-block* protocol is representative of the medium-weight protocols needed to support fine-grain distributed shared-memory systems [JKW95,SFL+94]. I do not consider a heavy-weight protocol, e.g., page-based DSM, since prior work indicates that a dedicated protocol processor will be of little use [KS96,ENCH96]. The processors randomly request a 128-byte block of data from the private segment of a remote processor. The protocol handlers manipulate the memory block state in a protocol table. Both the data block transfer and accesses to protocol table contribute to cache pollution.

Let L_{min} to be minimum round-trip latency under the Fixed policy. Under the modeled system assumptions, the protocol round-trip times are 1.7 µs for the null-handler protocol and 6.2 µs for the fetch-block protocol. We vary the following parameters in the experiment:

C	= mean computation time between requests,
U	= thread compute-utilization in the absence of protocol contention $(C/(C+L_{min}))$,
O_{int}	= overhead of handling an interrupt.

The experiment uses an exponential random stream with mean C to generate computation times, and adjusts C to derive various values for U. To vary O_{int}, the system delays executing a thread upon an interrupt for a fixed number of cycles. The number of iterations in a loop is inversely proportional to the number of compute processors per node, e.g., Floating on a two-processor node and Fixed on a four-processor node execute half and one-third as many iterations as Single, respectively.

Figure 4-2 (left) compares the performance for the null-handler protocol in one and two-processor node machines. The figure plots execution times of Single and Floating normalized to Fixed as thread compute-utilization increases. Points above the horizontal line indicate that Floating (Single) performs worse than Fixed. The thick and thin lines depict high and low interrupt overheads, respectively. The graphs for Single (solid curves) illustrate the intuitive result that communication-intensive programs (small U) benefit more from a dedicated protocol processor than computation-intensive programs (large U). When the program becomes communication-bound ($C \ll L_{min}$), however, the compute

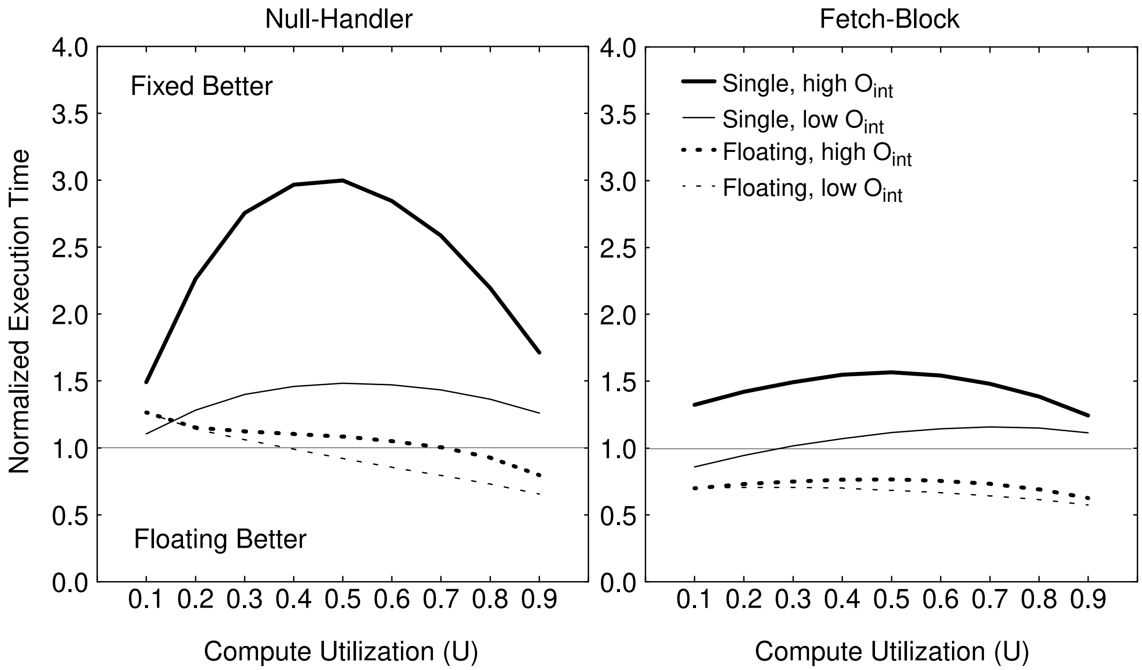


Figure 4-2. Relative performance with varying interrupt overhead. The figure compares execution times of Fixed and Floating against thread compute-utilization (U). The graphs plot execution times of Single and two-processor Floating normalized to two-processor Fixed for two values of interrupt overhead (O_{int}); low and high O_{int} correspond to values of 0.5 µs (200 cycles) and 5 µs (2000 cycles) respectively. Values over the horizontal line at 1 indicate better performance under the Fixed policy.

processor in Single becomes idle and acts like a protocol processor, reducing the number of taken interrupts. The graphs also indicate that, when interrupt overhead is high, even a small number of interrupts severely impacts the execution time.

The dashed curves plot the normalized execution time for a two-processor node under the Floating policy. With high interrupt overheads, the Floating policy behaves like the Fixed policy; the two (compute) processors alternate acting as the protocol processor eliminating the interrupt overhead. Protocol thread migration overhead, however, slightly reduces performance under Floating relative to Fixed. With low interrupt overheads, there is little benefit from a dedicated protocol processor, but potential gain from improving computation time. Under Floating, both processors perform computation, resulting in significantly better performance at higher compute-utilizations. This is not surprising since the microbenchmark is perfectly parallelizable.

Figure 4-2 (right) compares the performance of the policies for the fetch-block protocol. The figure corroborates the intuition that a dedicated protocol processor is more beneficial for light-weight protocols than for heavy-weight protocols. The result follows from the observation that Fixed does best when the interrupt overhead is much greater than the round-trip latency ($O_{int} >> L_{min}$). This result suggests that dedicated protocol processors may become more attractive as interrupt latencies go up (due to faster processors) and protocol weights go down (due to faster network interfaces).

This graph illustrates the surprising result that for a communication-bound program and low interrupt overhead, Single outperforms Fixed. This occurs because our synthetic protocol always reads message data into the protocol processor's cache. Under Fixed, the compute processor always misses on message data, resulting in a cache-to-cache transfer. Conversely, under Single, there is only one cache, so the transfer is eliminated. Fetch-block is a simple request/reply protocol and does not take advantage of Typhoon-1's block buffer for direct cache-to-cache transfer of the data block.

Unlike the null-handler protocol, the Floating policy maintains its advantage over Fixed even at low compute-utilizations. Overheads in the fetch-block protocol account for an small fraction of communication time. Moreover, at low compute-utilizations the extra compute processor in Floating parallelizes communication by doubling the number of outstanding requests per node. Much as in Single, it is also likely that an acting protocol processor under Floating reads message data into the requesting compute processor cache, eliminating the extra data transfer. The combined effect of the above improves Floating's performance over Fixed at low compute-utilizations.

4.3.2.1 Multiple compute processors per Node. More processors per node helps Floating by increasing the likelihood that an idle processor is acting as protocol processor. The

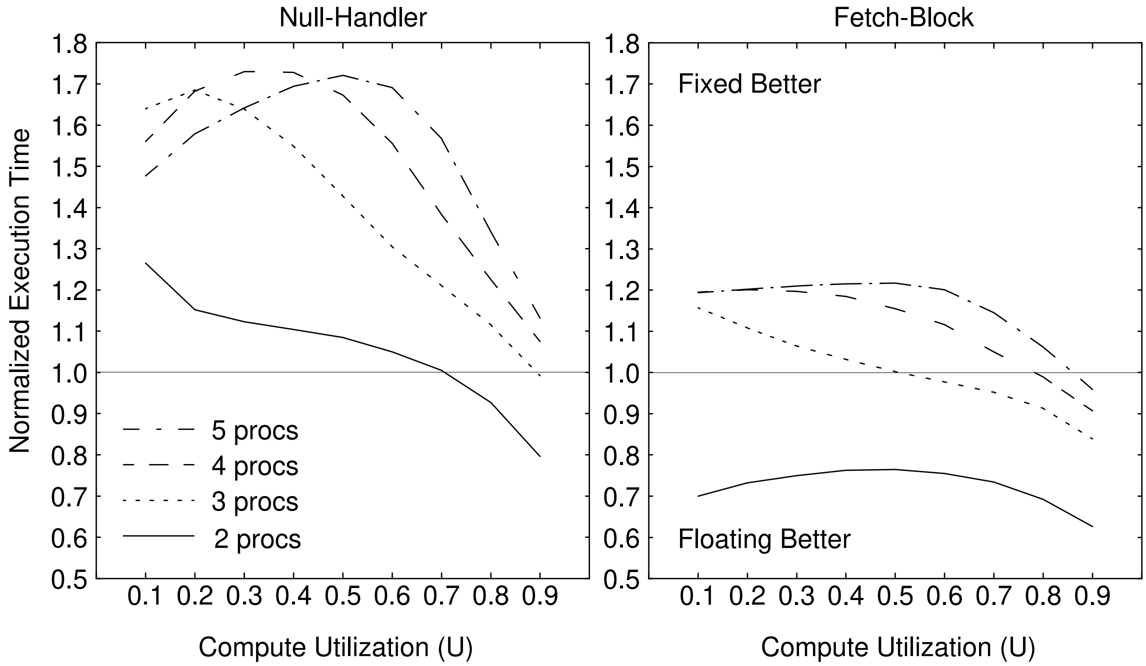


Figure 4-3. Relative performance with varying number of processors/node. The figure compares execution times of Fixed and Floating against thread compute-utilization (U). The graphs plot Floating's execution time normalized to Fixed for various number of processors per node (O_{int} = 0.5 μs). Values over the horizontal line at 1 indicate a better performance under the Fixed policy.

added benefit of an extra compute processor, however, diminishes with a larger number of processors. Multiple compute processors also increase the contention for the single protocol processor. Under low compute-utilization, Floating approximates Fixed, since an acting protocol processor eliminates the interrupt overhead. Fixed, however, provides better throughput by also eliminating the overheads associated with protocol thread migration.

Figure 4-3 (left) plots execution time for the null-handler protocol under the Floating policy normalized to Fixed, while varying the number of processors per node. Fixed generally outperforms Floating particularly with three or more processors per node; greater demand for protocol processing with a larger number of processors makes communication the bottleneck. Because the dedicated protocol processor minimizes protocol processor occupancy, Fixed provides greater throughput and can support a larger number of compute processors. The graphs indicate that performance under Fixed can significantly improve, by up to 75%, over Floating. With four or more compute processors, Floating fails to improve performance over Fixed even at a 90% compute-utilization.

At low compute utilizations, an increase in the number of processors increases Floating's performance relative to Fixed. Multiple compute processors running communication-bound applications increase protocol processor utilization and result in queueing at the protocol processor. Fixed experiences queueing more quickly than Floating because request rates in Floating remain lower than those in Fixed; an acting protocol processor under Floating must return to computation before it can contribute to request traffic. As such, the system begins to exhibit queueing with a fewer number of compute processors and at a higher compute-utilization under Fixed than Floating. Queueing delays under Fixed reduce the performance gap between the two policies.

Compute-intensive programs take advantage of the extra compute processor in Floating to improve computation time. An increase in the number of processors, however, gradually diminishes Floating's advantage over Fixed because the added benefit of an extra compute processor becomes insignificant.

Figure 4-3 (right) plots the same graphs for the fetch-block protocol. Much like the null-handler protocol, Fixed outperforms Floating when protocol processor utilization is high, i.e., there are more than two processors per node and compute-utilization is low. Because overhead in the fetch-block protocol accounts for a small fraction of protocol occupancy, Fixed loses its performance advantage over Floating. The graphs indicate that Fixed at most improves performance by 20% over Floating.

4.3.2.2 Cost/Performance. Cost-performance (rather than performance) becomes a more meaningful metric from a design perspective when increasing the number of processors per node. Adding processors may always result in incremental performance improvement. The performance improvement, however, may not be high enough to justify the additional cost of processors. Similarly, cost-performance also serves as a better metric for compar-

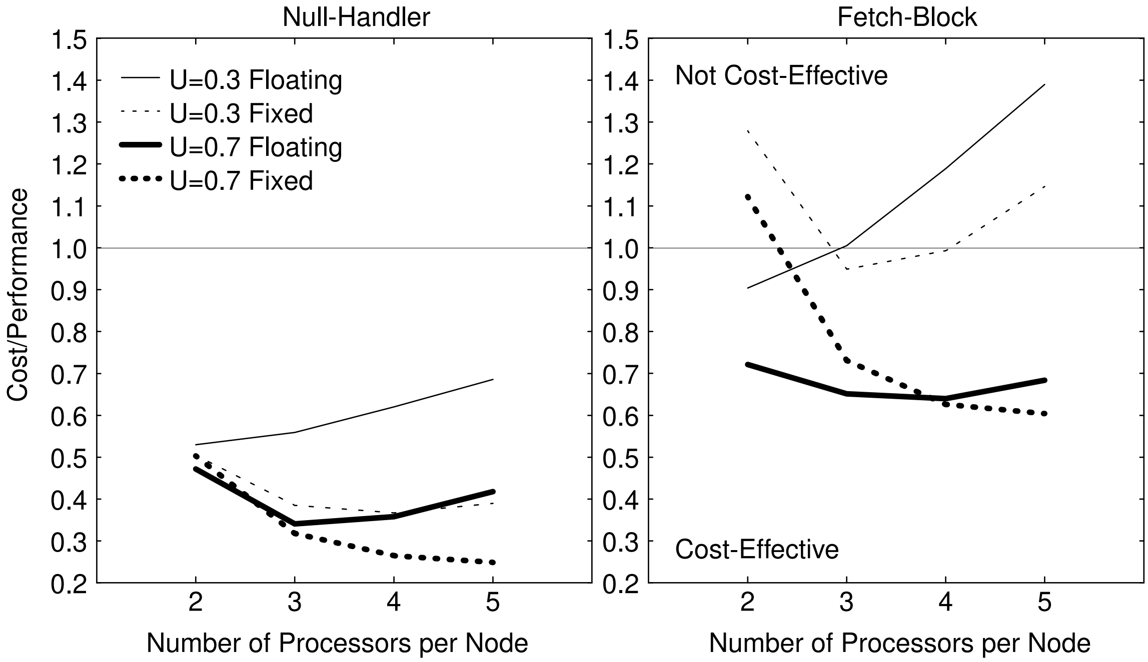


Figure 4-4. Relative cost-performance. The figure plots cost-performance of Fixed and Floating (O_{int} = 0.5 µs) against varying the number of processors per node for two values of compute-utilization (U = 0.3 and U = 0.7). Costups and speedups are calculated with respect to a uniprocessor node (Single). The graphs assume that the cost of a processor is 30% of the cost of a uniprocessor node. Values over the horizontal line at 1 indicate design points that are not cost-effective.

ing scheduling policy trade-offs. Given a large enough number of processors per node, Fixed will outperform Floating because communication eventually becomes the bottleneck. At this point, however, the resulting system may not be the most cost-effective design point.

Figure 4-4 (left) illustrates cost-performance for the null-handler protocol. The figure plots cost-performance ratio where 1 represents a uniprocessor node. Values under the horizontal line (at 1) correspond to systems that are cost-effective—i.e., systems with better (lower) cost-performance than a uniprocessor node. The figure examines both policies at two compute-utilizations, against the number of processors per node. Adding up to four processors to a uniprocessor node under either policy results in a cost-effective system. Interrupt overhead significantly increases execution time in Single for the null-handler protocol allowing Fixed and Floating to improve performance by virtually eliminating interrupts.

Not surprisingly, the Fixed policy always provides the most cost-effective system for the null-handler protocol. Figure 4-3 (left) indicated that Fixed always improves performance over Floating for low ($U = 0.3$) compute-utilizations. For moderate compute-utilizations ($U = 0.7$), Fixed outperforms Floating with three or more processors per node. Because of high compute-utilizations, the benchmark continues to exhibit reasonable speedups with a larger (> three) number of processors per node. A larger number of processors also reduce the relative cost-increment from an additional processor. The combined effect drives cost-performance lower under Fixed.

Figure 4-4 (right) illustrates the cost-performance graphs for the fetch-block protocol. Unlike the null-handler protocol, adding processors to a node is not always cost-effective. Interrupt overhead does not have as high of an impact on Single's performance because of high protocol occupancies in the fetch-block protocol. When compute-utilization is low ($U = 0.3$), Floating with three or more processors and Fixed with either two or five or more processors per node result in systems that are not cost-effective. A higher compute-utilization ($U = 0.7$), however, results in high speedups with a larger number of processors per node, offsetting the cost-increment of the additional processors, resulting in systems that are cost-effective.

Unlike the null-handler protocol, Fixed does not always result in the most cost-effective system. At the lower compute-utilizations, Floating (with two processors per node) results in the most cost-effective system. At high compute-utilizations, however, the system continues to achieve lower cost-performance with four or more processors per node. Fixed, however, outperforms Floating with a large number of processors per node (Figure 4-3 (right)) and therefore results in the most cost-effective system.

Table 4.1: Applications and input sets.

Benchmark	Description	Input Set
barnes	Barnes-Hut N-body simulation	16K particles
em3d	3-D electromagnetic wave propagation	76K nodes, degree 5, 15% remote, distance of 16, 10 iters
gauss	Gaussian elimination using a linear system of equations	1920x1920 matrix
lu	Blocked dense LU factorization	1200x1200 matrix, 16x16 blocks
water-sp	Spatial water molecule force simulation	4096 molecules

In summary, when interrupt overhead is high with respect to protocol occupancy—as in light-weight protocols, Fixed is likely to result in the most cost-effective system independent of an application's communication-to-computation ratio. In contrast, when interrupt overhead is low, Fixed results in the most cost-effective system for applications with high compute-utilizations.

4.3.3 Macrobenchmark Experiment

Although microbenchmark analysis helps develop intuition about relative performance, it makes many simplifying assumptions. For example, the experiments ignored synchronization, burstiness of communication, cache effects due to large data sets, and bandwidth limitations of the memory bus. In this section, I re-examine the policies in the context of a network of 16 multiprocessor workstations, each with five processors.

Table 4.1 lists the applications and corresponding input data sets we use in this study. *Barnes, lu*, and *water-sp* are all from the SPLASH-2 suite [WOT+95]. *Em3d* is a shared-memory implementation of the Split-C benchmark [CDG+93]. *Gauss* is a simple shared-memory implementation of a gaussian elimination kernel [CLR94].

This study uses the Stache fine-grain DSM coherence protocol to extend the shared-memory abstraction of an SMP across a cluster. Stache, uses Typhoon-1's block buffer to move memory blocks directly from the network into the requesting compute processor's cache, eliminating the transfer to the protocol processor cache. Such a mechanism under Floating either reduces cache pollution or increases cache-to-cache transfers depending on whether the acting protocol processor is also the requesting compute processor (as discussed in Section 3.4).

Unless specified otherwise, this study uses a 128-byte Stache protocol. The measurements indicate a minimum running time of 1490 processor cycles (3.73 μs) of round-trip latency for simply fetching a memory block from a remote node. While this latency is not competitive with high-end all-hardware implementations of fine-grain DSM (such as the SGI Origin 2000 [LL97]), it is a factor of two better than some embedded-processor-based implementations (such as the Sequent STiNG [LC96]).

4.3.3.1 Baseline System. Figure 4-5 compares the performance of Fixed and Floating with varying number of processors per node. The graphs plot application speedups for two-, three-, and four-processor nodes under Fixed and Floating over a uniprocessor node (Single). All systems include a total of 16 nodes. Except for *em3d*, adding a dedicated protocol processor to a uniprocessor node improves performance by at most 25%. *Em3d* is our most communication-intensive application with a compute-utilization of less than 25%. The application iterates over a bipartite graph, computing new values for each graph node. Fetching remote node values dominates the running time of an iteration. Eliminating interrupt overhead allows Fixed to improve performance by 41%.

Using the second processor for computation—under the Floating policy—improves performance by 35%-90% in all applications. Except for *em3d*, all the applications exhibit moderate to high compute-utilizations and can take advantage of the second computation processor. In *em3d*, the second processor both contributes to computation and alternates with the other processor to act as protocol processor. As such, Floating manages to main-

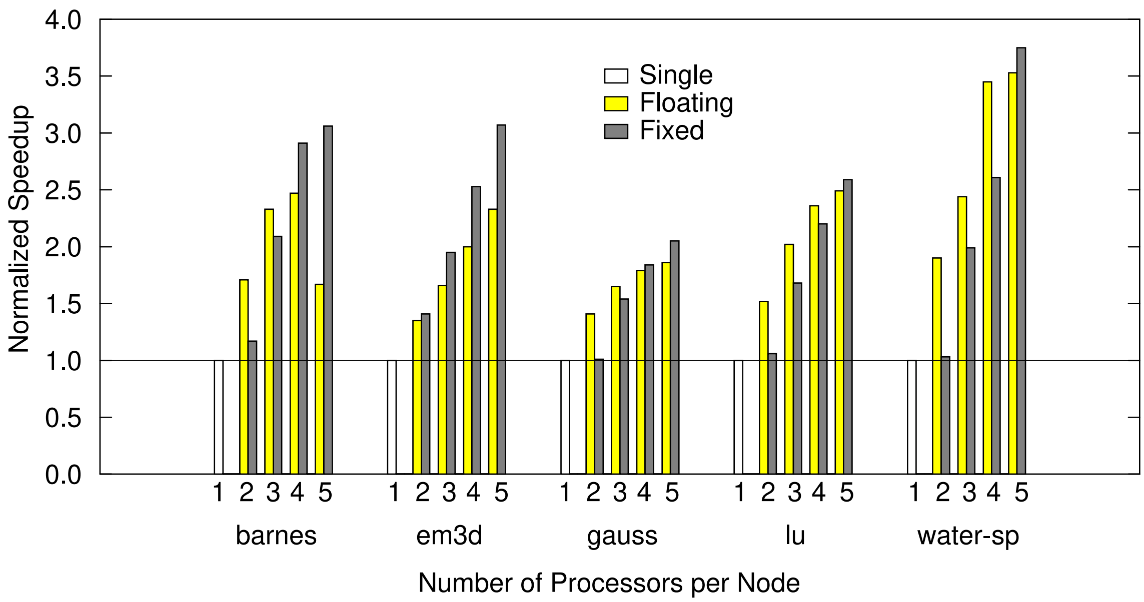


Figure 4-5. Baseline system performance comparison. The figure plots application speedups over Single (uniprocessor-node) under Fixed and Floating with increasing number of processors per node. The numbers appearing over the horizontal line at 1 indicate a better performance over Single.

tain performance to within 5% of Fixed. Therefore, as predicted by the microbenchmark experiment, at two processors per node Floating offers superior overall performance versus Fixed.

As we increase the number of processors per node, we increase both computational resources and demand for protocol processing. *Water-sp* is the most compute-bound application ($U > 0.9$) and primarily benefits from addition of compute processors. It takes five processors per node to increase protocol processor utilization high enough to allow Fixed to slightly (< 6%) improve performance over Floating.

Barnes is 50% compute-bound and exhibits moderate speedups with an increase in the number of processors per node. *Barnes* becomes communication-bound with four processors per node, at which point Fixed improves performance over Floating by 18%. Increasing the number of compute processors from four to five (under both policies) results in a performance drop due to high contention for locks. Under Fixed, *barnes* shows a speedup of three over Single with five processors per node.

Em3d with its low compute-utilization exhibits surprisingly high speedups. *Em3d*'s bipartite graph is initially constructed so that processors share a fraction of their edges with neighboring processors. An increase in the number of processors increases the likelihood that processors share graph data within a (SMP) node, reducing the aggregate number of remote accesses per node. Because *em3d* is primarily communication-bound, it heavily utilizes the protocol processor resulting in better performance under Fixed with only two (compute) processors. High protocol occupancy and protocol thread migration overhead further increases the performance gap between Fixed and Floating to 32% with five processors per node.

Gauss exhibits a more moderate level of compute-utilization (U = 0.75). Communication and computation in *gauss* proceed in synchronous phases. Communication consists of broadcasting a row of a matrix to all compute processors, and quickly becomes the bottleneck with an increase in the number of processors per node and prevents *gauss* from speeding up. Because communication does not overlap with computation, all compute processors remain idle for duration of the broadcast. As such, an idle processor remains as the acting protocol processor under Floating during the entire communication phase. By eliminating the protocol thread migration overhead, Floating mimics the behavior of Fixed and reduces the performance gap to 10% with five processors per node.

Although *Lu* is a compute-intensive application with a compute-utilization of 85%, it suffers from significant load imbalance [WOT+95] and exhibits only moderate speedup. Load imbalance in *lu* also increases the probability of a processor being idle on the nodes acting as protocol processor. The combined effect of high compute-utilization and load imbalance shifts the balance between the two policies in favor of Floating. Fixed marginally (< 5%) outperforms Floating with five processors per node.

In summary, the majority of the applications favor the Floating policy with two or three processors, and the Fixed policy with four or more processors per node. Large-scale SMP nodes with more than five processors per node may result in prohibitively high protocol processor utilization (unless the application is 100% compute-intensive). Single-threaded software protocols may not sustain enough bandwidth for machines with large-scale SMP nodes. Chapter 5 studies mechanisms for executing protocols in parallel.

4.3.3.2 Interrupt Overhead. Floating's (Single's) performance is sensitive to how quickly the system can interrupt a processor and dispatch a protocol handler. Today's commercial operating systems do not provide fast delivery of user-level interrupts. Exception handling on these systems can take up to 200 µs [TL94], one to two orders of magnitude longer than that on some carefully tuned parallel computers [RFW93]. This experiment studies the sensitivity of the policy trade-off to interrupt overheads.

Table 4.2 presents execution times of Single and two-processor Floating, normalized to two-processor Fixed for three values of interrupt overhead. The numbers appearing in bold are points where Fixed outperforms Floating. As predicted by the microbenchmark analysis, very high interrupt overheads severely impact Single's performance. Increasing interrupt overhead by two orders of magnitude can increase running time under Single by over 400%. This result corroborates the observation that with stock operating systems, networks of workstations (NOWs) [ACP95] may have to rely on program instrumentation [LS95,vECGS92] to perform periodic polling.

Another observation, consistent with the microbenchmark results, is that a very high interrupt overhead has a much smaller impact on Floating's performance than Single's. In all applications, a two orders-of-magnitude increase in interrupt overhead slows the program down by at most 45%. This is because an idle processor acting as protocol processor eliminates many of the interrupts. High interrupt overhead has the largest impact on *barnes*, because communication in *barnes* is primarily asynchronous and processors have a high probability of being busy computing when protocol messages arrive.

Table 4.2: Policy performance sensitivity to interrupt overhead.

Application	Interrupt Overhead (O_{int})					
	Single / 2-proc Fixed			2-proc Floating / 2-proc Fixed		
	0.5 µs	5 µs	50 µs	0.5 µs	5 µs	50 µs
barnes	1.17	1.45	3.25	0.69	0.76	**1.00**
em3d	1.41	1.85	5.44	**1.04**	**1.05**	**1.12**
gauss	1.01	1.04	1.57	0.72	0.75	0.75
lu	1.06	1.17	1.99	0.69	0.74	0.92
water-sp	1.03	1.10	1.59	0.54	0.56	0.70

4.3.3.3 Protocol Block Size. A key factor in the performance trade-off between the policies is protocol weight. Heavier-weight protocols reduce the impact of protocol processing overhead on performance (Section 4.3.2). By making overhead a small fraction of protocol occupancy, a higher-weight protocol shifts the performance trade-off towards Floating. Because much of protocol occupancy in fine-grain DSM is due to transferring data between memory and the network [Rei96], protocol weight is very sensitive to block size. Figure 4-6 compares the performance of Fixed and Floating for a 64-byte protocol (above) and a 256-byte protocol (below). The 64-byte and 256-byte protocol have round-trip miss times for a simple remote read of 1204 cycles (3.01 µs) and 2055 cycles (5.14 µs) respectively.

Much as in baseline system results, Floating's performance dominates for two-processor nodes even with the lower-weight 64-byte protocol. The 256-byte protocol increases Floating's performance relative to Fixed at a higher number of processors per node. *Em3d*'s performance under Floating significantly improves the performance break-even point between the two policies, increasing it from two to four processors per node. *Gauss* also exhibits a performance boost under Floating increasing performance to within 5% of

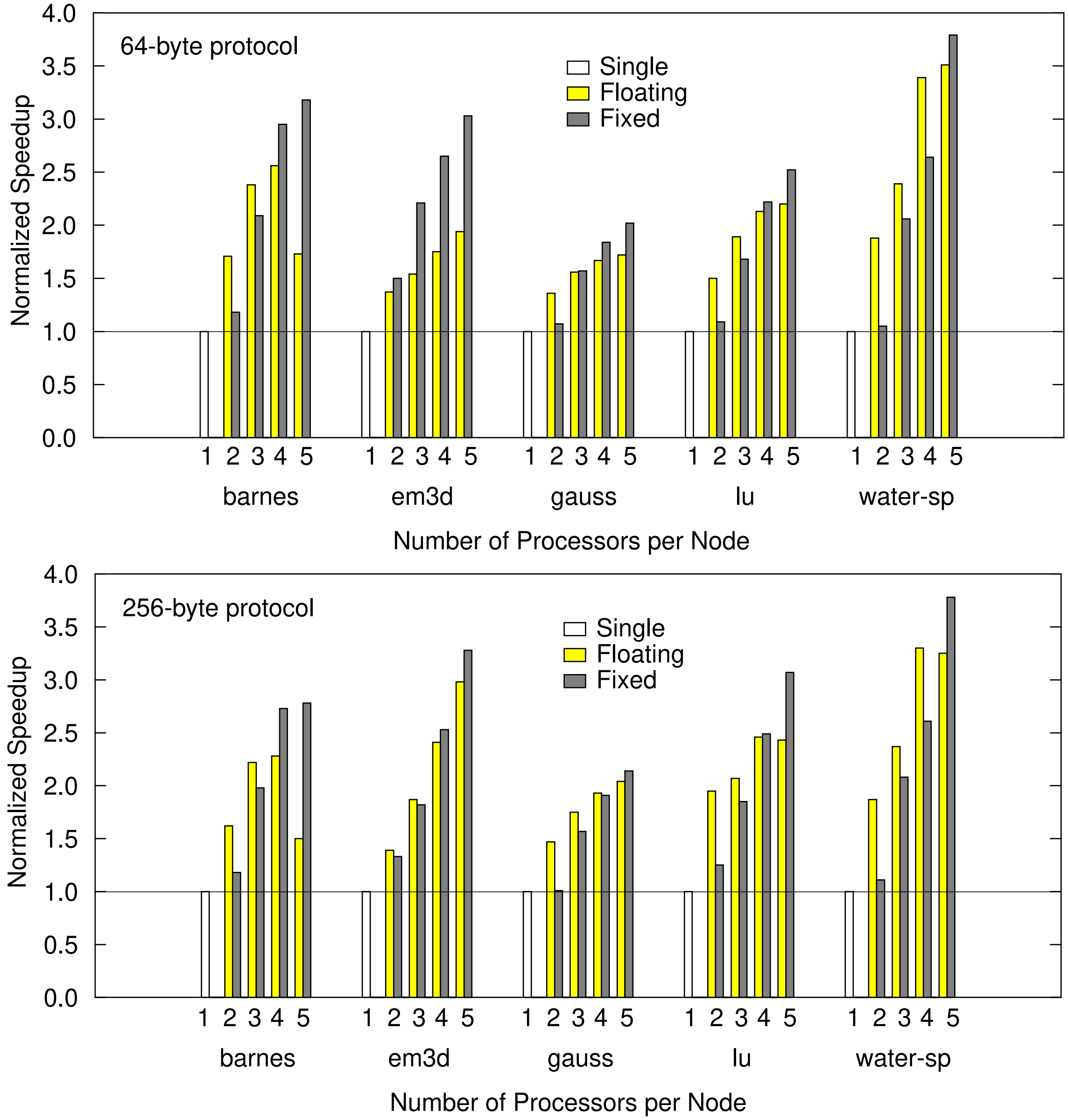


Figure 4-6. Performance sensitivity to protocol block size. The figure plots application speedups over Single under Fixed and Floating with increasing number of processors per node for a 64-byte protocol (above) and 256-byte protocol (below). The numbers appearing over the horizontal line at 1 indicate a better performance over Single.

Fixed at five processors per node. The change in protocol weight, however, is not high enough to shift the (average) performance break-even point between the two policies to more than four processors per node.

Surprisingly, a higher protocol weight increases the performance gap between the two policies at five processors per node for two of the applications. Both *lu* and *water-sp* experience false sharing due to the large block size. Performance under Floating drops relative to Fixed because false sharing increases the apparent communication intensity in these applications.

4.3.3.4 Cache Size. Under Fixed, a dedicated protocol processor is equipped with separate caches to store protocol code and data. In contrast, protocol references under Floating leave a footprint in the compute processor's cache, interfering with computation. Therefore, small compute processor caches can benefit from a dedicated protocol processor. Small caches, however, also may thrash if an application's primary working set does not fit in the cache [RSG93]. When processor caches thrash, communication becomes a small component of running time and a high-overhead protocol processing policy (e.g., Floating) becomes more competitive. The performance trade-off between the two policies depends on which of the above two factor has a dominant effect.

Figure 4-7 compares the performance of Fixed and Floating for 16-Kbyte (above) and 128-Kbyte (below) processor caches. Compared to the baseline system results with 1-Mbyte caches (Figure 4-5), Floating's performance increases relative to Fixed for both cache sizes in four of the applications. Consequently, the performance break-even point between the two policies increases from four to five processors per node. Smaller caches result in thrashing and thereby decreasing an application's apparent communication-to-computation ratio. A lower communication-to-computation ratio makes Floating more competitive.

Em3d with its large data set thrashes even with the 1-Mbyte processor caches. Therefore, smaller processor caches do not significantly affect the performance trade-off between the policies in *em3d*. Floating's performance slightly decreases relative to Fixed from 128-Kbyte to 16-Kbyte processor caches; protocol interference in the 16-Kbyte processor caches reduces Floating's performance. Much like *em3d*, *barnes* and *gauss* exhibit a slight

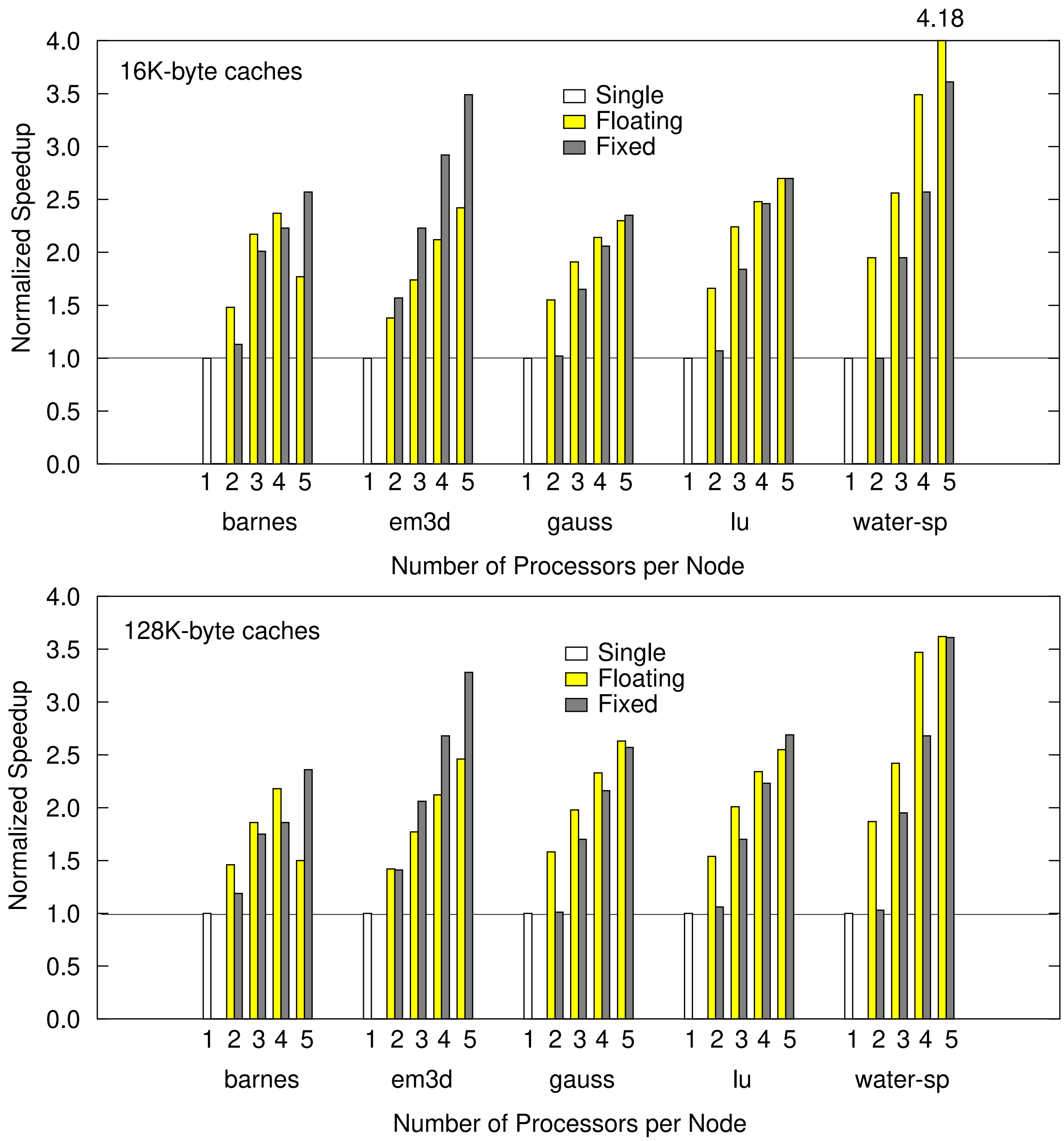


Figure 4-7. Performance sensitivity to processor cache size. The figure plots application speedups over Single under Fixed and Floating with increasing number of processors per node for 16-Kbyte processor caches (above) and 128-Kbyte processor caches (below). The numbers appearing over the horizontal line at 1 indicate a better performance over Single.

decrease in Floating's performance relative to Fixed when caches are scaled down due to protocol interference in processor caches.

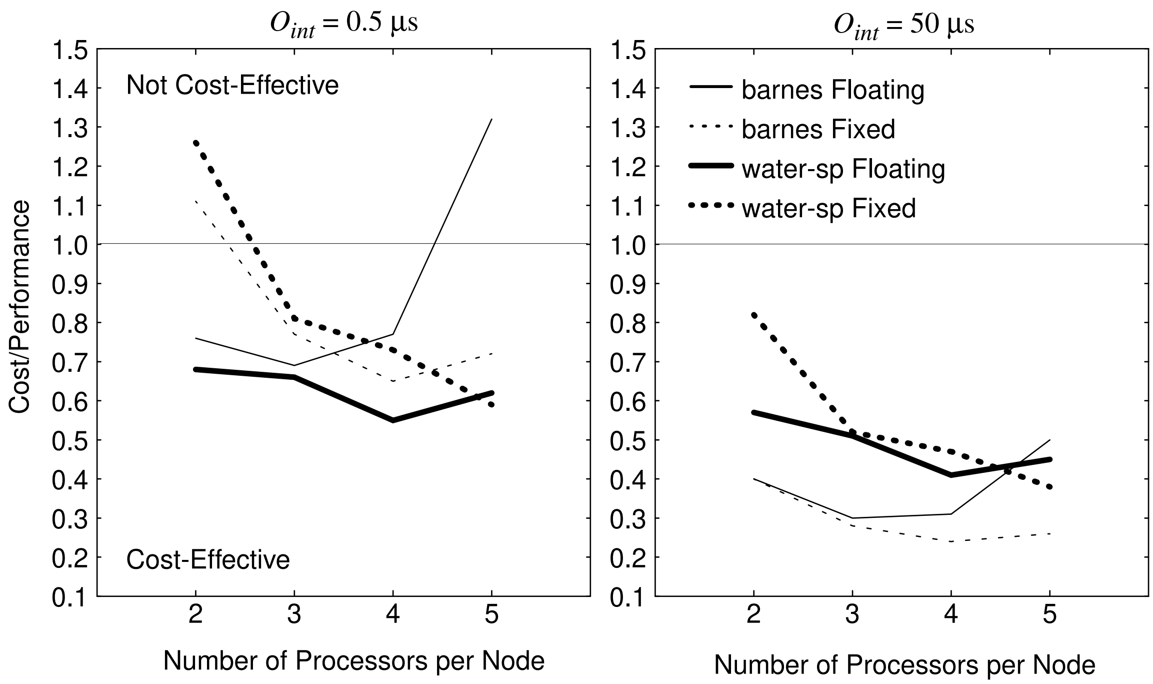


Figure 4-8. Relative cost-performance of Fixed and Floating. The figure plots cost-performance of Fixed and Floating for *barnes* and *water-sp* against varying the number of processors per node for two values of interrupt overhead (O_{int} = 0.5 µs and O_{int} = 50 µs). Costups and speedups are calculated with respect to a uniprocessor node (Single). The graphs assume that the cost of a processor is 30% of the cost of a uniprocessor node. Values over the horizontal line at 1 indicate design points that are not cost-effective.

4.3.3.5 Cost/Performance. Figure 4-8 plots cost-performance for two applications with moderate (*barnes*) to high (*water-sp*) compute-utilizations versus the number of processors. The graphs indicate that adding a dedicated protocol processor to a uniprocessor node is never cost-effective for the lower interrupt overhead (left). This is not surprising since performance improves by at most 17% whereas the system cost goes up by 30%. When overhead is high (right), performance in *barnes* improves by 45% justifying the cost of the dedicated protocol processor. Computation in *water-sp* remains the dominant factor in the running time. Even with higher interrupt overhead the program benefits little from a dedicated protocol processor. A second compute processor, however, improves performance in the two applications by at least 71% and is therefore cost-effective.

Much as the microbenchmarks predicted, when interrupt overhead is low—as compared to protocol weight—the system is most cost-effective under the Floating policy. For *barnes*, cost-performance under Fixed reaches a minimum close to, but not the same as, that under Floating. *Water-sp* speeds up linearly and therefore always reaches a lower cost-performance under Floating. When the number of processors is large enough (> six), speedup dominates cost-performance in *water-sp* causing it to eventually level off. At this point, Floating results in a marginal improvement in cost-performance over Fixed.

High interrupt overhead, however, changes the balance. *Barnes* achieves a minimum cost-performance under the Fixed policy. The high overhead increases protocol processor occupancy, resulting in a higher protocol processing to running time ratio. The Fixed policy reduces protocol processor occupancy, allowing the protocol processor to accommodate a larger number of processor before protocol processing saturates. At this point, the performance improvement due to a dedicated protocol processor is large enough to offset its incremental cost. Floating remains most cost-effective for the more compute-intensive application, *tomcatv*. High interrupt overhead, however, slightly closes the gap in cost-performance between to the two policies for this application.

4.4 Related Work

Several designs for parallel computers using SMP nodes dedicate one SMP processor on every node to run communication protocols. The Intel Paragon [Int93] and the proposed Message Proxies on a cluster of Power PC SMPs [LHPS97] implement a protected messaging boundary among SMP-node processors by dedicating a single processor on an SMP node to run software protocols in system mode. By eliminating high-overhead system calls, these designs allow for high-performance protected communication among SMP nodes. In contrast, my work focuses on SMP-node processors that collectively run a single application in user-level and where protection among multiple SMP-node processors is not required.

Karlsson, et al. and Erlichson, et al. have independently studied the trade-off of dedicated versus multiplexed protocol scheduling for VM-based software DSM protocols [KS96,ENCH96]. Both studies conclude a dedicated protocol processor will be of little use in systems with such heavy-weight protocols. In contrast, this study indicates that a dedicated protocol processor is advantageous for fine-grain DSM protocols when there are a large number (more than four) of processors per node, or when interrupt overheads are very high (as in commodity stock operating systems).

Previous studies did not consider cost in their analysis. Cost is important in the trade-off analysis of scheduling policies because given a large enough number of processors, a dedicated protocol processor eventually results in a superior performance but may not always result in a cost-effective design point (i.e., the cost of adding a dedicated protocol processor may not justify the performance gain). In addition to a performance analysis, this study also includes a cost-performance analysis of the two policies.

4.5 Summary

This chapter examined how a single-threaded protocol should be scheduled on an SMP-node parallel machine. The chapter presented results from synthetic benchmarks for two general request/reply protocols to illustrate the trade-offs between the policies. The results showed that:

- A dedicated protocol processor benefits light-weight protocols much more than heavy-weight protocols; overheads saved by a dedicated protocol processor represent a significant fraction of protocol occupancy in light-weight protocols.
- A dedicated protocol processor is generally advantageous when there are four or more processors per node.
- Interrupt overhead has a much higher impact on Single's performance than Floating's corroborating previous results [KS96]; protocol messages are likely to find an idle pro-

cessor on multiprocessor node acting as the protocol processor thereby eliminating interrupts.

- The system with the lowest cost-performance will include a dedicated protocol processor when interrupt overheads are much higher than protocol weight—as in lightweight protocols.

Finally, the chapter evaluated these policies in the context of a fine-grain user-level distributed shared-memory system. The chapter presented results from simulating a network of 16 SMP workstations—each with five processors—running five shared-memory applications using a software coherence protocol. Besides corroborating the findings from the first experiment, the results also showed that:

- Bursty and synchronous communication patterns in some applications reduce overhead and therefore decrease the benefit of the Fixed policy.
- Smaller processor caches make Floating more competitive. Smaller caches increase the local memory traffic on a node. A larger local memory traffic decreases the apparent communication-to-computation ratio (Section 3.4.3), shifting the balance towards Floating.

The results also indicated that—with the exception of highly compute-bound applications—single-threaded software protocol execution on SMP processors may become a communication bottleneck in clusters of large-scale SMPs (with five or more processors). These machines may require mechanisms for either faster protocol execution (e.g., an embedded protocol processor on the network interface card), or parallel protocol execution. Chapter 5 examines parallel software protocol execution as an approach to mitigate the single-thread protocol execution bottleneck.

Chapter 5

Executing Fine-Grain Protocols in Parallel

Chapter 4 evaluated scheduling policies for single-threaded protocol execution on SMP clusters. The results indicated that although a single protocol thread may be suitable for small-scale SMP nodes, communication may become a performance bottleneck with an increase in the number of processors per node. Applications requiring very fine communication granularities increase protocol activity in the system and exacerbate the software protocol bottleneck [CSBS95,BH86]. Parallel execution of software protocols can help reduce queueing delays by overlapping the execution of multiple protocol events.

In this chapter, I study parallel implementations of fine-grain DSM coherence protocols. Parallel dispatch queue (Chapter 2) provides a simple and efficient set of mechanisms for parallelizing fine-grain software protocols. I propose *PTempest*, a set of mechanisms for implementing parallel fine-grain DSM protocols. PTempest unifies the user-level fine-grain DSM mechanisms of Tempest [Rei95] with parallel protocol execution mechanisms of PDQ.

This chapter presents the design and evaluation of two parallel fine-grain DSM systems, *Hurricane* and *Hurricane-1,* that provide different levels of hardware support for PTempest. I compare the performance of the Hurricane systems to S-COMA [HSL94], an all-hardware implementation of fine-grain DSM. Much like Typhoon [RPW96], Hurricane integrates one or more embedded processors with fine-grain shared-memory support and the networking hardware (i.e., message queues) on a custom device. Similarly, Hurricane-1 is like Typhoon-1 [RPW96] and integrates shared-memory and messaging hardware on a custom device but relies on the node's commodity processors to run the software protocol.

The Hurricane devices differ from their Typhoon counterparts in that they use the PDQ mechanisms to support parallel protocol execution. To evaluate PDQ's potential to increase performance in a software fine-grain DSM, the Hurricane systems use a high-performance dynamically demultiplexed implementation of PDQ (Chapter 2). Dynamic demultiplexing evenly distributes the protocol execution load among the protocol processors by demultiplexing protocol events upon demand.

To facilitate the discussion in the rest of this chapter, I use the term *protocol processor* to refer to either S-COMA's finite-state-machine (FSM) hardware protocol implementation, an embedded processor on Hurricane, or a commodity SMP-node processor in Hurricane-1. Following this terminology, S-COMA is a single-processor device, and Hurricane and Hurricane-1 are either single-processor or multiprocessor devices.

In this chapter, I first describe PTempest, the mechanisms enabling the implementation of parallel fine-grain DSM coherence protocols, and PStache, a simple parallel coherence protocol. Section 5.2 describes the Hurricane family of devices implementing PTempest in hardware. Section 5.3 presents in detail a performance evaluation of the Hurricane devices. Section 5.5 presents a discussion of the related work. Finally, Section 5.5 concludes the chapter with a summary of results.

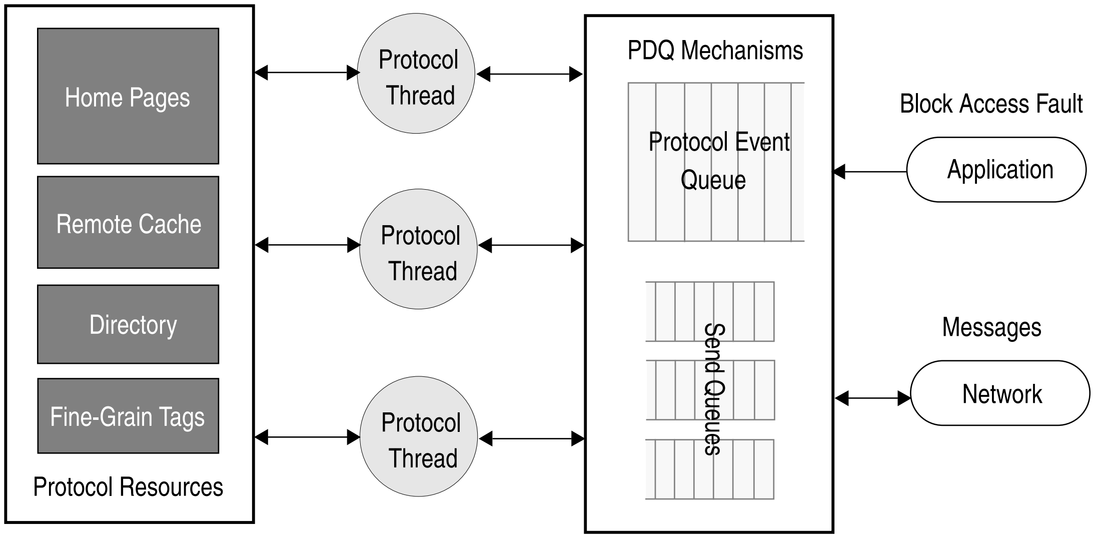


Figure 5-1. Accessing protocol resources in PTempest.

5.1 PTempest: Parallelizing Fine-Grain DSM Protocols

Parallel Tempest (*PTempest*) unifies the Tempest mechanisms for implementing user-level fine-grain DSM with the PDQ mechanisms for parallel protocol execution. Tempest's mechanisms enable a user-level protocol to map/unmap shared-memory pages in an application's address space. A user-level protocol also uses the mechanisms to manipulate shared-memory access semantics to fine-grain (e.g., 32-128 bytes) memory blocks on the mapped pages. Tempest also provides low-overhead fine-grain messaging to transfer memory blocks and invoke protocol action across machine nodes.

Tempest implements request/reply protocol actions in the form of a block access fault (i.e., request) or a message (i.e., reply/response) handlers. As in Active Messages [CDG+93], Tempest handlers are defined to a have a single-threaded execution semantics and must appear to execute atomically. To relax Tempest's single-threaded execution semantics, PTempest augments the active-message handler interface of Tempest with parallel handler dispatch mechanisms of PDQ.

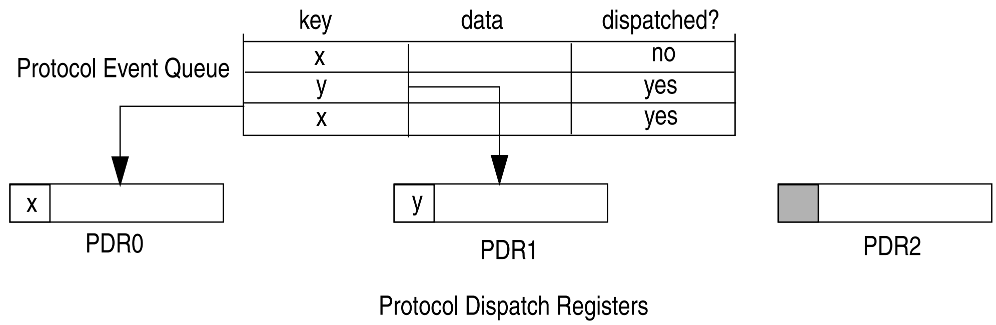


Figure 5-2. Parallel protocol dispatch in PTempest.

Figure 5-1 illustrates the protocol resources required in a typical PTempest implementation. Shared data pages are distributed among the nodes, with every node serving as a designated *home* for a group of pages. A *directory* maintains sharing status for all the memory blocks on the home nodes. A *remote cache* either on a custom network device or in main memory serves as a temporary repository for data fetched from remote nodes. A set of *fine-grain tags* enforce access semantics for shared memory blocks. A protocol event queue to gather block access faults (generated by the application) and incoming network messages. One or more message send queues (one per protocol thread) allow protocol threads to transfer memory blocks and invoke protocol action on remote nodes.

To guarantee mutually exclusive accesses to protocol resources from multiple protocol threads, PTempest requires protocols to partition the resources into object groups and label protocol events with the appropriate group id. In fine-grain DSM, a protocol event is typically associated with a fine-grain memory block (e.g., a block access fault or an invalidation message). Therefore, protocol resources can be partitioned so that accessing resources (e.g., fine-grain tags) corresponding to distinct memory blocks are mutually exclusive. By labelling a protocol event with a block's shared-memory (virtual) global address, a protocol allows PTempest to dispatch the event to the appropriate protocol thread.

Figure 5-2 depicts parallel protocol dispatch in a dynamically demultiplexed PTempest implementation (Section 2.3.3). A protocol event queue holds block access faults and incoming messages. Each entry in the queue is labeled with an object partition id which typically corresponds to a memory block's global address. PTempest dispatches events to protocol processors based on an event id much like a packet classifier in conventional network communication software/hardware [BGP+94]. The queue entries also maintain information as to whether a specific entry has been dispatched. A protocol thread receives an event through a per-thread protocol dispatch register (PDR). Upon event dispatch, the system searches for the first available entry with an id different from the currently dispatched ids in the PDRs. A protocol thread signals the completion of a handler execution by clearing the corresponding PDR.

PTempest also provides two other types of handler execution semantics using distinct id values corresponding to invalid shared-memory global addresses: single-thread execution semantics and parallel execution semantics without dispatch synchronization. The first allows executing the less-frequent protocol handlers such as virtual memory management code (e.g., allocating a page in the shared global address space) to execute mutually exclusively with other handlers. A PTempest implementation dispatches such a handler after all protocol events in front of it in the event queue have dispatched and completed. Furthermore, no further handlers are dispatched until such a handler complete execution. The second allows handlers such as those accessing read-only data structures or performing remote stores to always dispatch immediately without synchronization with other handlers.

Transparent shared-memory protocols running on Tempest can simply be parallelized using PTempest. This study uses *Parallel Stache* (*PStache*), a parallelized version of Stache [RLW94] (discussed in Chapter 4), an S-COMA-like invalidation-based full-bit-map coherence protocol which replicates remote data in the main memory on every machine node.

5.2 Hurricane: Hardware Support for PTempest

Hurricane is a family of custom devices that provide hardware support for PTempest. Hurricane devices are closely derived from the Typhoon family of custom devices [Rei96] and integrate shared-memory access control logic and networking hardware. The key difference between the Typhoon and Hurricane devices is that the Hurricane devices implement the PDQ mechanisms to collect and dispatch block access faults and incoming messages. This study evaluates two Hurricane designs: a fully-integrated high-performance *Hurricane* device which contains embedded processors to run the coherence protocol, and a less-integrated more cost-effective *Hurricane-1* device which uses a node's commodity processors to run the protocol. The section describes the hardware details of Hurricane and Hurricane-1.

5.2.1 Hurricane

Hurricane integrates the fine-grain access control logic, messaging queues, a number of embedded protocol processors, and the PDQ mechanisms for protocol event dispatch all on a single device. Figure 5-3 illustrates the architecture of a Hurricane custom device. A protocol event queue collects all block access faults (generated on the node) and incoming messages (from remote nodes). A protocol thread running on a protocol processor removes entries from a protocol event queue by accessing its PDR. Much as in Typhoon, the PDRs reside on the cache bus and can be loaded in a single cycle [Rei96].

PTempest protocols use shared-memory global addresses to name memory blocks. The access control logic, however, snoops on physical addresses on the memory bus to detect block access faults. PTempest uses a reverse translation table to map physical addresses from the memory bus to shared global addresses. The table also maintains the fine-grain tags associated with a memory block to enforce shared-memory access semantics. Hurricane implements a Typhoon-like reverse translation look-aside buffer (RTLB) in SRAM which caches the recently accessed entries in the reverse translation table. Upon a block

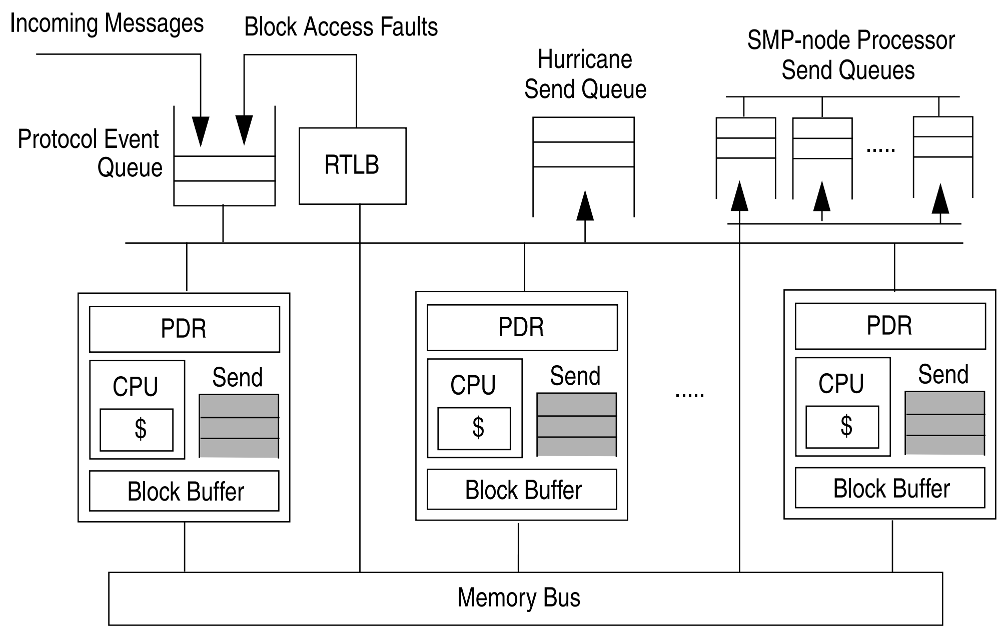


Figure 5-3. The Hurricane custom device. The device contains two or more protocol processors, each with a protocol dispatch register, a Typhoon-like block buffer, and a message send queue. The processors share an RTLB which holds the fine-grain access control tags for cached remote data. There is also a message send queue for every compute processor on the node, and a parallel protocol dispatch queue recording block access faults and incoming messages on the node

access fault, the contents of the tag are read and placed in a protocol event queue entry. Fine-grain tags are updated by writing to the RTLB.

Like other software fine-grain DSMs [K^+94,RLW94], Hurricane maintains the directory in main memory and uses protocol processor caches to access the directory entries. Many studies on fine-grain DSM have concluded that directory entries exhibit locality which can be exploited by small hardware caches [HKO^+94,SH91]. Such an approach obviates the need for on-board SRAM-based implementations which are both more expensive and limit scalability.

Protocol processors are each equipped with a block buffer which is a small direct-mapped cache that serves as an intermediary storage device for moving memory blocks between the node's memory hierarchy (e.g., processor caches or main memory) and the message queues. Removing/placing remote data from/into a node also typically involves modifying the state of the block maintained by the fine-grain tags. Shared-memory access semantics dictates that moving data and updating the corresponding tag value should appear to execute atomically. The block buffer implements this atomic operation directly in hardware [Rei96].

To obviate the need for synchronization upon sending messages, every protocol processor is equipped with a message send queue. As in Tempest, PTempest supports user-level messaging directly from the node's (computation) processors, and therefore Hurricane also provides a separate send queue for each compute processor.

Hurricane provides hardware for suspending/resuming computation on a compute processor after a block access fault. Upon a block access fault, Hurricane's hardware masks the faulting processor from bus arbitration and returns a request-and-retry as an acknowledgment to the faulting bus transaction. Upon receiving the memory block and resuming the compute processor, Hurricane simply unmasks bus arbitration for the resuming compute processor.

5.2.2 Hurricane-1

Hurricane-1 (Figure 5-4) combines the fine-grain access control logic with the messaging queues on a single device but uses the SMP-node commodity processors for running the software protocol. As in Hurricane, a single (dynamically demultiplexed) PDQ gathers information about all block access faults generated on the node and all of the incoming messages. Hurricane-1 provides a set of PDR and block buffer pairs implementing parallel protocol dispatch and execution. The number of PDR and block buffer pairs restricts the number of protocol threads scheduled to run on every SMP node. To allow synchronization-free user-level messaging from both computation and protocol processors, every SMP-processor also has a separate message send queue.

In Hurricane-1, each PDR is a cachable control register [RPW96]. Cachable control registers behave like memory and can be cached in processor caches. Polling on a cachable

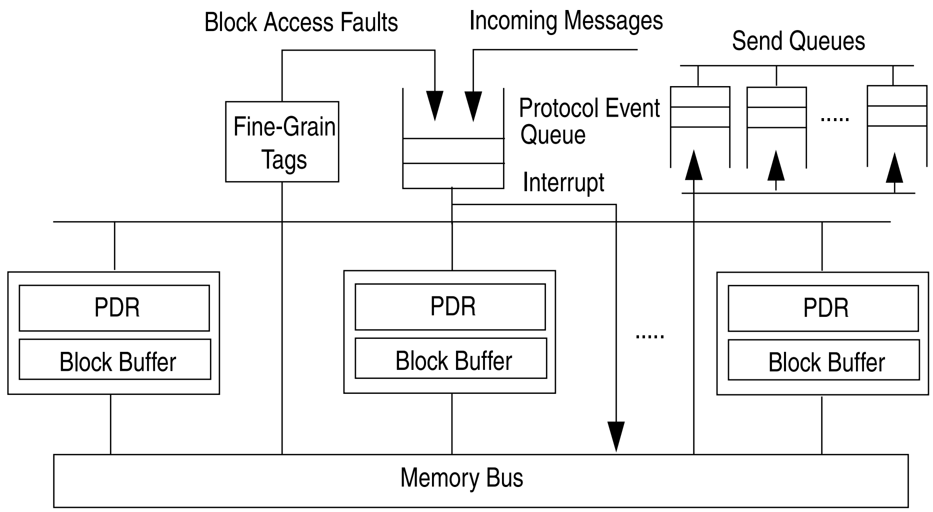


Figure 5-4. The Hurricane-1 custom device. The device contains two or more pairs of protocol dispatch registers and Typhoon-like block buffers, and a message send queue per processor on the node. There is a single SRAM device maintaining the fine-grain tags for cached remote data, and a parallel protocol dispatch queue recording block access faults and incoming messages on the node.

control register results in a cache hit in the absence of protocol events thereby eliminating polling traffic over the memory bus. Upon dispatching a protocol event, Hurricane-1 invalidates the cached copy of the PDR forcing a protocol processor to read the new contents of the PDR.

Much like Hurricane, Hurricane-1 implements the directory and a reverse translation table for mapping physical addresses to shared global addresses in main memory. Fine-grain tags, however, are directly stored in SRAM on the custom Hurricane-1 device. Because the tags only store two bits per memory block, the memory overhead is low for practical DSM implementations [Rei96]. Such an approach limits the size of shared physical memory per node but allows for a cost-effective implementation by not requiring a hardware cache.

To further simplify hardware complexity and reduce cost, Hurricane-1 also does not support masking/unmasking bus arbitration on the memory bus. As such, upon a block access fault, the faulting processor takes a bus error and spins on a per-processor software flag in memory inside the trap routine. The protocol software is responsible for resuming the compute processor by signalling the corresponding flag.

A Hurricane-1 device supports both dedicated and multiplexed scheduling of protocols on SMP processors. To allow multiplexed scheduling of a protocol thread upon a message arrival, Hurricane-1 provides mechanisms for invoking interrupts on the memory bus. A memory-mapped interrupt arbiter device located on the memory bus distributes interrupts among the processors in a round-robin fashion. Hurricane-1 uses a software interrupt masking/unmasking scheme similar to the one in Typhoon-1 described in Section 4.1.

5.3 Performance Evaluation

The system performance using the Hurricane devices depends on the memory system performance and the available parallelism in protocol event execution. Parallel protocol execution may increase the number of outstanding memory requests which in turn requires a higher bandwidth memory system (e.g., split-transaction bus, highly-interleaved memory, etc.). Available parallelism also depends on whether there are independent outstanding protocol events. Parallel protocol execution may make software protocols competitive to an all-hardware protocol implementation if the performance gain due to parallelism offsets the effect of higher protocol occupancy of software protocols.

This study compares the performance of DSM systems using Hurricane devices to a simple all-hardware protocol implementation, Simple-COMA (S-COMA) [HSL94,RPW96]. S-COMA is an invalidation-based full-map directory protocol much like Stache. The simulation model for S-COMA assumes minimum protocol occupancies accounting for only SRAM and DRAM memory access times. As such, S-COMA's performance numbers in this study are optimistic, making the comparison to S-COMA conservative.

In this section, I first present the methodology I use to carry out the experiments. Then, I analyze the minimum protocol occupancy in Hurricane and Hurricane-1, and compare them to S-COMA. Next, in a microbenchmark experiment, I measure the maximum bandwidth of data out of a node using the various custom devices. Finally, I use shared-memory applications to evaluate to what extent parallel software protocol execution improves application performance.

5.3.1 Methodology

I use the simulation methodology and system parameters described in Section 1.3.1 to simulate SMP clusters interconnected with Hurricane, Hurricane-1, and S-COMA network interfaces. The simulator assumes 1-Mbyte data caches for the Hurricane processors and 2-Kbyte block buffers for all the systems. This study models a infinitely-interleaved memory system to mimic the behavior of high-performance memory systems characteristics of large-scale SMP servers. The simulator assumes an interrupt overhead and a bus error trap overhead of 200 cycles in a Hurricane-1 system, characteristic of carefully tuned parallel computers [RFW93].

5.3.2 Protocol Occupancy

Communication in parallel applications running on a DSM cluster is either latency-bound or bandwidth-bound. Latency-bound applications are those in which protocol events experience little queueing delays at the network interface and communication performance is determined by minimum round-trip time of a remote miss. Bandwidth-bound applications, however, are those in which protocol events are generated in bursts and can lead to large queueing delays at the network interface.

Latency-bound applications primarily benefit from low-occupancy protocol implementations (such as hardware DSM) because lower protocol occupancy directly impacts round-trip miss times and thereby communication time. Bandwidth-bound applications, however, may eventually saturate a single protocol processor even in a low-occupancy protocol implementation due to a large number of outstanding protocol events which lead to queueing. Such applications may benefit from parallel network interfaces instead.

5.3.3 Microbenchmark Experiment

To compare the latency and bandwidth characteristics of the Hurricane systems to those of S-COMA, I use a simple remote read microbenchmark consisting of a tight loop. In every loop iteration, a processor simply reads a memory block belonging to a remote node but not available in the processor's cache/memory, i.e. a remote block in the invalid state. The benchmark measures the average time it takes for a loop iteration to complete.

Table 5.1 depicts the breakdown of time for various system events on a simple remote read of a 64-byte block. The table groups the system events into three categories. A request category on the caching node accounts for all the events from the detection of a block access fault to sending a request message to the home node. A reply category on the home node consists of all events from the detection and dispatch of the request message to sending the 64-byte block to the caching node. A response category on the caching node accounts for all events from the detection and dispatch of the reply message to resuming the computation.

S-COMA maintains the fine-grain tags and the directory state in SRAM. Similarly, Hurricane caches the fine-tags in the RTLB and accesses the directory state through the protocol processor caches. In the common case of cache hits, S-COMA and Hurricane both incur the same exact memory access times for updating the state and data for a memory block. Hurricane additionally uses software to dispatch message handlers and invoke the SRAM and DRAM accesses. This additional software overhead substantially increases the minimum request/response occupancy by 315%. The minimum reply occupancy and the total round-trip latency increase only by 41% and 33% respectively.

Similarly, in the common case accesses to the reverse translation table and the directory state (both stored in main memory) in Hurricane-1 hit in the protocol processor caches. A protocol processor, however, must traverse the memory bus to initiate a fine-tag change and access the block buffer. Resuming the compute processor also requires writing to a memory flag on which the compute processor spins. Both the write to the flag by the pro-

Table 5.1: Remote read miss latency breakdown for a 64-byte protocol.

		Network Interface (latencies in 400 MHz cycles)		
	Action	S-COMA	Hurricane	Hurricane-1
Cacher	detect cache miss, issue bus transaction	5	5	5
Request Occupancy	detect block access fault, reverse translation, dispatch handler	12	16	87
	get fault state, send message	0	36	141
	network latency	100	100	100
Reply Occupancy (Home Node)	dispatch message handler	1	3	51
	reverse translation, directory lookup	8	61	121
	fetch data, change tag, send message	136	140	205
	network latency	100	100	100
Response Occupancy	dispatch message handler	1	4	50
	reverse translation	8	16	34
	place data, change tag	0	34	29
	resume, reissue bus transaction	6	6	178
Cacher	fetch data, complete load	63	63	63
	Total	440	584	1164

tocol processor and the subsequent read by the compute processor incur cache misses. These high overhead operations increase the request/response occupancy, reply occupancy, and total round-trip latency in Hurricane-1 by 518%, 160%, and 165% as compared to S-COMA respectively.

The round-trip latencies in Table 5.1 indicate that for latency-bound applications, i.e., applications in which communication is not bursty and protocol events do not experience long queueing delays at the protocol processors, S-COMA offers a moderate advantage over Hurricane, but may significantly improve performance over Hurricane-1. Bandwidth-bound applications, however, may benefit from multiple protocol processors simultaneously processing independent protocol events.

I use two microbenchmarks to compare bandwidth characteristics of S-COMA with those of Hurricane and Hurricane-1. A reply-bandwidth benchmark measures the reply bandwidth out of a single node in the machine by running a tight loop on multiple (request) nodes generating read misses to distinct remote memory blocks satisfied by a single (reply) node. A request-bandwidth benchmark measures the request bandwidth out of a single node by running a tight loop on multiple processors of a (request) node generating read misses to distinct remote memory blocks satisfied by multiple (reply) nodes.

Figure 5-5 compares the protocol event bandwidth in S-COMA, Hurricane, and Hurricane-1 for a 64-byte protocol. Figure 5-5 (above left) compares the reply bandwidth against the number of requesters in S-COMA and Hurricane with up to four protocol processors. The graphs indicate that the reply bandwidth immediately saturates for systems with a single protocol processor with an increase in the number of requesters. The saturation bandwidth varies among the systems based on the reply occupancy (Table 5.1). Not surprisingly, S-COMA saturates highest at a bandwidth of 89 MBytes/sec due its low reply occupancy, improving the peak bandwidth by 51% over a single-processor Hurricane system.

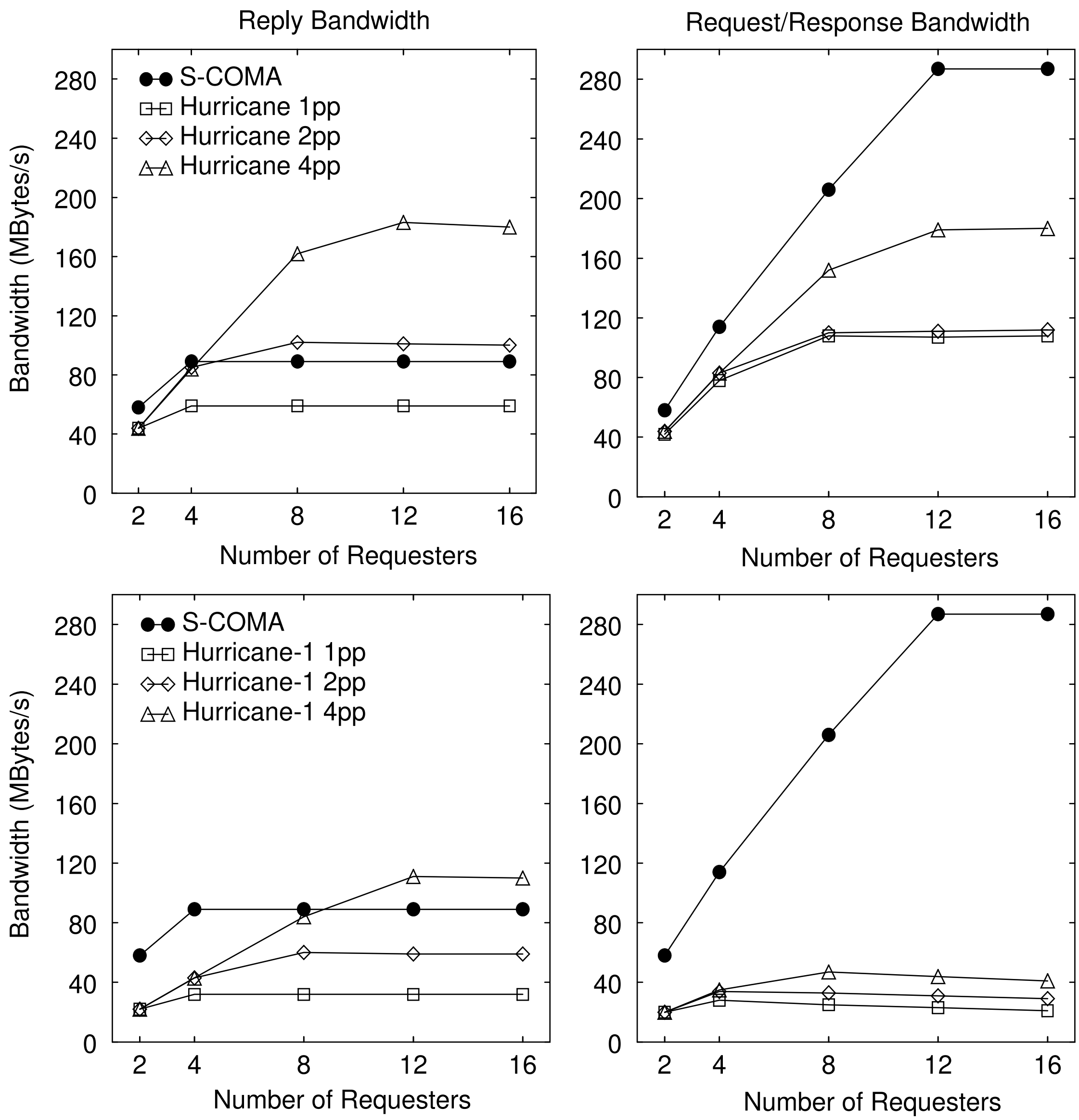


Figure 5-5. Protocol bandwidth in S-COMA, Hurricane, and Hurricane-1. The four figures compare S-COMA's reply (left) and request/response bandwidth (right) bandwidth with those in Hurricane (above) and Hurricane-1 (below) for a 64-byte protocol.

The graphs also indicate that increasing the number of protocol processors results in a significant increase the saturation bandwidth. The saturation bandwidth, however, does not increase linearly with the number of protocol processors. Using four protocol processors instead of one increases the saturation bandwidth only by a factor of three times. Because protocol state—such as directory entries—migrates between multiple protocol processor caches, the reply occupancy in multiprocessor Hurricane devices increases with respect to the single-processor devices. An increase in the reply occupancy prevents the saturation bandwidth from increasing linearly. Protocol state migration may be alleviated by using static (rather than dynamic) demultiplexing PDQ mechanisms in the Hurricane devices.

Parallel execution also allows the software protocol's reply bandwidth to become competitive with that of S-COMA. With four requesters, the reply bandwidth in two-processor and four-processor Hurricane systems reach that in S-COMA. When there are eight or more requesters, two-processor and four-processor Hurricane systems improve the reply bandwidth over S-COMA by 73% and 175% respectively.

Figure 5-5 (above right) compares the request/response bandwidth against the number of requesters in S-COMA and Hurricane. The request/response bandwidth in S-COMA reaches a much higher saturation point than the reply bandwidth because the combined request and response occupancies are much lower than the reply occupancy. The response occupancy remains low with an increase in the number of requesters even though the block buffers generate a considerable amount of writeback traffic in this microbenchmark; the block buffers are regularly flushed to memory to make room for newly arriving remote blocks. This study assumes large (2-Kbyte) block buffers and a highly-interleaved memory system. In systems with either smaller block buffers or lower memory interleaving, block buffer writeback traffic may limit the response or reply bandwidths.

The single-processor Hurricane system also reaches a higher maximum request/response bandwidth than the reply bandwidth. Much like the reply occupancy, protocol state migration among multiple protocol processor caches increases both the request and the response occupancies. Because the request and response occupancies, however, are rather small, incurring a single cache miss ($\approx$ 60 cycles) nearly doubles the occupancies. As a result, a second protocol processor in Hurricane only slightly improves the request/response bandwidth. Four-processor Hurricane systems help improve the request/response saturation bandwidth but fail to reach the peak bandwidth in S-COMA.

Figure 5-5 (below) compares the reply bandwidth (left) and the request/response bandwidth (right) in S-COMA and Hurricane-1 with up to four protocol processors. Multiple protocol processors increase the reply saturation bandwidth almost linearly; protocol state migration overhead accounts for a negligible fraction of the reply occupancy. Multiprocessor Hurricane-1 systems can also improve the maximum reply bandwidth over S-COMA. Because the saturation bandwidth in a single-processor Hurricane-1, however, is much lower than that in S-COMA, it takes four protocol processors in Hurricane-1 and twelve or more requesters to improve the saturation bandwidth by 25% over S-COMA.

Unlike a Hurricane device, the single-processor Hurricane-1 device reaches a lower maximum request/response bandwidth than reply bandwidth because of the high combined request/response occupancies. The saturation bandwidth also slightly decreases with an increase in the number of requesters because multiple requesting processors increase the memory bus traffic and result in queueing on the bus.

Much like Hurricane, protocol state migration in a two-processor Hurricane-1 system almost doubles the request/response occupancies relative to a single-processor. Such a large occupancy offsets the gains from parallel protocol processors resulting in a very low overall saturation bandwidth even with four protocol processors.

5.3.3.1 Protocol Block Size. Protocol block size impacts the relative performance of hardware and software protocol implementations. Larger blocks require a longer transfer time between the network and memory, increasing the fraction of protocol occupancy due to memory data transfer. Longer data transfer times, therefore, reduce the fraction of protocol occupancy due to software overhead and decrease the performance gap between hardware and software implementations.

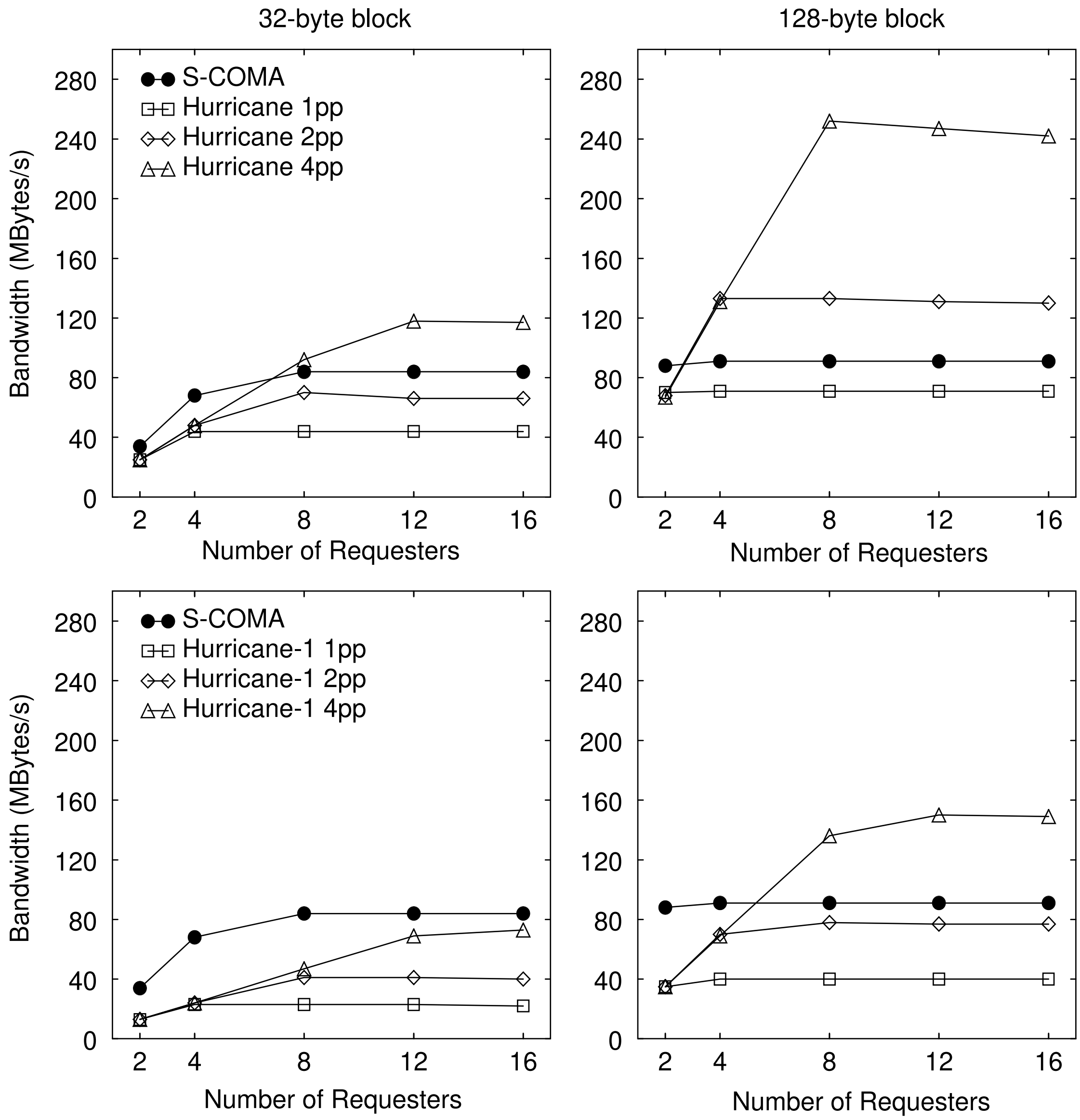


Figure 5-6. Impact of block size on protocol reply bandwidth. The figures plot reply bandwidth for a 32-byte (left) and a 128-byte (right) protocol for Hurricane (above) and Hurricane-1 (below).

Figure 5-6 illustrates the impact of changing the block size from 32 bytes (left) to 128 bytes (right) on the systems' reply bandwidth. Not surprisingly, the smaller block size makes the Hurricane devices less competitive relative to S-COMA. A two-processor Hurricane system (Figure 5-6 top left) no longer improves the maximum reply bandwidth over S-COMA. It takes a four-processor Hurricane system with eight or more requesters to slightly improve the reply bandwidth over S-COMA. A 32-byte block also reduces the overall peak bandwidth even with a four-processor Hurricane device to less than 120 MBytes/sec. Similarly, a 32-byte block prevents a Hurricane-1 system from reaching a maximum reply bandwidth greater than that in S-COMA even with four protocol processors.

Figure 5-6 (right) compares the request/response bandwidth in the Hurricane devices and S-COMA for a 128-byte block. A larger block reduces the performance gap between single-processor Hurricane devices and S-COMA. A larger block also reduces the impact of protocol state migration in multiprocessor devices because the larger data transfer time between the network and memory accounts for a larger fraction of the protocol occupancy. Therefore, larger blocks increase the performance improvement due to multiple protocol processors.

Block size impacts the request/response bandwidth (not shown) much like the reply bandwidth. A larger block size makes the parallel Hurricane devices more competitive with S-COMA. A 128-byte block allows a four-processor Hurricane system to increase the request/response bandwidth to 328 MBytes/sec improving S-COMA's peak bandwidth by 17%.

5.3.3.2 Protocol Processor Performance. Hurricane's performance is also sensitive to assumptions about the embedded processors' technology and clock speed. To reduce design time, some network interface devices with an embedded processor (e.g., Typhoon) use a previous-generation commodity processor consisting of a general-purpose integer core, TLB, and caches with the message queues and fine-grain DSM hardware support. This study models the technology gap between previous and current generation processors using different clock speeds.

Figure 5-7 plots the protocol event (reply and request/response) bandwidth in Hurricane with current-generation (400-MHz) processors and previous-generation (200-MHz) pro-

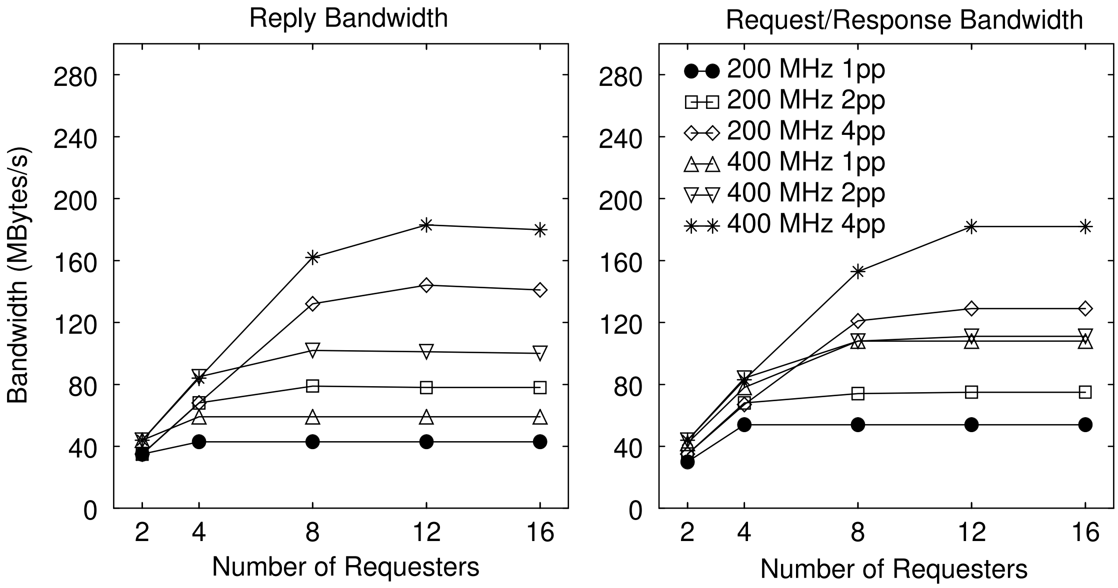


Figure 5-7. Impact of processor clock rate on Hurricane's protocol bandwidth. The two figures plot the reply bandwidth (left) and the request/response bandwidth (right) for one-, two-, and four-processor Hurricane devices for current generation (400MHz) and previous generation (200MHz) processor clock speeds.

cessors. The graphs indicate that the number of processors has a much greater impact on saturation bandwidth than processor clock rate. Whereas doubling the number of processors increases peak bandwidth by 81%, doubling the processor clock rate increases peak bandwidth by only 37%.

Because much of the protocol event occupancy is due to data transfer time between memory and the network, the system benefits much more from multiple outstanding memory requests than faster protocol code execution. This result suggests that hardware support for hiding memory latency—such as speculatively fetching memory blocks [HKO+94]—or increasing the number of outstanding memory request—such as non-blocking memory fetch—can help single-processor devices reach the performance of multiprocessor devices.

Table 5.2: Applications and input sets.

Benchmark	Description	Input Set
barnes	Barnes-Hut N-body simulation	16K particles
cholesky	Sparse Cholesky Factorization	tk29.O
em3d	3-D electromagnetic wave propagation	76K nodes, degree 5, 15% remote, distance of 16, 10 iters
fft	Complex 1-D radix-$\sqrt{n}$ six-step FFT	1M points
fmm	Fast Multipole N-body simulation	16K particles
radix	Integer radix sort	4M integers, radix 1024
water-sp	Spatial water molecule force simulation	4096 molecules

Unlike the reply bandwidth, the request/response bandwidth exhibits a higher sensitivity to protocol processor clock rate (Figure 5-7 right). Doubling the processor clock rate increases the request/response saturation bandwidth in a single-processor device by 100%. In contrast, doubling the number of processors increases the peak bandwidth by only 39%. Because requests and responses (in a single-processor device) do not involve memory data transfer, much of their occupancies is due to protocol instruction execution. Thus, a higher clock rate significantly impacts the request/response occupancies. A multiprocessor device, however, incurs protocol state migration overhead which offsets the benefits due to parallelism. With four slow processors, memory parallelism is high enough to offset the negative impact of protocol state migration improving the peak request/response bandwidth over a two-processor device with faster processors.

5.3.4 Macrobenchmark Experiment

The microbenchmark experiment in the preceding section helped analyze the latency and bandwidth characteristics of S-COMA and the Hurricane systems for simple remote read misses. Real applications, however, exhibit more complex interactions between the memory system and the protocol. Remote misses, for instance, can result in messages

Table 5.3: Application uniprocessor execution time and speedups.

Benchmark	Execution Time (Million Cycles)	S-COMA Speedup (Cluster of 8 8-way SMPs)
barnes	2,284	31
cholesky	976	5
em3d	740	34
fft	2,489	19
fmm	3,351	31
radix	903	12
water-sp	3,965	61

among three nodes in a producer/consumer relationship if neither the producer nor the consumer are the home node for the data. Real data sets also typically do not fit in caches and produce additional memory traffic on the bus. In this section, I re-evaluate the performance of the network interfaces using real applications.

Table 5.2 presents the applications I use in this study and the corresponding input parameters. *Barnes*, *cholesky*, *fft*, *fmm*, *radix* and *water-sp* are all from the SPLASH-2 [WOT+95] benchmark suite. *Em3d* is a shared-memory implementation of the Split-C benchmark [CDG+93].

Table 5.3 presents the applications' execution times (in cycles) on a uniprocessor and the corresponding speedup on an S-COMA interconnected cluster of 8 8-way SMPs implementing a 64-byte protocol. *Water-sp* is primarily computation-intensive and achieves near-linear speedups because communication overhead does not impact its performance. *Cholesky* is primarily communication-bound, suffers from a severe load imbalance [WOT+95], and does not speed up much. *Barnes*, *fmm*, and *em3d* have moderate communication-to-computation ratios and achieve a 50% efficiency with 64 processors. *Fft* and *radix* are communication-bound and exhibit poor speedups.

This section presents a performance comparison of S-COMA, Hurricane, and Hurricane-1 systems using the macrobenchmarks. Because Hurricane-1 uses SMP-node processors to execute the software protocol, the system can employ a variety of protocol scheduling policies (Chapter 3). This study evaluates two protocol scheduling policies on SMP processors. The base system, *Hurricane-1,* uses extra dedicated SMP processors and statically schedules a protocol thread to execute on every dedicated protocol processor.

The second system, *Hurricane-1 Mult,* uses a multiplexed scheduling policy. Much like the single-threaded Floating policy in Chapter 4, Hurricane-1 Mult obviates the need for extra dedicated protocol processors by using only idle SMP-node compute processors to assume the role of protocol processors. Under this policy, when a processor becomes idle (due to waiting for synchronization or a remote miss) it schedules a protocol thread. For example, in a system with 8-way SMPs, there may be up to eight protocol threads scheduled to execute depending on the number of idle compute processors. When there are no idle compute processors, the system invokes an interrupt and forces a processor to handle the protocol event. To eliminate extra scheduling overhead, the system at most interrupts the computation on one processor. Such a policy assumes that interrupts are infrequent and are only invoked to prevent long protocol invocation delays.

5.3.4.1 Base System Performance. Figure 5-8 depicts a performance comparison of the base case systems. The figure depicts application speedups normalized to S-COMA, for Hurricane systems with upto four embedded processors and Hurricane-1 systems with upto four extra (dedicated) SMP processors executing the protocol. The figure indicates that S-COMA improves performance over a software protocol running on an embedded processor (Hurricane 1pp) on the average by 32%. The figure also indicates that S-COMA significantly improves performance (by up to 89%) over a software protocol implementation running on a commodity SMP processor (Hurricane-1 1pp). These results are consistent with those of Reinhardt et al. comparing Typhoon and Typhoon-1 (which are similar to single-processor Hurricane and Hurricane-1) against S-COMA [RPW96].

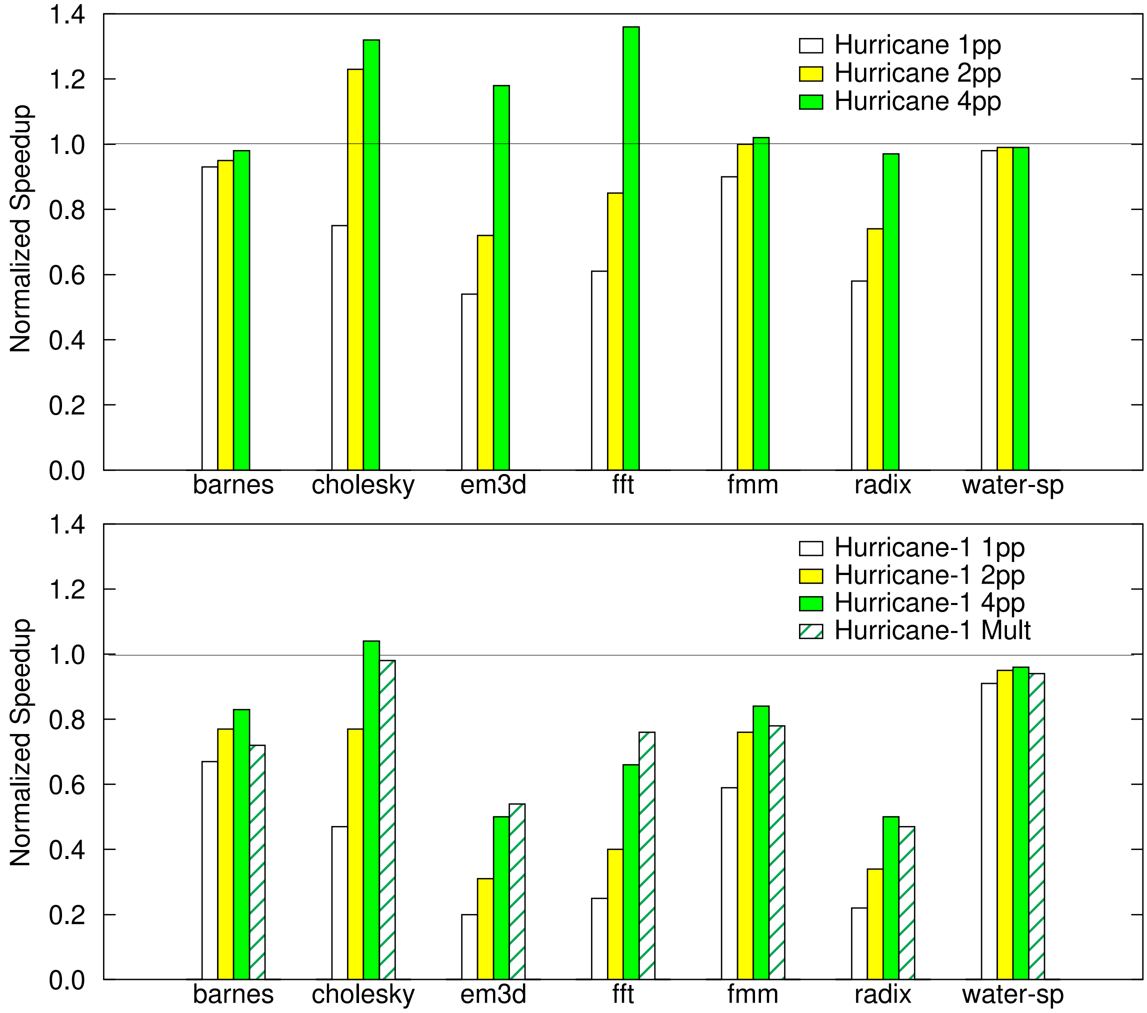


Figure 5-8. Baseline system performance comparison. The figure compares Hurricane's (above) and Hurricane-1's (below) performance with S-COMA on a cluster of 8 SMPs. The Hurricane, Hurricane-1 Mult, and S-COMA systems use 8-way SMPs. The rest of Hurricane-1 systems use additional dedicated protocol processors per SMP. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane and Hurricane-1 systems, and Hurricane-1 Mult system. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

The graphs indicate that there are three classes of applications. The first class is *water-sp* which is primarily computation-intensive and is not sensitive to protocol execution speed. All systems perform within 91% of S-COMA for *water-sp*.

The second class consists of *barnes* and *fmm* which are primarily latency-bound and do not substantially benefit from parallel protocol execution. In these applications, much of the execution time is spent in a force calculation phase between bodies in a galaxy. Communication in this phase is sporadic and evenly distributed among the nodes. These applications benefit from a reduction in protocol occupancy much more than an increase in protocol execution bandwidth.

A single-processor Hurricane system performs well (within 90% of S-COMA) running *barnes* and *fmm.* A two-processor and four-processor Hurricane system improve performance over a single-processor configuration by at most 11% and 13% respectively. A single-processor Hurricane-1 system reduces the performance to approximately within 60% of S-COMA making room for performance improvement. Nevertheless, adding protocol processors to Hurricane-1 increases the performance to at most within 84% of S-COMA. Furthermore, a Hurricane-1 with four dedicated protocol processors improves performance over Hurricane-1 Mult because the parallelism in protocol execution is not high enough to offset the multiplexed protocol scheduling overhead (Chapter 3).

The third class consists of *cholesky*, *em3d*, *fft*, and *radix* which are all bandwidth-bound applications. *Cholesky* incurs a large number of compulsory misses to data that is not actively shared. As such, the reply handlers in cholesky frequently involve reading data from memory and have high occupancies. Multiprocessor devices substantially improve performance over single-processor devices by parallelizing the memory accesses thereby increasing the reply bandwidth. A two-processor Hurricane actually improves performance over S-COMA by 23%. Limited parallelism in protocol execution, however, limits Hurricane's performance improvement over S-COMA to at most 32% with four protocol processors.

In *cholesky*, Hurricane-1's performance also extensively benefits from multiple protocol processors. Adding protocol processors significantly improves performance even up to four processors. The high protocol occupancy in Hurricane-1 results in large queueing delays at the protocol processor. Parallel protocol processors reduce queueing delays and thereby improve performance. The four-processor Hurricane-1 outperforms S-COMA, and the Hurricane-1 Mult system both performs very close to S-COMA and improves cost by eliminating the extra dedicated protocol processors.

Communication and computation in *em3d*, *fft*, and *radix* proceed in synchronous phases. Communication in these applications is highly bandwidth-intensive, bursty, and of a producer/consumer nature. In *em3d*, communication involves reading/writing memory blocks from/to neighboring processors. *Fft,* and *radix* both perform all-to-all communication with every processor exchanging its produced data with other processors.

The large degrees of sharing in *em3d*, *fft*, and *radix*, results in frequent coherence activity. Coherence events often involve executing protocol handlers that only modify state and send control messages (e.g., an invalidation). Because the handlers do not transfer data between the memory and the network, the handlers' occupancy in a software protocol is primarily due to instruction execution. Software protocol implementations, therefore, have a much higher occupancy for control messages than hardware implementations. The figure indicates that a single-processor Hurricane system at best perform within 61% of S-COMA. The single-processor Hurricane-1 systems exhibit extremely poor performance and at best reach within 25% of S-COMA's performance.

Multiprocessor Hurricane systems help mitigate the software protocol execution bottleneck in *em3d*, *fft*, and *radix*. The two-processor Hurricane systems improve performance over a single-processor system by at most 40% because parallelizing protocol execution incurs protocol state migration overhead (as discussed in Section 5.3.4.1). The four-processor Hurricane's performance ranges from competitive relative to S-COMA (in *radix*) to 36% better than S-COMA (in *fft*). Hurricane-1's also significantly improves with multiple protocol processors but at best reaches within 76% of S-COMA (in *fft* under Hurricane-1 Mult).

To summarize the results, a four-processor Hurricane system on the average increases speedups by 12% over S-COMA, and a four-processor Hurricane-1 on the average performs within 76% of S-COMA. More importantly, the most cost-effective Hurricane-1 Mult system performs within 74% of an all-hardware S-COMA system without requiring extra dedicated protocol processors.

5.3.4.2 Impact of Clustering Degree. The degree of clustering refers to the number of processors in every SMP node. This section evaluates the impact of clustering degree on the relative performance of the systems while maintaining the number of processors and the total amount of memory in the system constant.

Clustering typically increases the total amount of protocol traffic generated per machine node [SFH+97]. The increase in protocol traffic, however, depends on an application's sharing patterns. On the one hand, clustering allows processors to share a single cached copy of remote data, reducing protocol traffic generated per processor. On the other hand, in the absence of sharing, clustering may linearly increase protocol traffic in/out of a node with the increase in the number of processors per node. Clustering also reduces the number of network interfaces in the system, placing higher demands on the protocol processors favoring parallel protocol execution.

Figure 5-9 compares Hurricane's performance against S-COMA for a cluster of 16 4-way SMPs (above) and a cluster of 4 16-way SMPs (below). The graphs indicate that a higher clustering degree increases the performance gap between the single-processor Hurricane devices and S-COMA in most of the applications. This result indicates that queueing delays due to a smaller number of network interface devices in the system has a higher impact on performance than the gains from sharing remote data.

Multiple protocol processors in Hurricane systems help close the performance gap between software and hardware implementations. With a clustering degree of 16, a four-processor Hurricane system outperforms S-COMA in all the applications except for *water-sp*; Hurricane's performance in *water-sp* is within 99% of S-COMA. With an

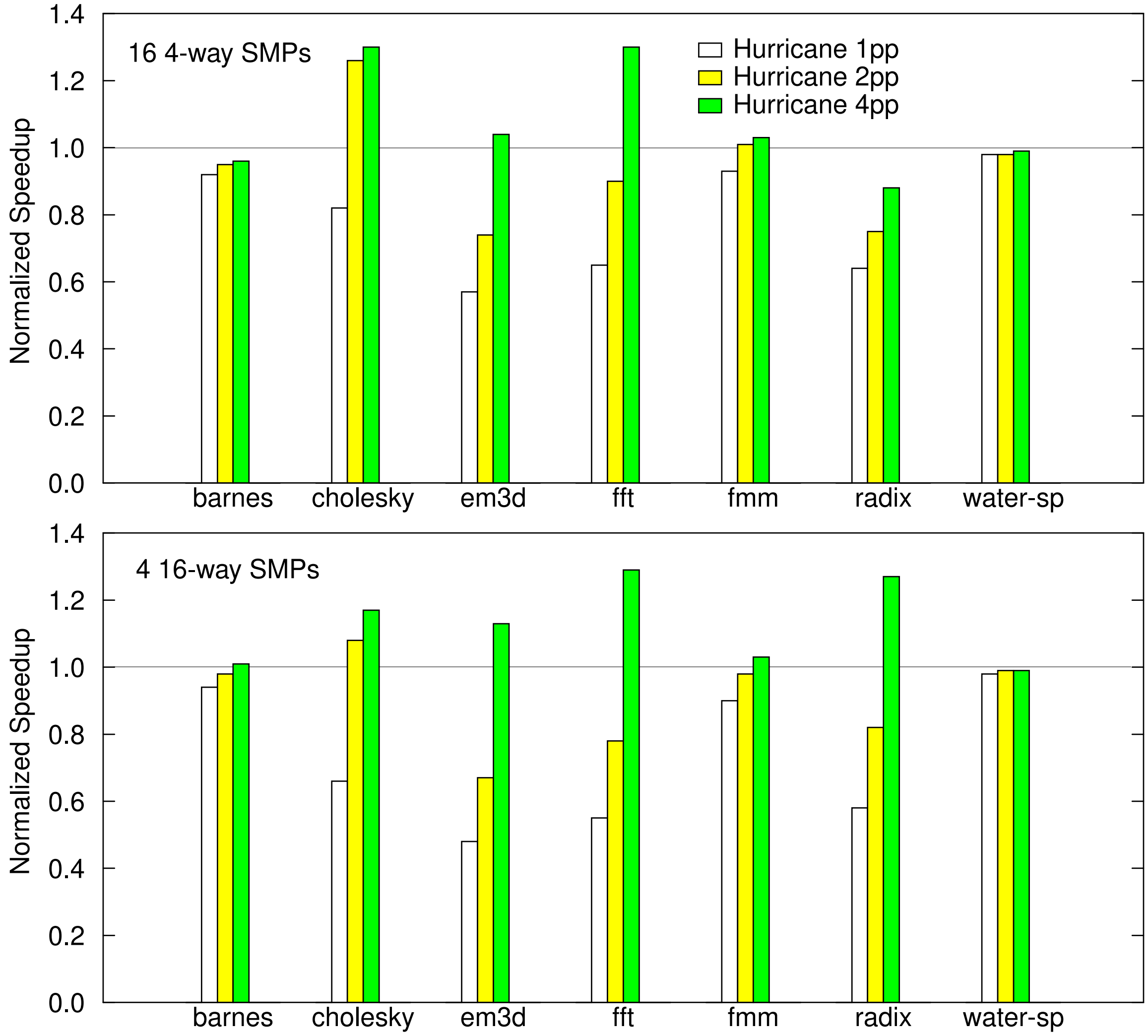


Figure 5-9. Impact of clustering degree on Hurricane's performance. The figure compares performance in S-COMA and Hurricane on a cluster of 16 4-way SMPs (above), and a cluster of 4 16-way SMPs (below). The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

increase in the clustering degree from 4 to 16, four protocol processor in Hurricane increase performance from a 7% to a 13% improvement over S-COMA's.

Figure 5-10 illustrates the impact of clustering degree on Hurricane-1's performance. A high clustering degree has large impact on the single-processor Hurricane-1's performance. Because of the poor performance of single-processor (base) system, even the large

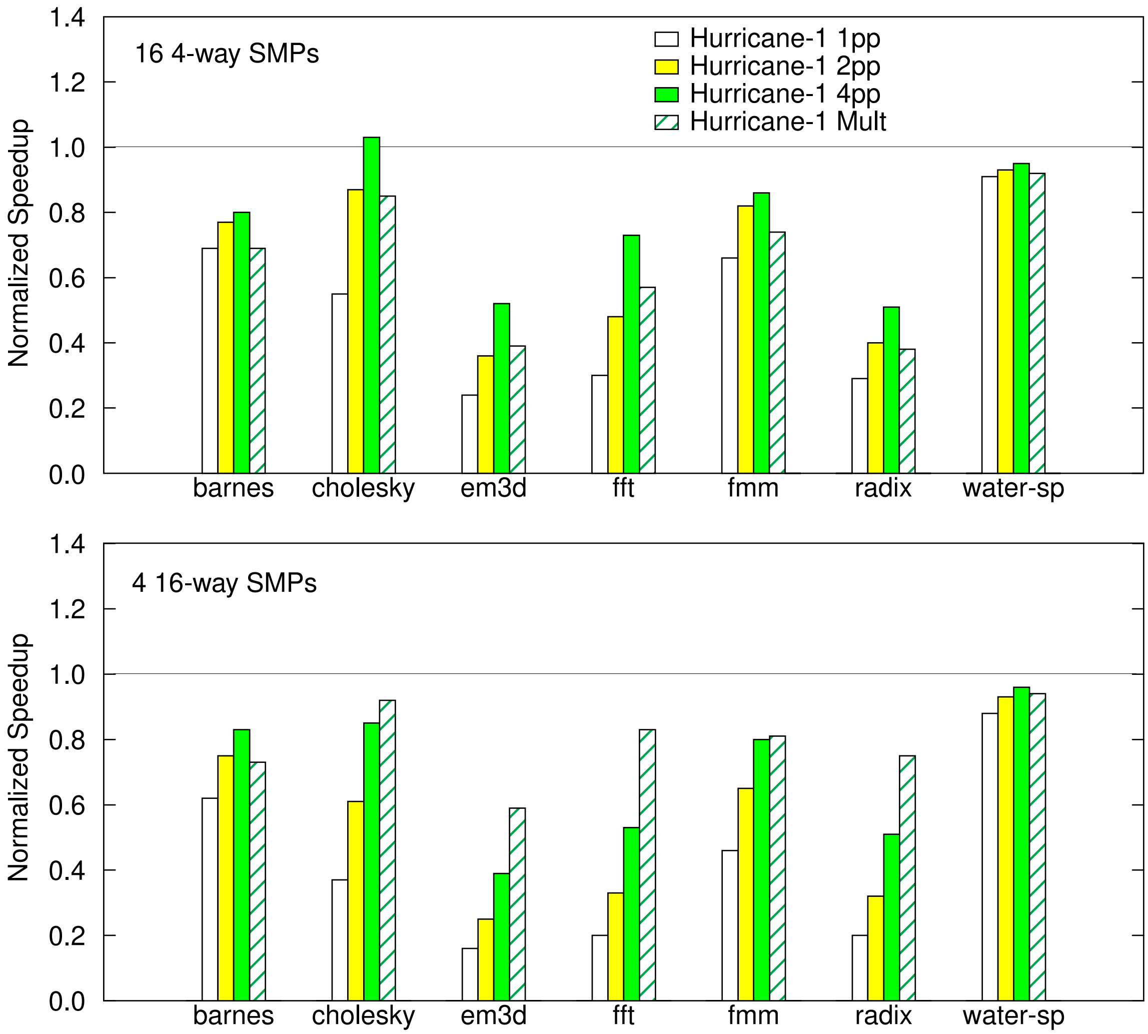


Figure 5-10. Impact of clustering degree on Hurricane-1's performance. The figure compares performance in S-COMA and Hurricane-1 on a cluster of 16 (above) and 4 (below) SMPs. The S-COMA and Hurricane-1 Mult systems use 4-way (above) and 16-way (below) SMPs respectively. The rest of the Hurricane-1 systems use additional dedicated protocol processors per SMP. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane-1, and Hurricane-1 Mult systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

performance improvements due to four protocol processors fail to make the Hurricane-1 system competitive with S-COMA. Not surprisingly, Hurricane-1 Mult substantially benefits from a high clustering degree and outperforms a four-processor Hurricane-1 system in all bandwidth-bound applications. Increasing the clustering degree from 4 to 16 also allows Hurricane-1 Mult to improve performance from 65% to 80% of S-COMA.

5.3.4.3 Impact of Block Size. An increase in the protocol block size increases the overall protocol bandwidth out of a node. Large block sizes also increase the fraction of protocol occupancy due to data transfer time between memory and the network. Amortizing software protocol overhead over a larger overall occupancy reduces the performance gap between software and hardware protocol implementations.

Large blocks, however, result in false sharing in applications with very fine sharing granularities thereby increasing protocol activity. Higher protocol activity intensifies queueing at the protocol processors and results in a larger performance gap between hardware and software protocol implementations. Parallelizing protocol execution alleviates the performance loss due to false sharing by reducing queueing at the protocol processors.

Figure 5-11 compares Hurricane's performance against S-COMA for a 32-byte protocol (above) and a 128-byte protocol (below). The graphs corroborate the intuition that an increase in the block size increases the performance gap between single-processor Hurricane systems and S-COMA in some applications. With a 128-byte block, *cholesky*, *em3d*, *fft*, *radix, water-sp* all exhibit better performance under single-processor Hurricane systems relative to S-COMA. *Barnes* and *fmm* share data at very fine granularities, suffer from false sharing with 128-byte blocks, and therefore experience a larger performance gap between the single-processor Hurricane and S-COMA.

The graphs also indicate that a large block size not only favors the single-processor base case Hurricane system, but also the multiprocessor systems. Two protocol processors make a Hurricane system competitive with S-COMA in all the applications. A four-processor Hurricane system on the averages speeds up application execution time by 20% over S-COMA. These results, indicate that pipelining protocol event execution to allow for multiple outstanding memory requests may allow single-processor devices to achieve a high protocol bandwidth with large blocks.

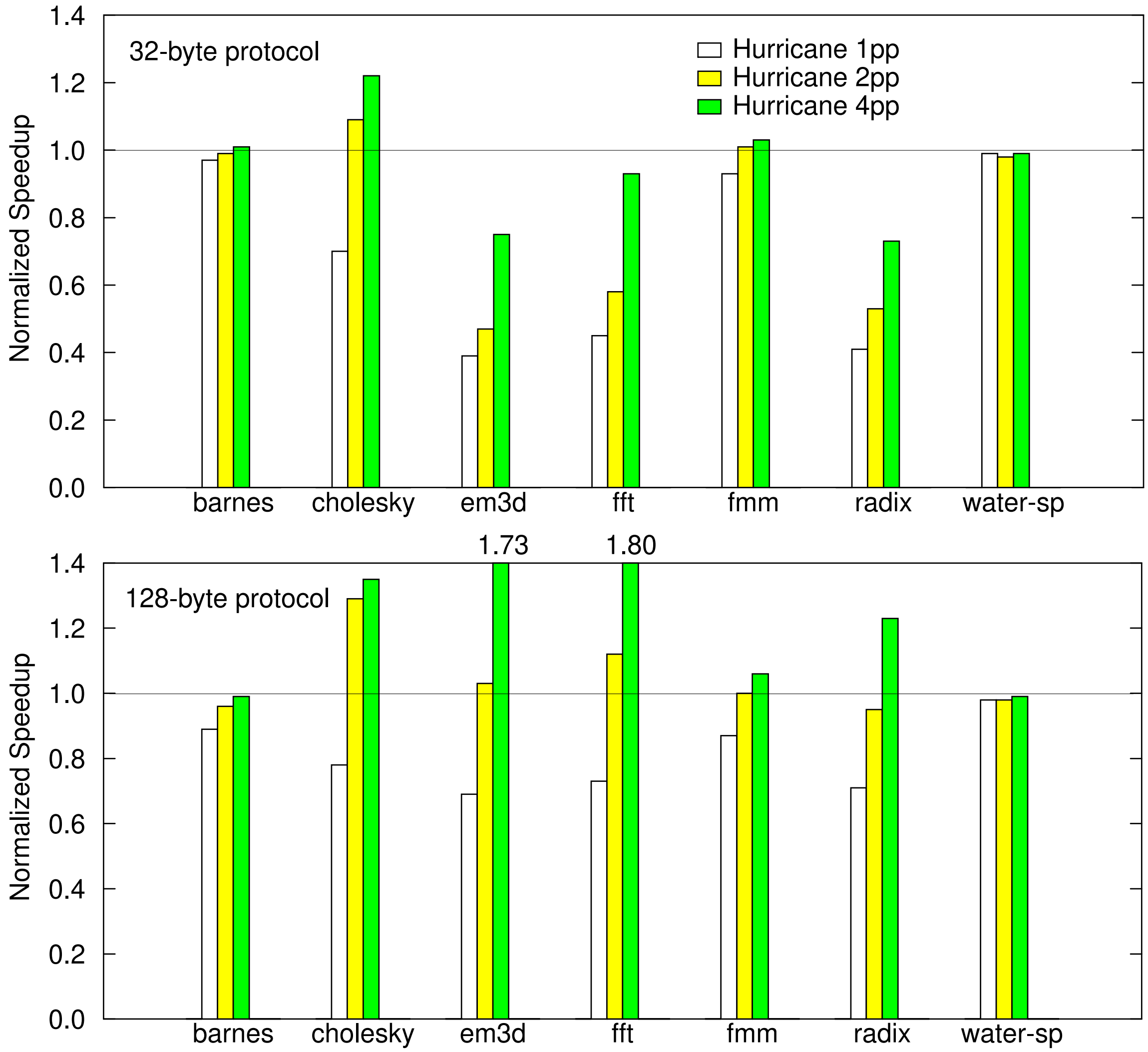


Figure 5-11. Impact of block size on Hurricane's performance. The figure compares performance in S-COMA and Hurricane for a 32-byte (above) and a 128-byte (below) block protocol. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

Figure 5-12 illustrates the impact of protocol block size on Hurricane-1's performance. A large block size has a higher impact on the performance of a single-processor Hurricane-1 system as compared to Hurricane. Large blocks benefit systems with high software protocol overheads (as in Hurricane-1) allowing the system to amortize the overhead over a larger protocol occupancy. Much as in Hurricane systems, multiprocessor Hurricane-1

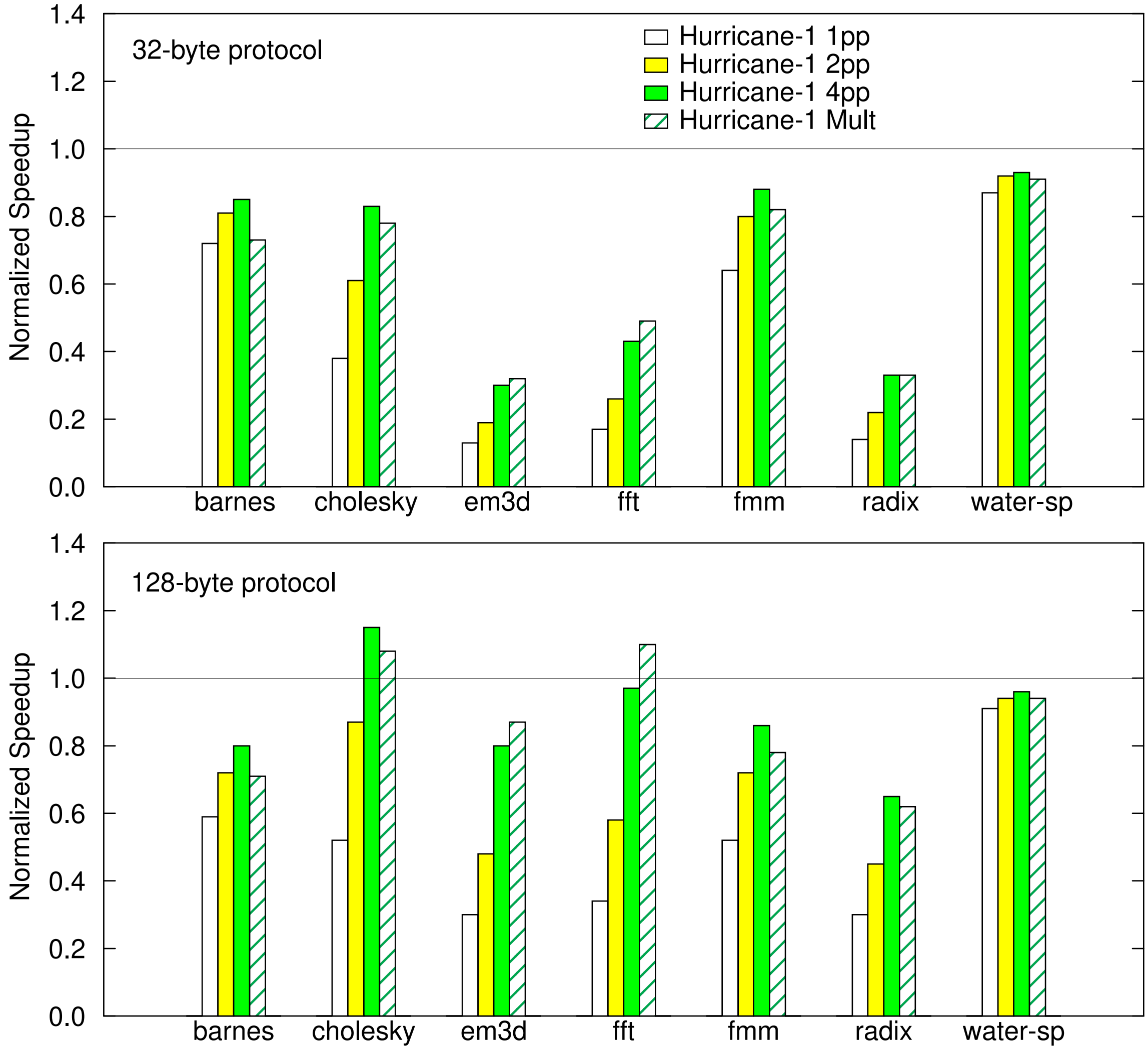


Figure 5-12. Impact of block size on Hurricane-1's performance. The figure compares performance in S-COMA and Hurricane-1 for a 32-byte (above) and a 128-byte (below) block protocol. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane-1, and Hurricane-1 Mult systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

systems close the performance gap between Hurricane-1 and S-COMA. A four-processor Hurricane-1 system, and a Hurricane-1 Mult system both reach approximately within 88% of S-COMA's performance.

5.3.4.4 Protocol Processor Performance. The microbenchmark results in Section 5.3.4.4 indicated that protocol processor clock frequency has a lower impact on the reply band-

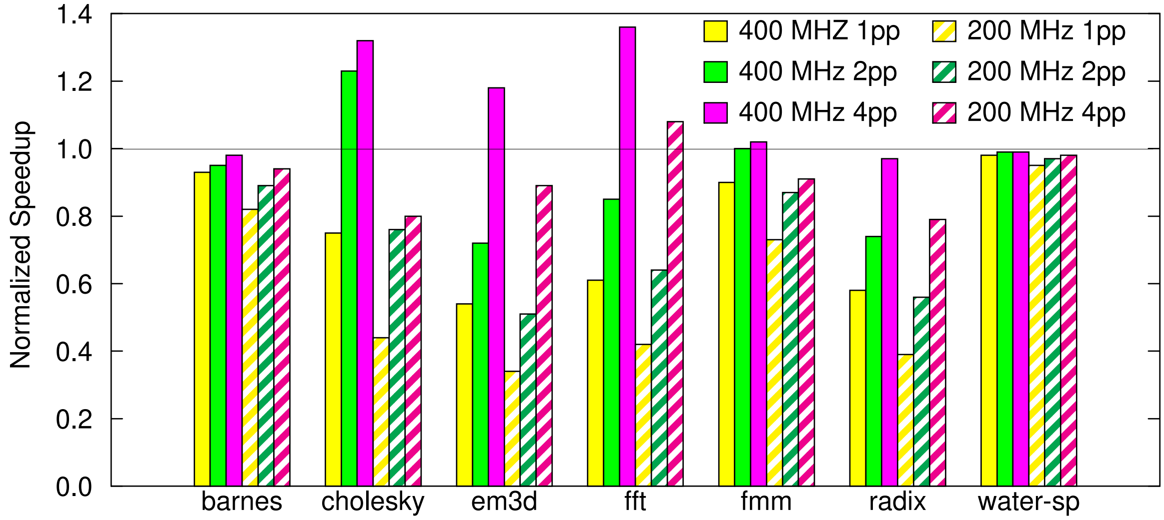


Figure 5-13. Impact of protocol processor clock rate on Hurricane's performance. The figure compares Hurricane's performance with 400-MHz and 200-MHz protocol processors on a cluster of 8 8-way SMPs. The graphs plot application speedups normalized to S-COMA for one-, two-, and four-processor Hurricane devices. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

width than the number of protocol processors. The results also indicated that the clock frequency may have a low or high impact on the request/response bandwidth depending on the number of protocol processors.

Figure 5-13 plots application speedups under 400-MHz and 200-MHz Hurricane devices normalized to those under S-COMA. The graphs plot performance for one-, two-, and four-processor Hurricane devices. The graphs indicate that doubling the number of processors in a previous-generation single-processor Hurricane device results in comparable performance to upgrading the single processor to current-generation technology. This result indicates that the applications equally utilize the available reply bandwidth (benefitting from a larger number of processors) and the request/response bandwidth (benefitting from a higher clock frequency) in the system.

The trade-off between a higher clock frequency and a larger number of processors becomes more visible for two-processor devices in bandwidth-bound applications. *Cholesky* exhibits limited parallelism in protocol event activity and as such benefits from a higher clock frequency. Two protocol processors can not exploit all the event parallelism in *em3d*, *fft*, and *radix*. These applications highly benefit from parallelizing protocol event execution. Furthermore, protocol state migration equally impacts protocol occupancies in multiprocessor Hurricane systems with higher or lower clock frequencies; protocol state migration overhead is time spent in the memory system and is independent of processor speed. In bandwidth-bound applications with high protocol event parallelism, adding more protocol processor from previous-generation technology improves performance over upgrading the existing processors to a newer technology.

5.4 Related Work

Erlichson, et al. study the impact of using multiple dedicated protocol processors on system's performance in a cluster of SMPs [ENCH96]. Their study focuses on page-based rather than fine-grain software DSM and they do not consider hardware support for parallel protocol execution. Holt, et al. study the effect of protocol occupancy on system's performance and conclude that high protocol occupancy significantly limits performance for bandwidth-bound applications [HHS+95]. Lovett, et al. also study a software protocol implementation on an embedded processor and conclude that for some bandwidth-bound database applications, the high protocol occupancy of software implementations severely limits performance [LC96]. Reinhardt, et al. compare software and hardware implementations of fine-grain DSM but only consider single-threaded protocol execution [RPW96,RLW94].

Michael, et al. compare S-COMA's performance with that of network interfaces with two embedded processors [MNLS97]. They implement a static partitioning of request/response handlers and reply handlers between the two protocol processors. They conclude that even with two protocol processors, software implementations significantly lag in performance for bandwidth-bound applications. Their results, however, also indicate a large load imbalance between the two protocol processors due to static partitioning of handlers.

5.5 Summary

PDQ is a novel set of mechanisms for efficient parallel execution of fine-grain software protocols. This chapter proposes and evaluates hardware support for parallel execution of fine-grain software DSM using the PDQ mechanisms. The chapter introduces two designs, Hurricane and Hurricane-1, for parallel fine-grain software DSM and compares their performance to an all-hardware S-COMA implementation. Hurricane integrates embedded processors, fine-grain sharing, and networking hardware into a single device. Hurricane-1 integrates fine-grain sharing and networking hardware into a single device, but relies on the host processors to run the software protocols.

Results from a set of microbenchmarks and seven fine-grain shared memory applications indicated that:

- PDQ helps significantly improve the performance of software protocol implementations, alleviating the software protocol execution bottleneck.
- A Hurricane system with four embedded processors either outperforms or performs within 95% of S-COMA.
- A cost-effective Hurricane-1 Mult system (with no extra dedicated protocol processors) on the average performs as close as within 75% of S-COMA.

Besides the main conclusions of the study, further evaluation of the systems led to the following results:

- Higher clustering degree increases demand for protocol execution and further increases performance improvement from parallel protocol execution.
- A high clustering degree makes Hurricane-1 Mult more competitive with S-COMA.
- Larger block sizes both reduce the performance gap between software and hardware implementations, and boost the performance improvement from parallel protocol execution.
- A multiprocessor Hurricane system benefits more from an increase in the number of protocol processors than an upgrade to faster protocol processors. Much of the proto-

col occupancy is spent moving data between the memory and the network. Multiple processors increase the protocol bandwidth by parallelizing memory requests.

Chapter 6

Conclusions

Parallel computing is becoming ubiquitous with the wide availability of SMP multiprocessors in the form of desktop machines and servers. Computer designers have begun to construct large-scale parallel computers using SMPs as cost-effective building blocks. SMP clusters interconnect several SMPs into a single parallel computer using a high-bandwidth low-latency commodity network. To communicate across SMP nodes, processors send messages through the interconnection network. Applications and systems use a variety of software protocols to coordinate and schedule the communication. This thesis proposes and evaluates techniques to improve fine-grain software protocol performance in an SMP cluster.

6.1 Thesis Summary

The contributions of this thesis are:

- a taxonomy for software protocol execution semantics;
- a novel set of mechanisms, *PDQ*, for parallel fine-grain software protocol execution;
- a taxonomy for software protocol scheduling policies on SMP processors;
- an evaluation of two scheduling policies for single-threaded protocol execution;
- an evaluation of two fine-grain DSMs—*Hurricane* and *Hurricane-1*—using PDQ to parallelize the software protocol execution.

The above contributions fall into two high-level strategies to improve software protocol performance. Much like an ordinary application, software protocol performance can increase by either (i) improving the sequential execution or (ii) parallelizing the execution of the protocol code.

The first contribution of the thesis is a taxonomy for protocol execution semantics: *single-threaded* or sequential execution, and *multi-threaded* or parallel execution. Traditionally, distributed memory parallel computers execute fine-grain software protocols sequentially either on embedded processor on the network interface card or on a node's single commodity processor. This thesis explores techniques for parallel execution of fine-grain software protocols.

Fine-grain software protocols by definition have very short running times. Conventional locking techniques (e.g., software spin-locks) to synchronize accesses to common systems resources from parallel protocol threads would incur prohibitively high overheads in fine-grain software protocols. This thesis proposes *PDQ*, a novel set of mechanisms for parallel execution of synchronization-free protocols. PDQ is based on the observation that by partitioning system resources among protocol threads, multiple protocol threads can execute in parallel free of explicit synchronization mechanisms.

Another approach to improve software protocol performance is to accelerate the sequential execution of the software. Protocol execution consists of scheduling and invoking the protocol's execution and subsequently executing the protocol code. This thesis evaluates protocol scheduling policies on systems in which software protocols execute on SMP processors. By not providing embedded protocol processors on a custom network interface card to execute software protocols, such systems reduce both hardware complexity and cost.

Another contribution of this thesis is a taxonomy for protocol scheduling policies. On an SMP node, software protocols can either run on one or more dedicated protocol processors, or the system software can schedule (i.e., multiplex) the application and protocol execution on the same processors. Therefore, SMPs give rise to two classes of protocol scheduling policy: *dedicated* and *multiplexed*. A dedicated policy eliminates scheduling overhead and maximizes communication throughput by dedicating one or more SMP processors to always execute the protocol software. A multiplexed policy, however, maximizes processor utilization by allowing all processors to contribute to computation and dynamically schedules the protocol code on one or more processors when the application needs to communicate.

Cost-effective (rather than high-performance) SMP clusters use low-cost small-scale SMPs and interconnect them with commodity networking hardware with little or no support to accelerate protocol execution. These systems typically execute software protocols in a single thread on every SMP. In the context of single-threaded protocol execution, this thesis asks the question "*when does it make sense to dedicate one processor in each SMP node specifically for protocol processing?*" The central issue is when do the overheads eliminated by a dedicated protocol processor offset its lost contribution to computation? The thesis addresses this question by examining the performance and cost-performance trade-offs of two scheduling policies for a single-threaded protocol:

- *Fixed*, a dedicated policy where one processor in an SMP is dedicated to execute the protocol thread, and
- *Floating,* a multiplexed policy where all processors compute and alternate acting as protocol processor.

Performance results from running shared-memory and synthetic applications on a simulation model for software fine-grain DSM together with simple cost model [WH95] indicate that:

- **Fixed is advantageous for light-weight protocols**. A dedicated protocol processor benefits light-weight protocols (e.g., fine-grain DSM) much more than coarse-grain protocols (e.g., page-based DSM) because the overheads eliminated by a dedicated protocol processor account for a significant fraction of protocol execution time in light-weight protocols.

- **Fixed offers superior performance when there are four or more SMP processors**. Systems with very small-scale SMPs (with less than four processors) underutilize a dedicated protocol processor. Eliminating protocol scheduling overheads does not significantly impact execution time in these systems. Because of a small number of SMP processors in these systems, the opportunity cost in lost computation due to a dedicated protocol processor is high. These systems would rather use all the processors for computation and dynamically schedule a protocol thread to execute on one processor.
- **Floating's performance is not very sensitive to protocol invocation overhead**. A two-orders-of-magnitude increase in protocol invocation (i.e., interrupt) overhead increases execution time in a two-processor SMP cluster under the Floating policy by at most 45%. In comparison, the same increase in interrupt overhead increases execution time in uniprocessor-node systems by 400%. It is highly likely that at least one processor in an SMP node is idle at any given time thus acting as protocol processor thereby eliminating interrupts.
- **Fixed is the most cost-effective design in fine-grain systems with high protocol invocation overhead.** Given large enough SMPs, software protocol execution eventually becomes a bottleneck and Fixed improves performance over Floating. Although increasing the size of an SMP may improve performance, it does not always result in a cost-effective system; the performance improvement may not justify the cost of additional processors. Fine-grain systems with high protocol invocation overheads allow SMP nodes to significantly improve performance over uniprocessor nodes by eliminating the protocol invocation overheads. These systems become most cost-effective—with respect to a uniprocessor-node system—with four or more SMP processors. At such a design point, the cost of an additional dedicated protocol processor becomes negligible and is easily offset by the performance improvement.

This thesis also examines two parallel fine-grain DSM systems based on PDQ. The thesis first proposes *PTempest*, a set of mechanisms for implementing parallel fine-grain software DSM protocols derived from Tempest [Rei95] and PDQ. The thesis evaluates two hardware implementations of PTempest, *Hurricane* and *Hurricane-1*. Much like Typhoon [RPW96], Hurricane integrates one or more embedded processors with fine-grain shared-memory support and the networking hardware (i.e., message queues) on a custom device. Similarly, Hurricane-1 is like Typhoon-1 [RPW96] and integrates shared-memory and messaging hardware on a custom device but relies on the node's commodity processors to run the software protocol.

To gauge the impact of parallel protocol execution on software fine-grain DSM's performance, this thesis compares the Hurricane systems against S-COMA, a hardware implementation of fine-grain DSM [HSL94]. Results from running shared-memory applications on simulation models for the Hurricane systems and an idealized (conservative) simulation model for S-COMA indicate that:

- **PDQ helps significantly improve software fine-grain DSM's performance.** Single-processor Hurricane and Hurricane-1 systems each perform within 24% and 53% of S-COMA respectively. Parallelizing the protocol execution allows a Hurricane system with four embedded processors to either outperform or perform within 95% of S-COMA. Similarly, a Hurricane-1 system with four extra dedicated SMP processors reaches within 76% of S-COMA's performance.
- **PDQ's impact is highest for Hurricane-1 systems with large-scale SMPs.** A Hurricane-1 system can multiplex SMP processors to both compute and execute software protocols. Under a multiplexed scheduling policy, large-scale SMPs (with eight or more processors) significantly increase the number of processors contributing to parallel protocol execution, improving performance of Hurricane-1 to within 80% of S-COMA.
- **Large-scale SMPs obviate the need for extra dedicated SMP processors**. Although multiplexed policies incur high protocol scheduling and execution overhead, the high

parallelism in protocol execution due to a large number of SMP processors offsets the protocol scheduling and execution overhead. Large-scale SMPs (with eight or more processors) allow a Hurricane-1 system with multiplexed protocol scheduling to outperform or perform close to a Hurricane-1 system with a small number of (e.g., up to four) extra dedicated protocol processors. A multiplexed policy is also advantageous because it reduces cost by obviating the need for extra dedicated SMP processors.

References

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[ACP95] Tom Anderson, David Culler, and David Patterson. A Case for NOW (networks of workstations). *IEEE Micro*, 15(1), February 1995.

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1989.

[Aga92] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[AS88] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–25, August 1988.

[BALL90] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[BCF+93] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka. Fortran 90d/hpf compiler for distributed memory mimd computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[BCL+95] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote queues: Exposing message queues or optimization and atomicity. In *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory. In *Proceedings of the 17th*

Annual International Symposium on Computer Architecture, pages 125–134, June 1990.

[BDFL96] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected User-level DMA for the SHRIMP Network Interface. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[Bel85] C. Gordon Bell. Multis: A new class of multiprocessor computers. *Science*, 228:462–466, 1985.

[BG93] Mats Bjoerkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *SIGCOMM '93*, pages 74–83, September 1993.

[BG97] Doug Burger and James R. Goodman. Billion-transistor architectures. *IEEE Computer*, 30(9):46–48, September 1997.

[BGP+94] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[BH86] J.E. Barnes and P. Hut. A hierarchical o(n log n) force calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 207–216, August 1995.

[BK93] Henri E. Bal and M. Frans Kashoek. Object distribution in Orca using compile-time and run-time techniques. In *OOPSLA '93: Eighth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 162–177, October 1993.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[BN84] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[BTK90] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[BVvE95] Anindya Basu, Vineet Buch Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. Technical report, Department of Computer Science, Cornell University, 1995. to appear.

[CA96] R. Clark and K. Alnes. An SCI interconnect chipset and adapter. In *Symposium Record, Hot Interconnects IV*, August 1996.

[CAA+95] Derek Chiou, Boon S. Ang, Arvind, Michael J. Becherle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StartT-ng: Delivering seamless parallel computing. In *Proceedings of EURO-PAR '95*, Stockholm, Sweden, 1995.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[CF93] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[Cha97] Satish Chandra. *Software Techniques for Customizable Distributed Shared Memory.* PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[CLR94] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, San Jose, California, 1994.

[CPWG97] Alan Charlesworth, Andy Phelps, Ricki Williams, and Gary Gilbert. Gigaplane-XB: Extending the Ultra Enterprise family. In *Symposium Record, Hot Interconnects V*, July 1997.

[CSBS95] Fredric T. Chong, Shamik D. Sharma, Eric A. Brewer, and Joel Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. *Parallel Processing Letters: Special Issue on Partitioning and Scheduling for Parallel and Distributed Systems*, December 1995.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, October 1991.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[DPD94] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM '94*, pages 2–13, August 1994.

[ENCH96] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[FLR+94] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[FW94] Babak Falsafi and David A. Wood. Cost/performance of a parallel computer simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, July 1994.

[FW96] Babak Falsafi and David A. Wood. When does dedicated protocol processing make sense? Technical Report 1302, Computer Sciences Department, University of Wisconsin–Madison, February 1996.

[FW97a] Babak Falsafi and David A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

[FW97b] Babak Falsafi and David A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[FW97c] Babak Falsafi and David A. Wood. Scheduling communication on an SMP node parallel machine. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, February 1997.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, June 1991.

[GJ91] David B. Gustavson and David V. James, editors. *SCI: Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications*, volume P1596/D2.00 18Nov91. IEEE, November 1991. Draft 2.00 for Recirculation to the Balloting Body.

[HHS+95] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[HKO+94] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 274–285, 1994.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling fortran d for mimd distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HSL94] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA node implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, page ?, January 1994.

[HT93] W. Daniel Hillis and Lewis W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[Int90] Intel Corporation. *iPSC/2 and iPSC/860 User's Guide*. Intel Corporation, 1990.

[Int93] Intel Corporation. Paragon technical summary. Intel Supercomputer Systems Division, 1993.

[int97] 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC). http://developer.intel.com/design/pcisets/datashts/290566.htm, 1997.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 213–228, December 1995.

[Kai93] Mathias Kaiserswerth. The parallel protocol engine. *IEEE/ACM Transactions on Networking*, 1(6):650–663, December 1993.

[KC94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, October 1994.

[KDCZ93] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. Technical Report 93-214, Department of Computer Science, Rice University, November 1993.

[KS96] Magnus Karlsson and Per Stenstrom. Performance evaluation of a cluster-based multiprocessor build from ATM switches and bus-based multiprocessor servers. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[K+94] Jeffrey Kuskin et al. The stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[LC96] Tom Lovett and Russel Clapp. STiNG: A CC-NUMA compute system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LHPS97] Beng-Hong Lim, Phillip Heidelberg, Pratap Pattanik, and Marc Snir. Message proxies for efficient, protected communication on SMP clusters. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, February 1997.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[LS95] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[Mei93] Meiko World Inc. Computing surface 2: Overview documentation set, 1993.

[MFHW96] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[MNLS97] Maged Michael, Ashwini K. Nanda, Beng-Hong Lim, and Michael L. Scott. Coherence controller architectures for SMP-based CC-NUMA mulitprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[MPO95] David Mosberger, Larry L. Peterson, and Sean O'Malley. Protocol latency: Mips and reality. Technical Report TR 95-02, Department of Computer Science, University of Arizona, 1995.

[MRF+97] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[Muk98] Shubhendu S. Mukherjee. *A Simulation Study of Network Interface Design Alternatives*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1998.

[PC94] Scott Pakin and Andrew A. Chien. The impact of message traffic on multicomputer memory hierarchy performance. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1994.

[Pfi95] Robert W. Pfile. Typhoon-Zero implementation: The vortex module. Master's thesis, October 1995.

[QB97] Xiaohan Qin and Jean-Loup Baer. On the use and performance of explicit communication primitives in cache-coherent multiprocessor systems. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, February 1997.

[Rei95] Steven K. Reinhardt. Tempest interface specification (revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[Rei96] Steven K. Reinhardt. *Mechanisms for Distributed Shared Memory*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1996.

[RFW93] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[RSG93] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, June 1993.

[SBS93] Per Stenstrom, Mats Brorsson, and Lars Sandberg. Adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[SCB93] Daniel Stodolsky, J. Brad Chen, and Brian Bershad. Fast interrupt priority management in operating systems. In *Second USENIX Symposium on Microkernels and Other Kernel Archtitectures*, pages 105–110, September 1993. San Diego, CA.

[Sch97] Ioannis Schoinas. *Fine-Grain Distributed Shared Memory on a Cluster of Workstations*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[SFH+97] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James Larus, and David A. Wood. Sirocco: Cost-effective fine-grain distributed shared memory. Submitted for publication, July 1997.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[SGA97] Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal. Fine-grain software distributed shared memory on SMP clusters. Technical Report 97/3, Digital Equipment Corporation, Western Research Laboratory, February 1997.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[SH91] Richard Simoni and Mark Horowitz. Dynamic pointer allocation for scalable cache coherence directories. In *International Symposium on Shared Memory Multiprocessing*, April 1991.

[SKT96] James D. Salehi, James F. Kurose, and Don Towsley. The effectiveness of affinity-based scheduling in multiprocessor networking. 4(4), August 1996.

[SL90] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, April 1990.

[SP88] J. E. Smith and A. R. Plezkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

[SS92] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of the Summer '92 USENIX Conference*, pages 1–9, June 1992.

[TG89] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP)*, pages 159–166, December 1989.

[TL94] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, 1994.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[VZ91] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 26–40, October 1991.

[WG94] David Womble and David Greenberg. LU factorization and the LINPACK benchmark on the Intel Paragon. ftp://ftp.cs.sandia.gov/pub/papers/dewombl/paragon_linpack_benchmark.ps, March 1994.

[WGH+97] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu Thomas Wicki, and Winfried Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[WH95] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, February 1995.

[WHJ+95] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communic ation with computation. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, July 1995.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.