Design and Evaluation of Multi-Protocol Communication on a Cluster of SMP's

by

Steven Sam Lumetta

B.A. (University of California at Berkeley) May 1991
M.S. (University of California at Berkeley) December 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David E. Culler, Chair
Professor Katherine A. Yelick
Professor Eric A. Brewer
Professor Shamit Kachru

Fall 1998

The dissertation of Steven Sam Lumetta is approved:

Chair Date

Date

Date

Date

University of California at Berkeley

Fall 1998

Design and Evaluation of Multi-Protocol Communication on a Cluster of SMP's

Abstract

Design and Evaluation of Multi-Protocol Communication on a Cluster of SMP's

by

Steven Sam Lumetta
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David E. Culler, Chair

Modern computers have deep and increasingly complex data hierarchies. The task of managing the motion of data between the levels is of prime importance, particularly with regard to communication. It requires a combination of automatic control and application-specific knowledge. Automatic control handles the bulk of the work, applying heuristics grounded in general principles and system parameters. Application-specific knowledge allows a program to make more effective use of the system, integrating the application's needs with the capabilities of the architecture. Striking a proper balance between these two approaches is not easy, and effective abstractions are a necessary aid in finding appropriate solutions.

This thesis addresses the management of data motion in the context of a cluster of symmetric multiprocessors, or SMP's. The shift from uniprocessors to multiprocessors as the basic unit of cluster computing reflects the trend toward deeper hierarchies of data, extending the hierarchy in a cluster of workstations with an intermediate level—the memory interconnect—at which processors communicate and share physical resources. The resulting systems, known as Clumps, offer potentially superior performance to applications capable of exploiting the tight coupling within each SMP. For some applications, however, resource contention and other interactions degrade performance relative to a cluster of uniprocessors.

We address interprocess communication on a Clump through a uniform message-passing interface that exposes the performance of the underlying hardware. The interface transparently routes messages through the appropriate medium, providing the necessary automatic control to allow a programmer to obtain reasonable performance with minimal

effort, yet provides the locality information necessary to support the incremental use of application-specific knowledge.

In constructing our uniform interface, we carefully engineer and tune a transport protocol for passing messages across a cache-coherent interconnect, introducing in the process a new concurrent queue algorithm that obtains good performance on both dedicated and multiprogrammed machines. We integrate this protocol with a similarly well-engineered network protocol to present a uniform interface to programmers. We expose the problems involved with coupling protocols of disparate speed and present a solution that dynamically tunes our communication layer to the underlying architecture. Using both applications and benchmarks derived from the message-passing literature, we measure the performance of our system and highlight the phenomena that counteract the advantages of faster communication. Through a model of shared communication resources, we explain these same phenomena analytically.

The communication layer developed in this thesis demonstrates the value of a uniform interface in abstracting a hierarchy below the level addressed by an application programmer. The concurrent queue algorithm illustrates a good approach to the development of concurrent data structures, backed up by a wealth of performance comparisons. The dynamic adaptation solution also proves very effective, enabling applications to address a general Clump architecture, from an SMP to a NOW, with only a single binary. Similar solutions might be used to address a range of other problems. Taken in part, the shared memory protocol is also a powerful building block for higher-level interfaces within an SMP.

Through the work described in this thesis, we develop an understanding of the issues for fast, user-level communication between processes in a Clump and for the broader problem of coupling levels of a hierarchical system within a single abstraction.

Professor David E. Culler
Dissertation Committee Chair

To my wife,

Jennie Chung-Yi Hsu-Lumetta,

and our sons,

Andrew Jia-Wei Lumetta

and

Justin Jia-Yao Lumetta

Acknowledgements

In writing this thesis, I have examined a broad range of previous efforts, and have found them both intimidating in their importance and inspiring in their irrelevance. I now thank the many who aided me in the process of building my own vast and trunkless legs of stone.

First, I wish to thank my advisor, David Culler, for his continued support and encouragement throughout my graduate career. We have had many fun times and many hard times, and I will always remember his help. As I start my new career as a faculty member, I can only hope to help my students develop their own interests and skills as effectively as David has helped me to develop mine. I particularly appreciate his aid in enabling me to understand the importance of communicating with the intellectual community. I tend to be a perfectionist, and without his guidance might still be trying to figure out where my last two cycles had gone so as not to have to admit that I do not know.

Kathy Yelick has been very supportive over the years, and aided extensively in pointing me to the right sources for concurrent algorithms and in helping me to understand them. Eric Brewer has been a source of inspiration and insight in developing my ideas. Shamit Kachru, a longtime friend and recent addition to the faculty, agreed to read my thesis on short notice. My thanks also go to my other committee members and informal advisors through the years: Jim Gray, Jim Demmel, Larry Rowe, Phil Colella, and Richard Fateman.

Andrea Dusseau, Alan Mainwaring, and Girija Narlikar, my officemates and friends, gave helpful suggestions on all aspects of the work, from design to implementation to measurement to analysis. I appreciate most their camaraderie and patience with my ranting and raving as I explored the landscape that has become my thesis. In previous years of graduate school, I have also had the pleasure to share an office with Seth Goldstein. Michael Mitzenmacher, a longtime and very good friend, gave freely of his time to comment on talks, drafts, and general wisdom, in addition to inviting me to an occasional cider to put things in perspective. Arvind Krishnamurthy, with whom I began and am now ending graduate school, has always been a close companion. We share equally broad interests and a common focus in parallelism. Countless other friends have come and gone during my seven and a half year tenure as a graduate. I cannot name them all here, but I often think of them. I also wish to express my gratitude to the late Ted Einwohner for his support and encouragement as well as his help early in the thinking process preceding the thesis.

I wish to thank Sun Microsystems, Inc. for donating the Enterprise 5000 servers on which this work is based. A fellowship from the National Science Foundation let me explore my own interests in my first years, and I have received additional indirect support from them in later years. Lawrence Livermore National Laboratory and the Defense Advanced Research Projects Administration have also contributed to my support.

Brent Chun developed much of the Myrinet AM-II protocol, which played a key role in enabling this work. Thanks are also due to the members of the Clumps and NOW groups at U. C. Berkeley for their encouragement and support. Rich Martin helped with the selection of applications and input parameters. Finally, I wish to acknowledge the efforts of David Gay and Randi Thomas in developing and tuning the 3-D FFT code. Ziqiang "Chon-Chon" Tang suggested the use of epoch numbers to improve the performance of the lock-free algorithm; the new form provides FIFO semantics.

The faculty here at Illinois have also been quite supportive in allowing me to finish my thesis concurrently with beginning my employment. David Daly, a student of Bill Sanders, contributed cycles on his Ultra 10 for the benefit of my performance comparison.

My two sons, Andrew Jia-Wei and Justin Jia-Yao, have often inspired my progress, and I have thoroughly enjoyed spending my mornings with them while in graduate school. Watching them grow has been an amazing experience. My parents and parents-in-law have also helped through the years, providing endless cooking, cleaning, and babysitting services as well as a generous helping of love while my wife and I pursued our graduate degrees. Finally, and most importantly, I thank my wife, Jennie, who has given selflessly of her time, energy, and love to help me through the process of obtaining my degree. I am forever grateful for her companionship and support.

Having mentioned but a few of the shoulders on which I have stood, I invite you to look upon my works, ye Mighty, and despair.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last few years have brought fruition to cluster-based computing, enabling the construction of scalable systems that simultaneously support both interactive and batch use by sequential and parallel jobs. A combination of engineering and economy has pushed the basic computing unit in these clusters from a uniprocessor workstation to a symmetric multiprocessor, or SMP. These newer systems, known as Clumps, offer potentially superior performance to applications capable of exploiting the tight coupling of resources within each SMP. For some applications, however, resource contention may degrade performance relative to a cluster of uniprocessors.

In this thesis, we develop an understanding of the issues for fast, user-level communication between processes in a Clump and for the broader problem of coupling levels of a hierarchical system within a single abstraction. We carefully engineer and tune a transport protocol for passing messages through shared memory, introducing in the process a new concurrent queue algorithm that obtains good performance on both dedicated and multiprogrammed machines. We then encapsulate this protocol in a uniform communication interface that transparently routes messages through the appropriate medium: a fast network between SMP's or a cache-coherent interconnect inside an SMP. We expose the problems involved with coupling protocols of disparate speed and present a solution that dynamically tunes our communication layer to the underlying architecture. Using both applications and benchmarks derived from the message-passing literature, we then measure the performance of our system. Finally, we construct a model of shared communication resources to explain our results. Through our communication layer, we illustrate the com-

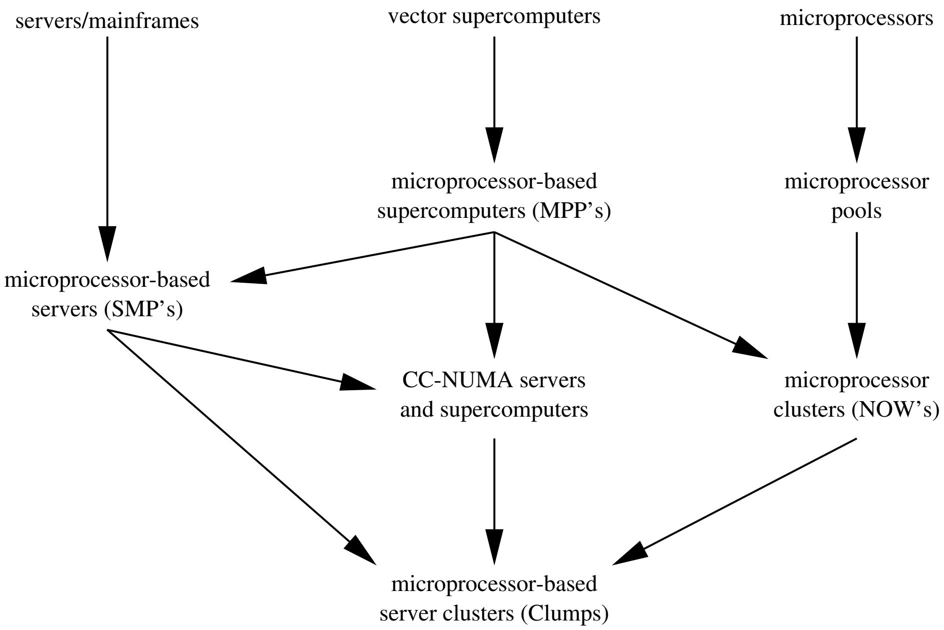

Figure 1.1: Clumps as a convergence of architectures.

plexity involved in managing multiple protocols in a hierarchical system, demonstrating the value of abstracting the hierarchy below the level addressed by application programmers.

1.1 Convergence of System Architectures

Clumps are the product of a convergence in high-performance computer architecture over the last two decades. The traditional classes of computers—supercomputers, mainframes, and personal computers—have gradually converged into Clumps. A diagram of these trends appears in Figure 1.1; the nodes represent architectures, and the arcs represent the transitions of applications and user communities over time.

The initial phase of the convergence, in which architectures converge to their microprocessor-based counterparts, is nearly complete. The growth of microprocessor performance has far exceeded those of mainframes and supercomputers, and the performance gap between these systems has been closing steadily for many years. With feature sizes constantly shrinking, microprocessors have been able to adopt features of their more powerful counterparts, such as wide, general-purpose register files and multi-level caches. The result has been an explosive growth in performance. In the same time frame, rapidly expanding and highly competitive markets have driven microprocessor costs lower and lower. Despite the existence of substantial bodies of legacy code, the attractiveness of the cost-performance ratio for microprocessors has drawn an ever-growing portion of the supercomputer and mainframe markets to their microprocessor-based counterparts.

A shift from vector machines onto massively parallel processors (MPP's) defined this transition for supercomputers. MPP's replaced the high-performance processors and shared memory systems designed specifically for the vector supercomputers with microprocessors and individual DRAM memories borrowed directly from workstations. MPP's then coupled these components through high-performance, custom interconnection networks. Although unable to compete on a per-processor basis, MPP's leveraged their low cost components to obtain supercomputer performance on a wide range of problems. The bankruptcy of Seymour Cray's last company, Cray Computer Corporation, in 1995, heralded the end of this phase of the convergence for supercomputers.[1]

While vector computers dwindled in number, a fairly large market for mainframes allowed them to remain an active part of many businesses. For newcomers to the mainframe market, however, symmetric multiprocessors (SMP's) became an increasingly attractive alternative. An SMP couples microprocessors to memory banks through a fast interconnect. Access times are uniform, or symmetric, between any processor and any memory location, and the interconnection hardware helps to maintain coherence between the processors' individual caches. SMP's typically run a single copy of an operating system and maintain a single system image, exposing an interface very similar to that of a workstation. By the early 1990's, SMP technology had matured to the point that IBM decided to shut down its bipolar work in favor of CMOS technology. Today, IBM's high-end platform, the S/390 Parallel Enterprise Server [IBM98b], is an SMP [IBM98a], although some marketing literature still touts it as a mainframe. Legacy codes thus represent the only remaining market for the original mainframes.

In the same time frame, research groups began to pool microprocessors into clusters, as indicated by the upper right arc in Figure 1.1. Linking computers together into a cluster has been a popular notion since the early days of mainframes, and clusters supporting job-level parallelism were offered as commercial products as early as the mid-1970's [Pfi98].

[1]Vector supercomputers have meanwhile become a hot architectural trend in Japan, with recent offerings from NEC, Fujitsu, and Hitachi. Vector computing may also be resurrected inside of microprocessors through recent processor-in-memory research [PAC+97].

By the late 1980's, workstations had become fairly common in research communities, and their owners began to build their own clusters. The first of these efforts were perhaps Sprite [DO87] and Condor [LLM88], which allowed the remote execution of UNIX jobs to reclaim idle cycles.

When the high cost and small market of supercomputing finally brought many of the MPP companies to bankruptcy in the mid-1990's, the choice of a replacement architecture divided the user community. One group, shown to the right in the figure, moved to networks of uniprocessor workstations, or NOW's. Spurred by the revolution in network technologies in the early 1990's, researchers had retargeted the advances in interconnection hardware on MPP's such as the Thinking Machines CM-5 and the Intel Paragon to the microprocessor pools [BLA+94, RLW94, BCF+95, BDF+95]. Coupled with similar advances in software [vECGS92, RLW94, LC95, SS95, Mar94, vEABB95], NOW's created the possibility of efficient medium- and fine-grained sharing amongst the nodes and promised a scalable supercomputer. The ensuing flurry of research activity [ACPtNT95] brought significant progress to cluster computing. Lightweight, user-level communication systems [PLC95, WBvE97] exposed the high performance available with the new networks. Background batch-processing [LL92] and gang-scheduling [Ous82] matured into job management systems capable of balancing both interactive and batch jobs across a cluster with a high degree of transparency [GPR+98]. Novel forms of decentralized job scheduling allowed clusters to maintain high-performance while time-sharing individual nodes [ADCM98]. Minor extensions to network hardware allowed support for fine-grained distributed shared memory [RLW94], and compilers began to address optimizing communication operations [KY96, YSP+98]. Tools for automatic network configuration were developed [MCSW97], and tools for system monitoring and fault detection are on the horizon [AP97]. With these advances, clusters became an attractive design point, both for pooled sequential demands and for parallel processing.

A second portion of the supercomputing community moved onto SMP's, as shown on the left in the figure. SMP's are an attractive platform for many applications, but they cannot scale quickly enough to keep up with the needs of high-performance computing. As Yeung argues, the cost of an SMP scales more rapidly than the number of processors, and the increased development costs must be amortized over a smaller market [Yeu98]. Early experience with SMP architectures had demonstrated their lack of scalability, leading researchers to explore more scalable approaches while retaining the notions of a single system image and a uniform address space. The result was a cache-coherent, non-uniform memory access (CC-NUMA) architecture, perhaps best illustrated by the Stanford DASH [LLG+90]. Like SMP's, CC-NUMA architectures provide hardware-based cache-coherence support for all memory locations. Unlike SMP's, however, the access time to a particular location can depend on the physical location of the memory in the machine. Accesses to memory close to a processor take less time than accesses to memory far from the processor. Industry quickly accepted the CC-NUMA approach, introducing a range of new platforms, including the Convex Exemplar [BA97], the Sequent Sting [LC96], the SGI Origin [LL97]. The remaining portion of the supercomputer community has moved to these systems, as shown in the center of the figure.

These three architectures—NOW's, SMP's, and CC-NUMA machines—currently dominate high-performance computing, but are slowly converging into Clumps. The driving force behind this final transition is the need for highly available systems. Today's global economy creates expectations of instantaneous service, and enterprise computing systems must remain available at all times. But the single system image that attracts users to SMP's and CC-NUMA architectures also presents a single point of failure: the operating system. Recent research [CRD+95, TBG+97] allows CC-NUMA architectures like the SGI Origin to partition themselves into independent sets of processors, but applications that make use of the cache-coherence hardware across partition boundaries are little more tolerant to faults than are applications on an unpartitioned machine [GC98]. The need for partitioning makes recent advances in clusters very attractive targets for integration. IBM recognized this fact when it started the move to CMOS technology and simultaneously began development of cluster technology. The result was the S/390 Parallel Sysplex, a Clump [NMCB97]. Thus, although the hardware of the future may be CC-NUMA, the lack of global cohesion produced by partitioning will require a programming style closer to that of Clumps than that of SMP's.

For cluster computing, Clumps offer several advantages over NOW's. First, they reduce management overheads for both software and hardware by reducing the number of boxes and external connections. Second, software costs may be less as well, since many packages are sold per-box rather than per-processor. Finally, Clumps couple processors above the level typically available with NOW's, allowing communication over the memory interconnect rather than the slower I/O bus. As we show in this thesis, realizing the performance improvement implied by this last feature requires some thought.

1.2 Significance of the Convergence

The convergence to Clumps reflects a broader trend towards complex hierarchies of data. Processors interconnect at many levels, and each level supports shared resources and provides a data path for communication. In terms of processor speed, the latency between the levels increases steadily, threatening to degrade application-level performance. The task of managing the motion of data through the hierarchy thus becomes more important over time. As this trend continues, we must expect to address issues of data affinity and motion explicitly, overriding the usual heuristics with platform- or application-specific strategies. In this section, we expand upon this argument and discuss our general strategy for addressing the problem.

A data hierarchy deepens in response to diverging performance amongst the levels. Individual processors must continue to track the exponential performance growth predicted by Joy's Law ($2^{\text{year}-1987}$ MIPS) to remain competitive, and must do so in spite of their interactions with system components that improve at much slower rates. Due to these differences in growth, the performance gap between two adjacent levels of the data hierarchy widens constantly. When a gap grows too wide, a new level is inserted to bridge the gap, providing faster access to a smaller data set than that provided by the level below. The memory hierarchy for microprocessors, for example, has grown from a simple memory and a few registers to a model that includes a large register set, an on-chip cache, an off-chip cache, and main memory.

Extending a hierarchy improves performance by taking advantage of spatial and temporal locality of access. Programs commonly access only a subset of their data at any point in time, and use the data in the subset many times before moving on to another subset. Each time the hierarchy expands, we rely on a program to exhibit locality at the new level, but no program does so perfectly. These imperfections in locality couple with unpredictability in access patterns to reduce performance across each level of the hierarchy. As these levels grow further apart, the losses grow.

A programmer can often improve performance by addressing the hierarchy in a program, changing the algorithms and data access patterns to improve locality and to give hints about future requirements. Programmers have traditionally exercised a great degree of explicit control over the lower levels of the data hierarchy—tapes, disks, and to some extent main memory. The upper levels, however, are managed implicitly by heuristics in both the hardware and the operating system: caching policies, coherence protocols, and virtual memory. Addressing these levels requires that a programmer understand the governing heuristics well enough to predict the outcome of changes to the program. The tradeoff between implicit and explicit methods of control generates some tension for programmers. No programmer wants to manage all data motion explicitly; even if the dynamic overhead of such management were eliminated, the cost in human effort would override the benefit to performance. Similarly, no programmer wants an inappropriate heuristic to prevent a program from obtaining satisfying performance. The proper balance between these extremes is debatable, and is further complicated by the possibility of control by intermediate agents such as a compiler or a runtime system. Growing latencies between levels of the data hierarchy, however, tend to shift the balance toward more explicit control.

Multiprocessing further tips the balance towards explicit management. Sequential programs executing on a multiprocessor compete with other programs at lower levels of the data hierarchy, and programs with multiple processes or threads must break the notion of locality when communicating. Although the appearance of low-end SMP's may shift this focus, current data hierarchies and management heuristics are tuned for sequential performance, favoring efficient access by a single processor over the efficient transmission of data between processors required for communication.

Despite the increasing importance of applying application-specific knowledge to data access, the notion of automatic control remains very attractive. We can compromise by retaining the heuristics for managing data in the default case yet allowing these defaults to be overridden when necessary. Ideally, this solution combines straightforward initial results with the possibility of incremental tuning. After completing an initial version of a program, the programmer—or compiler, or runtime system—can apply semantic information to eliminate performance bottlenecks by overriding heuristic decisions. This combination of methods requires a clear set of control abstractions that allows the introduction of explicit control at a fairly fine granularity.

Broadly speaking, the work in this thesis is not limited to Clumps, but rather attempts to address some of the issues involved in designing software environments on hierarchical systems. We build a thin abstraction that implicitly manages communication across two levels of the hierarchy and provides fast paths through each level; this abstraction is easily bypassed when a programmer wants explicit control. We examine the impact of bringing multiple levels of a hierarchy together in the context of a single interface—one

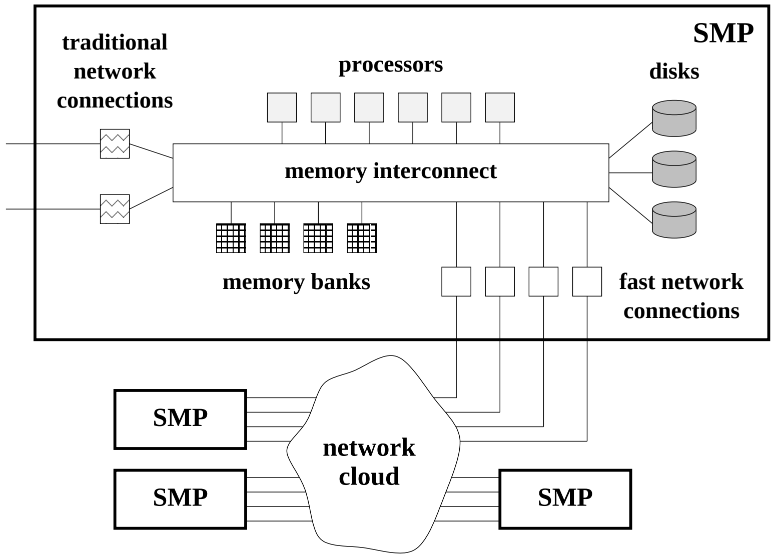

Figure 1.2: A prototypical Clump.

that is likely to be useful on the machines of the future. We establish the utility of explicit control over the movement of data between levels of the hierarchy. Finally, we study the issues involved in sharing multiple resources among user-level processes.

1.3 Addressing the Hierarchy

A Clump supports two communication paths through the data hierarchy, across a cache-coherent memory interconnect or through a fast network. In this section, we discuss the question central to the thesis: what abstraction best combines these two architectures for communication? In the previous section, we reasoned about the goals for such abstractions: to provide simple but effective default behavior and to establish clear methods for overriding the default to improve performance. We begin this section with a definition of the Clump architecture, then discuss previous systems in light of our requirements. We follow with a description of the approach used in this work, differentiating it at a high-level from past efforts and explaining how it meets the stated goals. We close with an explanation of the importance of our solution.

A prototypical Clump consists of a group of symmetric multiprocessors connected with a fast network, as shown in Figure 1.2. In each SMP, a backplane connects the processors to memory banks, disks, fast network connections, and traditional network connections.

All resources in the SMP are shared between the processors. The memory interconnect—a bus in most small to mid-range SMP's, or a switched or partially switched network in high-end models—helps to provide coherence between cached data. The fast network connections, each on a separate I/O bus, handle communication within the Clump. If present, the traditional network connections link the Clump to the outside world and provide a backup to the fast network within the Clump.

Effectively addressing the hierarchical nature of communication in a Clump requires a substantial effort. Clearly, many applications will require that a programmer recognize the hierarchy to some extent to obtain optimal performance, but an effective abstraction can relieve the programmer of the bulk of the workload. Several systems from the literature provide only a thin veil of synchronization and control functions. These systems leave nearly the full burden of complexity to the programmer, who must write threaded, shared memory code within each SMP and message-passing code between the SMP's.

A better approach abstracts communication into a single interface. Ideally, the abstraction produces reasonable performance for programs that ignore the hierarchy and allows a programmer to tune performance by selectively recognizing and handling hierarchical aspects of the application. The difficulty of designing and tuning a uniform interface is similar to that required to address the communication hierarchy directly in an application.

In either case, an effective abstraction requires careful engineering of the protocols and mechanisms for intelligently combining them. Individual protocols must expose the performance available at their level of the hierarchy, avoiding extraneous overhead and unnecessary functionality. The protocol for the cache-coherent interconnect must tackle problems of data layout and concurrent access to minimize the cost of coherence transactions. The protocol for the network must strike a balance between the time spent sending data across the relatively slow I/O bus and the time spent compressing or recovering data sent across the bus. Each process in an application operates independently, and a hierarchical abstraction must support interaction via either protocol regardless of a local process' decisions. Synchronization constraints imposed at a higher level, using another hierarchical abstraction, can simplify such support, but renders an abstraction less general and more prone to incorrect use. An abstraction must encapsulate the protocols in a way that maintains correct behavior under distributed control yet avoids adversely impacting the performance of local operations based on either protocol. A uniform interface provides a reusable hierarchical abstraction, reducing the necessary effort for every application that builds on it. In contrast, application-specific abstractions can be reused only with very similar applications.

Two options present themselves: extend shared memory across the network, or extend message-passing across the memory interconnect. The uniform interface commonly studied in relation to Clumps is that of distributed shared memory, or DSM. A DSM interface provides a single, global address space abstraction and manages cache-coherence issues for the programmer. The DSM runtime layer performs all inter-SMP communication, sending messages across the fast network as necessary to maintain data consistency. This implicit communication can obscure the relationship between the performance of hardware and that of software, and a programmer must have a reasonably deep understanding of the runtime system in order to tune performance. Furthermore, the importance of data layout to performance in a DSM system can couple the performance of otherwise only distantly related portions of the code: an optimal layout in one phase of an application can result in very poor performance in a second phase. We return to these comparative evaluations in Chapter 10, after describing our own work in detail.

We instead focus on a user-level message-passing interface, and in particular on an interface for function shipping. Such an interface provides methods to invoke functions in other processes; blocks of data can be shipped along with an invocation. Two interpretations of this approach are possible: active messages focus on the operations used for data transport, while distributed objects focus on the ability to build high-level abstractions. The two interpretations are fundamentally equivalent, but the object model requires additional compiler support. We provide only for active messages.

A message-passing interface closely couples the performance of an application to that of the underlying hardware. Invoking a remote function takes a fair amount of time regardless of the location of the receiving process. A programmer must attempt to hide message latency by overlapping it with computation within the process. Improving the performance of a program merely requires that the programmer refine this cost model, recognizing that functions invoked in a process on another SMP take longer than functions invoked in a process within the same SMP. Locality information is readily available, and the programmer need tune only those portions of an application that result in performance bottlenecks. The performance of any phase of an application depends more or less entirely on the algorithmic form of that phase, allowing distinct phases to be optimized independently.

Building software to pass messages through cache-coherent shared memory may seem counterintuitive. Such systems are fairly common, however, and play an important role on SMP's regardless of the view provided to an application programmer. Messages are the standard mechanism for interprocess communication and have been central in this regard to the development of modern operating systems such as Mach [YTR+87]. The implicit synchronization encapsulated in the message abstraction simplifies the construction of otherwise asynchronous data transfers and provides an efficient mechanism for serializing operations on complex data structures. Messages are also a generally accepted tool for parallel programming and often serve as a tool for building more complex abstractions. Typical DSM systems make use of messages, for example, to support the single address space abstraction between distributed memories [SGT96, SDH+97] and to optimize protocols within an SMP [SGA98].

In summary, we address hierarchical communication by providing a simple, uniform interface for passing messages. The layer combines two highly optimized, user-level protocols and exposes the performance of both with very little overhead. This layer provides a fundamental component of any runtime system for Clumps, but can also serve as the programming interface to the system. An application can obtain a reasonable level of performance without addressing the hierarchy, but the path to improved performance through the use of locality information is clear. Finally, applications built with this interface migrate readily between Clump configurations.

1.4 Technical Issues for Clumps

We now explore the relationship between the structure of a Clump and the performance realized by programs running on such a machine. As with any modern computer, issues of spatial and temporal locality in data access play important roles in performance. In this section, we focus instead on issues specific to the Clump architecture, particularly on the effects of sharing resources between processors within each SMP. By eliminating the high overheads commonly associated with accessing a peripheral device, user-level communication exposes these underlying performance phenomena. In this section, we describe the issues and discuss the utility of our communication layer in exposing their effect on performance.

A Clump extends the environment from a cluster of uniprocessors in two important ways for fast communication: by allowing multiple processors and by allowing multiple network devices. A uniprocessor can technically support multiple devices, but the idea of installing more than one network card in a machine with only one I/O bus has prevented the high-performance communication community from seriously considering this possibility.

As mentioned earlier, the processors in an SMP share attached devices. Traditionally, the operating system provides centralized management, scheduling the processors' demands onto the available resources. The high-performance communication systems commonly used in clusters, however, bypass the operating system to improve performance. Multiple processors can thus simultaneously access a network device, and we must virtualize these devices to prevent processes from interfering with one another. The reduction in overhead also highlights three issues specific to interactions between the processes. Each of these phenomena presents benefits and penalties for performance, with the exact balance depending both on the specifics of the Clump platform and on the application.

Tight coupling, the first issue, pertains to the merging of multiple processors' data hierarchies at the level of the memory interconnect. This coupling acts effectively as a crossbar linking all processors in an SMP to all devices, moving devices that might otherwise reside across the fast network from a processor upwards in that processor's data hierarchy. Processors hence perceive lower access latency and higher bandwidth to these devices. Not all devices benefit from this change, however; many devices still reside across the network, in distinct SMP's. Determining the appropriate method by which to access a particular resource incurs overhead unnecessary in a single-protocol system.

The second issue, pooling, addresses the aggregation of demands and of devices within an SMP. Demand pooling refers to aggregation of the processors' resource demands. Pooled demands can be less bursty than the individual demands, but contention among processors can also reduce the efficiency of the resources. Device pooling refers to the aggregation of devices into a single virtual device, providing a view of multiple devices within an SMP as a single resource. Pooled devices offer potentially higher throughput, but the overhead of spreading demands amongst devices may increase latency or even decrease throughput in some cases.

Fractional scaling, the last issue, pertains to the incremental scalability of the architecture. If we pool the devices of some type in an SMP into a single, more powerful virtual device, we can improve upon the pooled power with better than integral precision. We need not, for example, double the power of the pooled device, but might instead choose to boost it by a fourth or a third. Of course, we must install an integral number of physical devices, but the pooled device permits a greater degree of freedom in matching the capabilities of the system to the needs of a particular workload.

The Clumps architecture covers a broad spectrum of systems—everything between a NOW and a single SMP—and a complex relationship links the issues just discussed to performance within this space. In this thesis, we measure and model these phenomena to provide a framework for understanding of the relationships between the underlying hardware, the programming model, and application performance.

1.5 Contributions of the Thesis

In this section, we briefly describe the contributions embodied in this thesis. They include multi-protocol design, system construction, measurement techniques, performance evaluation, hardware optimization, and application modeling.

We identify the important issues in the design of a uniform interface for communication across both cache-coherent interconnects and fast networks. These issues include careful engineering of data structures and operations to minimize cache-coherence transactions; addressing concurrent access to message queues with an algorithm that performs well on both dedicated and multiprogrammed machines; and coupling the underlying protocols in a way that exposes the performance available from each.

We construct a fully operational example of a uniform communication interface and tune the system to the underlying architecture. We carefully engineer the data structures and operations used to communicate across the cache-coherent interconnect and explain the rationale behind them. Our effort to minimize cache-coherence transactions demonstrates the level of effort required for effective shared memory design. We present a concurrent message queue algorithm that obtains superior application performance on both dedicated and multiprogrammed SMP's. Our algorithm is robust under contention yet incurs overhead competitive with the fastest spin locks. We explain the machine-level interactions that lead to superior performance and comment on the value of our design approach. Finally, we develop and tune a parametrized, adaptive polling strategy that dynamically adjusts to the underlying architecture without incurring significant overhead. The adaptive strategy allows a single application binary to perform well on any Clump architecture, from an SMP to a NOW.

We outline a methodology for measuring the performance of a communication layer on a Clump, commenting on the difficulty of obtaining accurate results due to execution dependencies propagated through the hardware and the operating system. We explain a range of microbenchmarks and tools for measuring those parameters of an architecture critical to communication performance: memory latency and bandwidth, cache-to-cache transfer cost, NIC memory access time, and synchronization primitive delay. Drawing on the message-passing literature, we set forth a new suite of benchmarks for passing messages through shared memory. This suite includes microbenchmarks for exploring performance at the extremes of high and low contention, and a set of applications with distinct communication patterns for investigating performance at the application level. For each application, we report both the memory and communication requirements. Finally, we outline a methodology for measuring and reporting application performance in a multiprogrammed environment that more accurately reflects expected performance on a time-shared system than do previous approaches.

We apply the microbenchmarks and benchmark suites to our experimental system, consisting of Sun Enterprise 5000's connected with Myrinet. We first measure and report the important parameters for this architecture. After tuning the shared memory protocol to the experimental platform, we employ our benchmark suite to compare the performance of our message queue algorithm with an array of alternative algorithms, demonstrating competitive or superior performance on both dedicated and multiprogrammed SMP's. We measure application execution times and compare them with times on similarly constructed NOW's. We tune the polling strategy parameters using data from an SMP, a NOW, and a Clump, then measure microbenchmark and application performance on the Clump to evaluate the impact of transparent protocol integration. We compare application performance on a Clump to performance on a comparable NOW, revealing the relationship between the Clump architecture, the programming model, and the communication pattern.

We break down the costs of shared memory message-passing with our system and explain those costs in terms of the cost of cache-coherence transactions in the underlying hardware. We forecast the effect of current architectural trends on future message-passing performance, then suggest an approach to improving that performance through minimal changes to a commercial SMP's instruction set and interconnect hardware.

We analyze the tradeoffs involved with shared communication resources and present a simple model that qualitatively predicts performance. Casting our experimental platform into the abstract form used in the model, we explain the relationship between our empirical results and the phenomena predicted by the model. We comment on the limitations of our model and on the difficulty of accurately predicting performance on Clumps.

1.6 Outline

The remainder of the thesis divides roughly into four parts: a description of the uniform communication architecture, a presentation of the performance achieved with that architecture, a construction of an analytic model to explain the observed performance, and a discussion of this work in relation to past and future efforts.

The first part includes three chapters. Chapter 2 introduces the uniform communication architecture, defining common terminology in a brief overview of the system, then summarizes the Active Messages-II message-passing specification and the relevant portions of the interface. Within this framework, the chapter lays out the design challenges for multi-protocol communication. Chapter 3 details the design of the shared memory protocol, our implementation of active messages over a cache-coherent memory interconnect. The chapter begins by illustrating the data layout and discussing the rationale behind the structures. An example of ping-pong communication follows, defining the operations used by the protocol in initialization and communication. Next, the chapter describes a lock-free algorithm that provides superior performance for managing concurrent message queues. After commenting on the hardware synchronization primitives necessary to support the algorithm, we construct two proofs regarding its behavior. We conclude with a discussion of the queue semantics supported by the algorithm and the advantages over alternative approaches. In Chapter 4, we address the issues that arise when combining two protocols with disparate costs. Due to the relatively high cost of checking for the arrival of network messages, our multi-protocol layer dynamically adapts its polling strategy to observed traffic patterns. The chapter discusses this problem in more detail, then presents a mathematical framework for managing the polling interactions.

The second part also consists of three chapters, and is analogous to the first in construction. Chapter 5 introduces the methodology and experiments used to measure performance, including an array of microbenchmarks and a set of applications. The chapter also presents our experimental platform and reports parameters of the memory hierarchy and synchronization primitive timings. Chapter 6 discusses the performance of the shared memory protocol in isolation, first tuning the protocol for optimal performance on our experimental platform. The chapter next describes an array of alternative algorithms for managing concurrent queues, and the bulk of the chapter is devoted to measuring the performance of these alternatives at the extremes of zero and maximal contention and with a number of phase-structured (or bulk synchronous) applications. We follow with a comparison of the algorithms on multiprogrammed systems. These measurements demonstrate the superiority of our lock-free algorithm. We conclude with a detailed breakdown of overhead with the lock-free algorithm, which allows us to predict the impact of architectural trends and to suggest hardware support for improving the performance of messages passed through shared memory. In Chapter 7, we investigate the performance of the multi-protocol layer across entire Clumps. We begin with an empirical investigation of polling strategies, selecting parameter values to optimize performance across applications. We then report both microbenchmark and application-level results for the final communication layer. The chapter evaluates the impact of multi-protocol communication on performance and compares performance on a Clump with performance on a NOW. The chapter concludes with a discussion of the impact of aspects of the hardware and the programming model on overall performance.

In the third part, comprised solely of Chapter 8, we present an analytical model of shared resources, focusing on the use of multiple network interface cards within each SMP. After explaining the underlying queueing theory, we extend the model to include aspects of the phase-structured programming paradigm and qualitatively explain the results seen in Chapter 7 and the performance issues discussed in this chapter. We close with a discussion of the limitations of the model and comment on the difficulty of predicting quantitative performance.

The fourth part combines a discussion of related work in Chapter 9 with our conclusions and suggestions for future study in Chapter 10. After a brief history of shared memory message-passing and concurrent message queue algorithms, Chapter 9 describes other multi-protocol systems. Programming models are covered next, beginning with the large body of work on distributed shared memory and moving on to compiler-based and non-uniform approaches. We last address other efforts in concurrent algorithm performance evaluation, focusing on the problem of developing a metric for multiprogramming. In describing other work, we relate it to the issues uncovered by this thesis and discuss how the work complements our own. We conclude the chapter with descriptions of a few less directly related efforts. Chapter 10 starts with our conclusions from the thesis, advocating the uniform message-passing approach and the components that enable its success: the lock-free algorithm and the adaptive polling strategy. The chapter then offers some ideas about programming models for Clumps and closes with commentary on the importance of cluster architectures.

Chapter 2

A Uniform Communication Architecture

In this chapter, we outline key aspects of our multi-protocol, active message communication layer. We begin with an overview of the architecture, describing the basic mechanisms for message transport. An introduction to the active message model of communication follows the overview. Within this framework, we then outline the design challenges for multi-protocol communication. The following two chapters present our solution in full detail: Chapter 3 focuses on the construction of the shared memory protocol, and Chapter 4 discusses the full, multi-protocol implementation.

2.1 Architecture Overview

The multi-protocol layer provides a uniform interface for communicating between processes distributed across a Clump. A programmer views communication as a black box, ignoring the relative locations of destination processes when developing an application. The layer then transparently de-multiplexes outgoing messages to the appropriate protocols and multiplexes incoming messages from the protocols at runtime, as depicted in Figure 2.1. Each protocol is highly tuned to the characteristics of its transport medium, and the layer further optimizes the multiplexing and demultiplexing functionality. The programmer thus avoids the need to directly address the hierarchical nature of the Clump with multiple mechanisms, runtime support, and tuning for each distributed concept used in an application.

In building the multi-protocol layer, we incorporated an existing active message

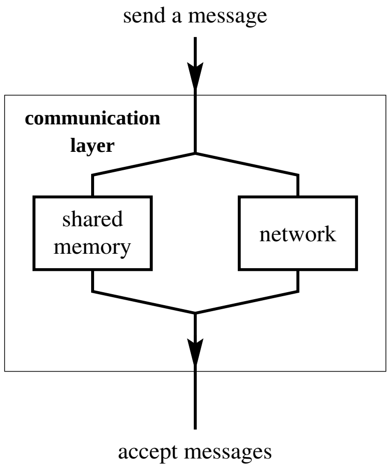

Figure 2.1: Abstract message-passing communication layer. The communication software transparently routes messages through either shared memory or through the network and pulls messages from both protocols when polled.

implementation [CMC98] as the primary component of the network protocol, extending the implementation with a shared memory protocol and with support for combining the two protocols. Messages destined for other SMP's, or *remote* messages, travel via the network protocol. A sending process, or sender, deposits the message into an outgoing queue, and the processor on the network interface card (NIC) pushes the message onto the wire. When a NIC on the destination SMP receives the message, the NIC deposits the message into an incoming queue for the destination process. *Local* messages—those destined for processes in the same SMP as the sender—travel via the shared memory protocol. The sender deposits the message directly into a shared memory queue associated with the destination process; no external agent need handle the data in this case.

After a message arrives in one of the queues, it must wait for the destination process to receive it. Reception is synchronous with respect to the application: a process must call into the communication layer to receive messages. The interface includes an explicit poll operation for this purpose, but the layer also checks for arrival when sending a message. During a poll, the process receives any messages that have arrived at either the incoming network queue or the shared memory queue. If several messages are present in the queues, the process accepts as many as possible, bounded by a predefined limit. When a process receives a local message, the SMP's cache-coherence mechanism moves the data across the memory interconnect.

2.2 Introduction to Active Messages

The active message model of communication is well-known in the parallel programming community and is commonly among the fastest methods of communication available for a given platform [vECGS92, Mar94, TM94, PLC95, SS95, vEABB95, WBvE97]. Each active message contains a reference to a handler routine. When a message is received, the communication layer passes the data to the handler referenced by the message, typically as formal parameters. The association between message arrival and the execution of a particular block of code is the origin of the term "active message." Our system implements a relatively recent specification of active messages known as AM-II [MC96].

2.2.1 Communication abstraction

The active message model defined by AM-II is similar to a remote procedure call (RPC) mechanism in which each *communication endpoint* acts as both client and server to other endpoints. Each *request* message sent from one endpoint to another must be matched by a *reply* message sent in the opposite direction; AM-II guarantees that each message's handler is executed at most once. Messages can be of two types: a *short message* carries up to eight 32-bit arguments; a *bulk data transfer* extends a short message with a block of up to 8 kB of data. A 32-bit *tag* governs access rights: a sender must know an endpoint's tag before sending any messages to that endpoint. These tags provide a reasonable level of protection against both inadvertently misdirected and malicious messages.

The main differences between active messages and RPC are twofold: asynchronous execution and application-defined return semantics. RPC invocations execute synchronously: the calling process blocks at the point of invocation until remote execution completes and the return value arrives. RPC admits only one interpretation for the data returned, as a temporary object of the static procedure type. This object can then be cast or modified as desired. In contrast, active messages allow asynchronous execution: a remote procedure's invocation can occur at a separate location in the code from its return. Both request and reply messages are active, allowing a process to redefine return semantics.

As mentioned earlier, active messages are not fully asynchronous. Messages are received only at certain points in a program, when sending a message or when polling for message arrival. The reason for this approach is partly historical: obtaining acceptable performance demanded that an implementation avoid the high cost of interrupting a processor and performing multiple context switches to receive a message. However, preventing the invocation of handler functions at arbitrary points in a program substantially simplifies a program's data structures, which must otherwise permit concurrent operations. Isolating concurrent access through synchronous reception is also an effective technique on SMP's. If the low-cost interrupt solutions in the research literature ever migrate into commercial hardware, their advantage for active message communication will be primarily to drain messages from network hardware into queues similar to those defined for our shared memory protocol.

2.2.2 High-performance emphasis

The focus on high performance in active message implementations separates these efforts from many traditional communication layers. Towards this end, active messages shift work out of the critical path for sending and receiving messages and into less common operations such as initialization. Removing operating system involvement in communication is the most important aspect of this shift. A process accesses hardware communication resources directly when sending an active message, thereby avoiding context switches and kernel traps. To permit such access, an implementation handles most protection and security issues during initialization.

Active message layers use preprocessing and function splitting to further improve performance. Before communicating, a process defines its handler functionality and notifies the active message layer of all potential message destinations. These steps, which typically involve the operating system, allow an implementation to preprocess communication data and to reduce the time required to send a message. Function splitting expands the communication interface to include distinct send operations for different message lengths. For example, long and variable-length messages require more overhead than do short, fixed-length messages, hence active message implementations provide separate operations, optimized for each type.

2.2.3 Network interface virtualization

In early active message implementations, direct access to the network hardware implied dedicated use of the system by a single job. The Thinking Machines CM-5, a commercial MPP with global operating system support, circumvented this implication by

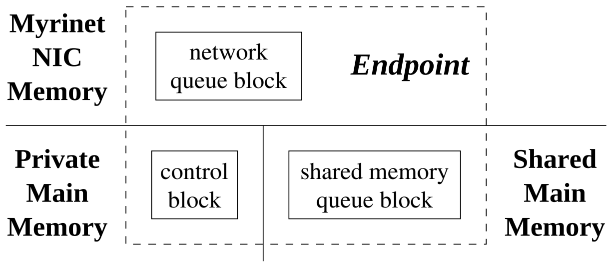

Figure 2.2: Block diagram of an AM-II endpoint. The control block resides in private main memory, the network queue block resides on a network interface card (NIC), and the shared memory queue block resides in shared main memory. Other processes access only the shared memory queue block.

including network state as part of an application's context. During a global context switch, the operating system drained the network of messages from the most recently scheduled job and reinserted the next job's messages into the network before starting it. On modern clusters, the software advances that allow simultaneous interactive and parallel execution also eliminate the notion of a global scheduler and prohibit the view that a single job occupies the entire system for a significant period of time. High-performance communication research has thus begun to address more general solutions.

The AM-II specification targets time-shared, communicating processes spread over a system area network (SAN)—a machine room, a building, or a campus. AM-II improves on earlier efforts by providing the functionality required for general-purpose distributed programming while retaining the performance potential of the earlier work. The abstraction central to this improvement is the communication endpoint mentioned earlier, a virtualization of the physical NIC [CMC98]. Many endpoints can be associated with one NIC, with each retaining simultaneous direct and protected access in the common case. The communication layer provides a reasonable degree of fairness between all actively communicating endpoints that share a NIC. When an endpoint is idle, AM-II support in the operating system reclaims its physical resources for use by other endpoints, moving any remaining data into main memory. If main memory becomes scarce, the conventional virtual memory system can then back the entire endpoint onto disk.

The multi-protocol representation of an endpoint is a natural extension of that developed for the Myrinet implementation. As shown in Figure 2.2, an endpoint breaks into three blocks, one for control information and one for each protocol's queues. The shared memory queue block, which contains the local message queue, resides in shared main memory to allow access by other processes within the SMP. The virtual aspect of an endpoint is thus maintained implicitly: the shared section represents a network device operating within the SMP.

As a part of virtualization, an AM-II implementation assigns a unique, physical name to each endpoint. This name provides enough information to locate the endpoint's NIC within the SAN and to restore an idle endpoint's network queue block from main memory. In the multi-protocol implementation, we extend the name to identify the endpoint's SMP, thereby enabling its use in protocol selection when sending a message.

2.2.4 Process structure

In this thesis, a parallel application creates one process per physical processor and associates each process with a single communication endpoint. We use multiple processes within each SMP rather than multiple threads to provide a closer match to the traditional message-passing model and to maintain address space protection boundaries. The resulting applications are also more easily ported between various Clump configurations, as they use no thread-specific techniques.

A threaded model can provide fairly significant advantages, however, including reduced switching and copying overheads. When competing with other programs, for example, switching between threads is less expensive than switching between contexts. Data sent between threads in the same address space might also permit fewer memory copy operations. Technically, data can be transferred implicitly by synchronizing the exchange of ownership with a short message. However, issues of data alignment and distribution reduce the practical value of such transfers. Further, the application must be aware of the machine hierarchy to take advantage of implicit transfer. Single-copy transfers are more promising, but require that data sent to another thread be left untouched until received (*i.e.*, copied). In either case, short messages remain an important component of the system.

2.3 Multi-Protocol Issues

In this section, we describe the design challenges for multi-protocol communication in terms of the framework defined in the previous sections. The thesis focuses on two aspects of multi-protocol communication: the development of a high-performance message-passing protocol over a cache-coherent memory interconnect, and the integration of that protocol with an optimized network protocol to provide an effective communication layer for Clumps. This work is quite timely in raising the issues of integrating high-performance networks and shared memory. Industry has recently begun to incorporate the ideas developed in the last five years of high-performance communication research into their product lines. The most visible of these efforts is the Virtual Interface Architecture (VIA) Specification [VIA97] authored jointly by Compaq, Intel, and Microsoft. Using components from several academic research efforts, VIA attempts to standardize high-performance, user-level communication; but in focusing on clusters of uniprocessor, personal computers, it neglects to consider the integration of such an interface with communication between processes in an SMP.

Recall the abstract layer depicted in Figure 2.1. Given the existence of a network protocol, the construction of a such a layer poses three interesting problems. The first challenge, which occurs at the send side of the figure, is to develop an efficient send operation for local messages. The endpoint data structures must be laid out and tuned to maximize performance through the de-multiplexing switch and the shared memory protocol. Remote messages also pass through the switch, but are less sensitive to added overhead. The second challenge, developing an effective, concurrent queue algorithm, falls within the boundary of the shared memory protocol. Local messages must perform well on a wide range of platforms, from dedicated Clumps to multiprogrammed SMP's. The third challenge, which occurs at the receive side of the figure, is to develop a strategy that minimizes the impact of polling each protocol upon messages passed through the other. We discuss these problems in the order described.

2.3.1 Data layout

The data structures used in a multi-protocol active message layer must be designed with several factors in mind. First, they must support a short critical path for communication. Second, they must minimize the number of memory transactions, as the cost of such transactions can easily dominate the time to send a local message. Third, they must employ design parameters that result in near-optimal performance on a range of Clumps.

Supporting a short critical path for communication requires that operations and the data structures be developed concurrently. Such coordination is standard practice, but can imply additional effort for parallel systems due to the complex relationships between concurrent access, data layout, and cache behavior.

Local messages pass through an endpoint's shared memory region and move between processors via cache-coherence transactions. As the cost of such transactions is fairly high, the data structures in our implementation must be designed to eliminate them whenever possible. Reducing the impact of memory transactions for a parallel system requires a fair amount of effort. If a structure stores too little data on a cache line, operations on the structure may require many transactions to obtain the necessary data, an effect known as fragmentation. On the other hand, if a structure merges too much rarely related data into common cache lines, operations on the structure may cause the memory system to thrash when another processor competes for unrelated data, an effect known as false sharing. A programmer must strike the proper balance between the extremes of delivering too little data with each transaction and causing thrashing by claiming unnecessary data. As shared memory applications must often employ this type of optimization, many techniques already exist. Application of these techniques, however, relies on experience and intuition, and discovering the proper layout requires a process of trial and error.

In addition to cache alignment issues, we must tune the length of the shared memory queue. A short queue may not adequately buffer incoming messages between poll operations, particularly when a process is descheduled on a multiprogrammed machine. A long queue, on the other hand, may pollute a program's data cache with communication data, or require an unacceptable amount of memory. The full queue timeout—the period that a sender waits at a full queue before giving up and yielding its processor—is closely related to queue length. A fast timeout yields too often and incurs extra switching penalties. A slow timeout wastes valuable cycles waiting for a descheduled process to make space in its queue. Appropriate values for these parameters depend to some degree on the particular application being considered, but are primarily defined by underlying hardware and operating system parameters.

Although none of these tasks is particularly hard, they are all time-intensive. Performing this tuning once for the communication layer is easier than evaluating the same problems repeatedly for each new application, as is otherwise necessary.

2.3.2 Concurrent access

An endpoint's local message queue is unique. Multiple processes can insert messages into this queue concurrently, and the enqueue operation must be carefully designed to ensure that concurrent operations occur atomically with respect to one another. Concurrent access also increases the importance of the queue block layout, as several processors' caches may compete for the data at any point in time. Most shared memory message-passing systems in the literature sidestep the difficulty of efficient concurrent access through the use of separate resources for each sender-receiver pair, but such an approach can hurt performance in a well-tuned, user-level communication layer. Concurrent queues provide superior scalability in both time and space.

The overhead of polling additional queue structures is roughly the same as the overhead of concurrent access on mid-range SMP's. As one adds processors, polling overhead grows linearly, whereas the cost of concurrent access increases only slightly with most communication patterns. If many-to-one communication is common, adding processors can result in significantly more contention, but the receiver's message handling rate dominates the effect of the increased insertion contention in such a case.

In terms of memory requirements, concurrent queues scale linearly with the number of intercommunicating processes in an SMP, as one queue is created for each process. Pairwise queues scale quadratically and hence require substantial amounts of memory for larger SMP's. If we instead divide a fixed amount of memory among the necessary queues, the resulting shortening of the queues leads to increased frequency of overflows and reduces performance on multiprogrammed systems. These problems become most apparent when we report performance data in Chapter 6.

Developing a good concurrent queue algorithm is not easy, however. The literature offers a myriad of possibilities, but little information as to the algorithms' performance with message-passing. The challenge is to develop a concurrent queue algorithm that obtains good multiprogramming performance yet is also competitive with the fastest algorithms on a dedicated machine.

2.3.3 Adaptive polling

Fast communication layers typically poll for incoming messages when sending a message in order to remain responsive to incoming traffic. In our multi-protocol system, polling for remote messages requires roughly an order of magnitude longer than polling for local messages. The difference lies in the location of the data. The shared memory queue resides in main memory, and polling information can be cached close to the processor. The network queue resides in uncacheable memory on the NIC, and each poll operation must pull the information across the I/O bus. Due to the disparate access times, a straightforward approach to polling results in unsatisfactory local message performance.

Previous systems have addressed this issue by polling the slower protocol less frequently [FGKT97]. This approach allows a static tradeoff between the protocols. If polling of the slower protocol is too frequent, the faster protocol continues to suffer a performance penalty. However, if polling of the slower protocol is too infrequent, it suffers a similar penalty. As an application based on our multi-protocol layer may execute on a NOW or an SMP, neither case is very attractive.

In our system, we employ a strategy that adapts dynamically in response to recent traffic. When local traffic dominates incoming messages, the strategy examines the network only rarely. When remote traffic is common, the strategy checks the network during every poll. Our strategy adapts naturally to the specifics of the underlying platform and can also enhance the benefit of separating local and remote traffic when tuning an application.

2.4 Summary

The multi-protocol layer provides a uniform interface for communicating between processes distributed across a Clump, rendering the hierarchical nature of the Clump transparent to an application programmer. A multi-protocol layer poses three challenges to development: an efficient send operation, an effective concurrent queue algorithm, a low-overhead integration approach. The shared memory protocol is particularly sensitive to unnecessary overhead, demanding careful management of coherence transactions through data layout and algorithmic manipulation of communication operations. A concurrent message queue forms the central component of the shared memory protocol. The algorithms used to control access to the queue must be robust enough to maintain high levels of performance under multiprogramming but lean enough to be competitive on dedicated machines. Integrating the two protocols with minimal overhead presents the last challenge, but low overhead in an environment in which both protocols are active is not adequate. Merging the two protocols into a single interface must also avoid penalizing performance on a more general Clump architecture: the presence of multi-protocol support must degrade neither network communication performance on a NOW nor shared memory communication performance on an SMP.

In the next chapter, we describe our solutions to the first two challenges, providing detailed information on both the data structures and the operations used to optimize the shared memory protocol. We also present a lock-free algorithm that offers a robust approach to managing a concurrent message queue without the high overhead incurred by many alternative algorithms. In Chapter 4, we address the final challenge, integration of the shared memory protocol and the network protocol, with an adaptive approach to polling.

Chapter 3

The Shared Memory Protocol

In this chapter, we describe the design of the shared memory protocol, addressing the challenges of data layout and concurrent queue management. We begin by defining the data structures, which illustrate the optimization techniques discussed in the previous chapter. An example of ping-pong-style communication follows, describing the operations performed by the layer to construct, to use, and to destroy channels for communication. We next introduce our solution for managing concurrent message queues—an algorithm that provides performance superior to traditional approaches—and discuss the necessary hardware synchronization primitives and expected performance. After proving a few properties of our algorithm, we conclude with a few comments on the queue semantics that it provides and a brief comparison with aspects of other approaches.

3.1 Data Structures

Local messages pass through an endpoint's shared memory region and move between processors via cache-coherence transactions. The data structures used in these operations—the control and shared memory queue blocks in an endpoint—must hence pay careful attention to how the data are laid out in relation to cache boundaries. At the same time, the structures must reflect the philosophy of finding the minimal critical path for sending and receiving messages through preprocessing of information and function splitting. This section describes the data structures used in our implementation, briefly relating their purpose and illustrating the techniques used to eliminate false sharing and to re-

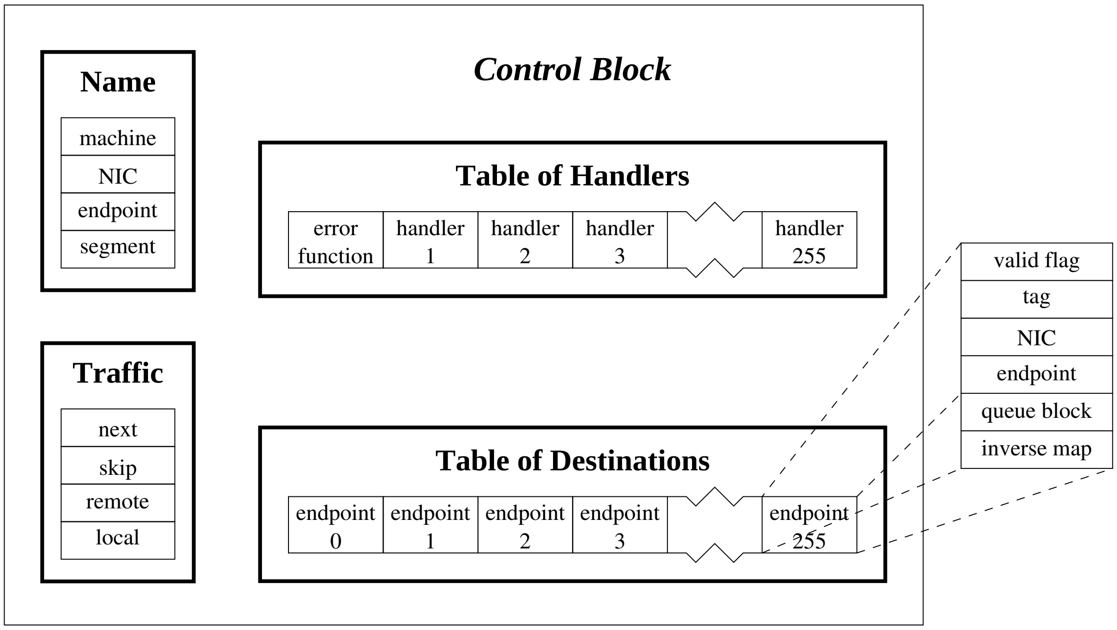

Figure 3.1: Block diagram of a control block. The control block defines an endpoint's physical name, handler functionality, and message destinations. The block also maintains traffic statistics for adaptive polling.

duce the number of coherence transactions. The exact use of the individual fields becomes apparent in the next two sections on operations.

3.1.1 The control block

An endpoint's control block holds information used by the AM-II library functions, as depicted in Figure 3.1. The control block also maintains information about network flow control and caches for network data, but local message operations use only the parts shown.

The name structure represents the physical endpoint name assigned at creation time by the Active Message layer. These names, which are opaque to higher levels of software, provide enough information to find endpoints within the network. The first field holds a machine identifier that uniquely identifies the computer on which the endpoint was created and allows other endpoints to select the appropriate protocol for communication. Our implementation uses the 32-bit value returned by `gethostid()` as the identifier for a given machine. The next two fields locate an endpoint within the network. The NIC field identifies the NIC associated with the endpoint, and any NIC can translate this information into a route. The endpoint number distinguishes between endpoints serviced by the destination NIC. The last field holds an identifier for the segment in which an endpoint's shared memory queue block resides and allows a process within the same computer to map the queue block into its address space.

The traffic structure maintains estimates of the frequency of local and remote traffic for adaptive polling purposes. Adaptive polling reduces the impact of remote message poll operations on local message performance, as demonstrated in Chapter 7. Two of the fields, remote and local, maintain exponential moving averages of arrival frequency. The skip field holds the total number of calls to the poll operation between remote polls, and the last field, next, holds the actual number of calls remaining.

The table of handlers is simply an array of function pointers indexed by default from 0 to 255. The value 0 is a special case and is used by the Active Message layer to return messages to the sender in the case of a network failure or a security exception.

The table of destinations translates the short integer names used in communication operations into an optimized form of the physical endpoint name. The NIC identifier and endpoint number are unmodified, but the machine and segment identifiers are replaced with a pair of shared memory queue block pointers. For a remote endpoint, these pointers are set to NULL. For a local endpoint, each pointer refers to the queue block of one endpoint in the address space of the process that owns the other. Each destination structure also contains a flag to indicate the validity of the other data in the structure and a copy of the destination endpoint tag for checking access rights.

3.1.2 The shared memory queue block

The shared memory queue block, our extension to the endpoint abstraction, holds local message queues in a System V shared memory segment to allow access by multiple processes within an SMP.[1] A diagram of the shared memory queue block appears in Figure 3.2. A copy of the endpoint tag is used for access control, while two queue structures

[1]The default parameters for System V IPC can be fairly restrictive. In the case of Solaris, for example, a process can map only five segments of at most 1 MB simultaneously. Fortunately, these parameters are readily changed (in Solaris) by appending the following magical lines to the `/etc/system` file on each machine and rebooting:

```
set shmsys:shminfo_shmmax=268435456
set shmsys:shminfo_shmmin=8192
set shmsys:shminfo_shmmni=65536
set shmsys:shminfo_shmseg=8192
```

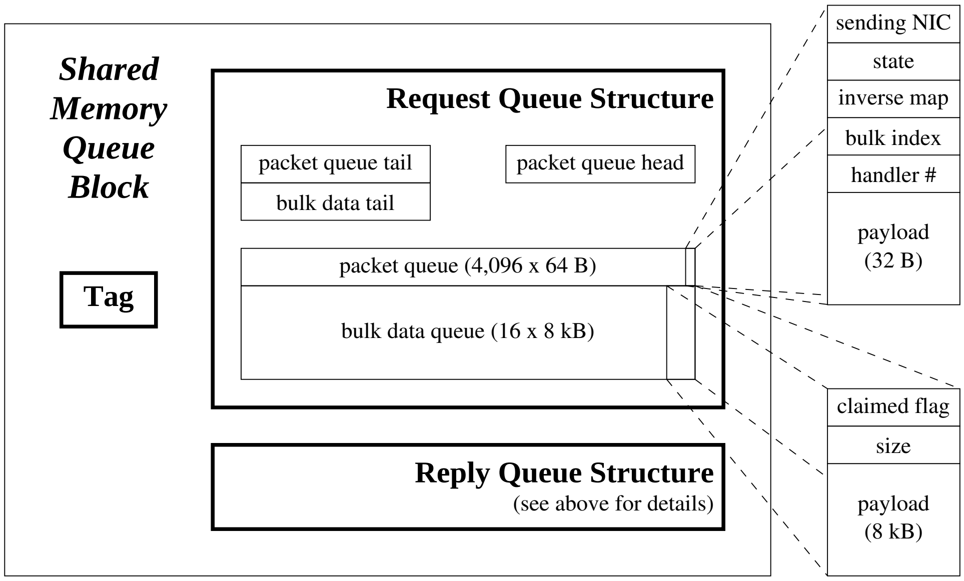

Figure 3.2: Block diagram of a shared memory queue block. Short messages use only the packet queue. Bulk data transfers use the bulk data queue as well.

hold request and reply messages received by the endpoint. Each queue structure further divides into three sections: queue tail information, accessed only by senders; queue head information, accessed only by recipients; and two cyclic data queues, accessed by both senders and recipients. The queues are the packet queue, which contains the handler index and arguments, and the bulk data queue, which holds data for bulk data transfers. Short messages use only the packet queue, while bulk data transfers use both queues.

To eliminate false sharing and thereby reduce the number of memory transactions, the following data occupy distinct L2 cache lines:

- the endpoint tag
- the two queue tails (on one cache line)
- the packet queue head
- each packet
- each bulk data block claimed flag and size field
- each bulk data block payload

In addition to the handler index and arguments, entries in the packet queue contain four other fields: a NIC number, a packet state, an inverse queue block mapping, a bulk data index. The first of these, the NIC number, is used by the Active Message layer to select the appropriate protocol for a reply message (see Section 3.2). The packet state differentiates between short messages and bulk data transfers and serves as the handshake state in transferring data from a sender to a recipient. A claimed flag serves the latter purpose for the bulk data queue. The inverse queue block mapping points to the shared memory queue block of the sending endpoint in the address space of the process that owns the receiving endpoint, enabling reply messages to avoid a potentially expensive lookup operation. The last field, the bulk index, records the association between a bulk data transfer packet and the data itself.

Two factors govern the length of the queues: multiprogramming performance and memory footprint. The long packet queue allows an application to receive a relatively large number of short messages while descheduled. With medium messages, greater insertion overhead makes queue length less important, and we select a shorter queue to keep the memory requirement reasonable: 0.75 MB for the full block.

3.1.3 Queue structure rationale

The shared memory queue block differs significantly from the network queue block in its lack of send queues. The absence arises from a fundamental difference between the methods used to transmit data over the network and within an SMP. In the network case, a sender cannot directly deposit data into memory located across the network, and must instead rely on a third party, such as a Myrinet NIC, to move the data. Within an SMP, the situation is just the opposite: direct access is possible through shared memory, and no third party exists to perform the transfer.

The separation of the request and reply queues avoids a well-known deadlock scenario [CSwAG98]. Consider two endpoints with single queues, each of which is filled with requests. If the endpoints simultaneously issue request messages to one another, both must wait. Neither endpoint can make space in its own queue without handling a request and issuing a reply, but the target queue for the reply may also be full, and a third endpoint may fill the newly created space, in which case the replying endpoint returns to its original state with a deeper call stack. The stack is finite, but failing to make space results in deadlock.

The problem can also be solved by limiting the total number of requests (not the number per processor) in the network to few enough that a process' stack can absorb them all. This approach does not scale to large systems, however.

As shown in the figure, only two queue structures are allocated for each endpoint. Multiple endpoints can insert messages into these queues concurrently, and the enqueue operation must be carefully designed to ensure that concurrent operations occur atomically with respect to one another. Concurrent access also increases the importance of the queue block layout, as several processors' caches may compete for the data at any point in time.

3.2 Ping-Pong Example

Communication protocols like active messages obtain high performance by optimizing frequent operations, such as sending a message, at the expense of infrequent ones, such as creating an endpoint. In this section, we illustrate the techniques used for this optimization through an example of a two-process ping-pong communication. Given processes A and B within a single SMP, we step through endpoint creation and naming, discuss resource location and access control, cover the actual communication, and conclude with comments on endpoint destruction. We focus in slightly more detail on the communication operations, but defer the most important operation—the insertion of a message into a queue—until Section 3.3.

3.2.1 Endpoint creation

Process A begins by creating endpoint 1. In the creation process, the active message layer allocates space for the three major components of the endpoint and initializes all fields to their default values. The control block is allocated from private host memory. The endpoint is assigned a unique physical name by the NIC, and the traffic statistics are initialized to indicate no traffic and no network skipping. Handler 0 is set to the default error function. All other handlers are set to `abort()`, and all destination endpoint entries are marked as invalid. A shared memory segment is allocated for the shared memory queue block, and the segment identifier is recorded as part of the endpoint's physical name in the control block. The segment is mapped into process A's address space. The tag is set to a distinguished value that causes rejection of all messages. The queue structure head and tail values are initialized, and the packets and the bulk data blocks are marked as FREE. Backing storage for the network queue block, which contains information similar to that found in the shared memory queue block, is allocated from kernel memory and mapped into process A's address space. The block is dynamically paged onto the NIC when process A inserts its first remote message or performs certain control operations.

After creating the endpoint, process A performs application-level endpoint initialization, installing a reply handler for the ping-pong communication and setting the tag to a value known by process B. AM-II does not specify the methods through which tags are found. We typically preselect one value for each application for simplicity,[2] but one can construct many more complex and secure schemes with or without the use of active messages. As an alternative, process A can set the tag to accept all incoming messages using a second distinguished tag value.

3.2.2 Resource location

Process A has completed the creation of its communication endpoint and is ready to initiate contact with process B. As with tags, AM-II allows an application to define its own methods for the discovery of endpoint names. A simple hash table server suits the purpose for our example. First, process A generates a virtual, application-level name for endpoint 1, "A's endpoint," and declares the virtual-to-physical name translation publicly through the hash table server. Process A then waits for the appearance of "B's endpoint," at which point it learns the physical name of endpoint 2.

3.2.3 Destination preparation

Process A is now prepared to map endpoint 2 into endpoint 1's table of destination endpoints, bringing endpoint 2's shared memory queue block into its address space to permit the direct insertion of messages. As the queue block might be accessible to a number of endpoints, this mapping assumes a high level of trust between processes A and B. Mapping an endpoint thus also implies making an access control decision. If this level of trust is not acceptable, a process can sacrifice performance in favor of security by allocating a separate endpoint for each process with which it communicates.

[2]For reasons beyond the author's ken as a vegetarian, tags are commonly set to the value 0xDEADBEEF.

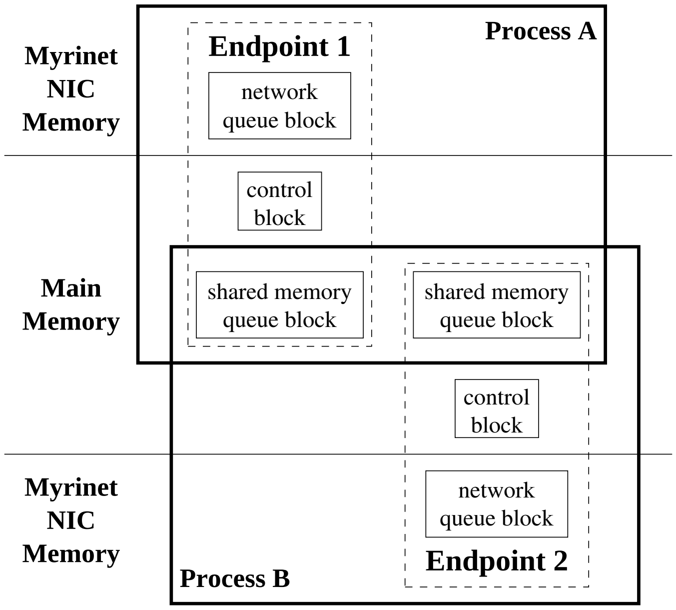

Figure 3.3: Two processes ready to communicate. Each process owns one endpoint and has mapped the shared memory queue block of the other process' endpoint into its address space.

The use of System V shared memory segments as storage implicitly ties access control to the model supported by System V interprocess communication (IPC). The IPC model is identical to the one used by traditional Unix file systems. Each segment has distinct read and write access bits for the owner of the segment, for a Unix group associated with the segment, and for all other users. In future operating systems, we expect shared memory segments to benefit from advances in file system access methods. Access control lists are on the horizon with commercial operating systems, and a more general capability system is perhaps not much more distant. Extensions to the Myrinet NIC driver code offers another alternative: process A might be required to present endpoint 2's tag before being allowed to map endpoint 2's queue block into its address space.

The mapping operation takes as arguments a destination index, an endpoint name, and a tag. The NIC and endpoint numbers from the endpoint name are simply copied into the corresponding fields in the destination entry, as is the tag. Using the name, the active message layer decides whether the endpoint is local or remote. Process A finds that endpoint 2 is local and checks the shared segment identified as a part of endpoint 2's name. After checking a hash table for existing mappings of the segment, process A maps the segment, adds an entry to the hash table, and records the segment address in the destination entry. The entry is then marked as valid. The last field of the destination entry, the inverse mapping, must be assigned by process B. Process A sends a local message to endpoint 2 requesting that process B map endpoint 1's shared memory queue block and return the segment address to endpoint 1. When the reply arrives, the address is copied into the destination entry, completing the mapping operation. Figure 3.3 shows the situation at this point. Although unimportant to the example in this chapter, control operations such as setting the tag have paged the network control blocks in the figure onto the NIC's.

3.2.4 Ping-pong communication

The communication layer is fully initialized, and process A is ready to communicate with process B. Sending a short request requires only three arguments: the source endpoint, the destination endpoint index, and the handler index. Additional arguments are passed with the message and delivered to the handler as formal parameters. As the first step in sending a message, the active message layer validates the state of the system, checking that all initializations are complete and that the source and destination endpoint are valid.

Once the arguments have been checked, the communication layer decides which protocol to use. Through precomputation in the mapping operation described earlier, this decision is reduced to a single comparison. The shared memory queue block pointer in the destination entry is NULL for a remote endpoint, and non-NULL for a local endpoint.

A poll for incoming messages constitutes the next step. Checking for message arrival on every send operation helps the layer to remain responsive to incoming traffic. The local poll operation need only check the state of the packet at the head of each packet queue. When a message is available, the recipient advances the packet queue head and passes the arguments and, for bulk data transfers, the associated data block, to the appropriate handler routine. After this call returns, the packet and the data block are marked as FREE.

The two steps following the poll are specific to the shared memory protocol. The inverse map field of the destination entry must be copied into a request message to optimize the process of sending a reply. If this field is NULL, the send operation polls until the arrival of the reply carrying the datum. Once a valid inverse mapping has been found, the active message layer checks the tag in the destination entry against the tag in the destination endpoint's shared memory queue block. Messages with invalid tags are returned immediately via the sending endpoint's handler 0. For the network protocol, the tag check is performed by the NIC associated with the destination endpoint.

At this point, only message insertion remains. We defer the details of this operation to Section 3.3, but note that, as an effect of the insertion, process A changes the state of one of the packets in endpoint 2's request queue structure to indicate the presence of the message. Successful insertion of the message into endpoint 2 marks the end of the send operation, and process A enters a polling loop to await process B's reply.

Process B, which has also completed its initializations, has been waiting in a similar loop and notices the request message when it appears in endpoint 2's queue. The communication layer passes information about the message to the application's handler via an opaque message token, and the application hands this token back to the layer as one of the arguments to a reply operation. The second argument to the reply is a handler index, and the remaining arguments are passed as formal parameters to the reply handler.

A reply requires only protocol selection, polling, and message insertion. Argument checks are skipped—the token and its contents are assumed to be valid. Selecting the appropriate protocol for a reply message again requires only a single comparison. The message packet for remote messages contains the NIC number of the source endpoint; local messages instead fill this field with the NIC number of the destination endpoint. The reply operation compares this value with the replying endpoint's NIC number: equality indicates a local message, and inequality indicates a remote message. As messages from an endpoint to itself are never remote, overloading the meaning of the NIC number field does not cause any problems. After a poll, the reply operation fills a packet in endpoint 1's reply queue structure and returns.

3.2.5 Endpoint destruction

The ping-pong communication is complete, and the two processes are almost ready to terminate. Process A first removes the translation of "A's endpoint" from the hash table server, then destroys endpoint 1, unmapping endpoint 2's shared memory queue block and freeing the memory associated with endpoint 1. The semantics of shared memory segments delays the actual destruction of endpoint 1's segment until all other processes (in the case of our example, process B) have unmapped the segment. Having freed these resources, process A terminates.

3.3 Lock-Free Algorithm

The shared memory message queues permit multiple processes to insert messages concurrently, requiring atomic enqueue operations to prevent interference. Our many-to-one approach distinguishes this thesis from much of the existing work on shared memory message-passing. Managing concurrent access efficiently is the aspect of the shared memory protocol most critical to performance and presents a complex and challenging problem. In this section, we focus on the algorithm used in our active message implementation and provide a few basic notions about alternatives. A more detailed discussion of the latter appears in Chapter 6.

Traditional concurrent access algorithms use critical sections to prevent interference between processes: a process obtains a mutually exclusive lock to enter a critical section, thereby preventing other processes from entering concurrently. A process that stalls or completes its scheduling quantum while holding a lock can delay other processes indefinitely, and a process that waits busily to obtain a *spin lock* can waste processor cycles. Numerous techniques for avoiding these problems appear in the literature, and together are known as preemption-safe locking [MS97]. Such techniques generally require interaction with the operating system and can thus incur significant overhead. We avoid mutual exclusion through the use of a lock-free algorithm. Our algorithm couples synchronization tightly to the data structure and results in superior performance.

The section begins with a description of the algorithm, discussing logical synchronization primitives and their specific use in message insertion. We then consider the actual requirements for hardware support of the algorithm, highlighting alternative primitive implementations. We conclude with performance predictions relative to locking algorithms based on the structure of our algorithm. In Chapter 6, we confirm our predictions on an Enterprise 5000, demonstrating results superior to those obtained with alternative algorithms on both dedicated and multiprogrammed machines using a suite of microbenchmarks and applications. We also compare performance on an SMP with performance on a comparable NOW, illustrating the impact of the faster protocol at the application level.

3.3.1 Message insertion

The lock-free algorithm relies on two synchronization primitives, FETCH&INCREMENT (F&I) and COMPARE&SWAP (CAS), to ensure atomicity with respect to other mes-

FETCH&INCREMENT($\hat{}\,address$)
    $value \leftarrow address\hat{}$
    $address\hat{} \leftarrow value + 1$
    return $value$

COMPARE&SWAP($\hat{}\,address, old, new$)
    if $address\hat{} = old$
        $address\hat{} \leftarrow new$
        return TRUE
    return FALSE

Figure 3.4: Pseudo-code for synchronization primitives. Each operation is atomic with respect to other memory accesses.

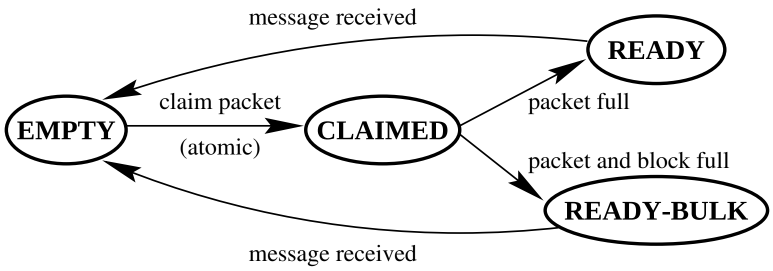

Figure 3.5: State diagram for message packets. The FREE to CLAIMED transition requires instruction-level atomicity for correctness. Unique threads of control perform all other transitions, allowing for non-atomic operations.

sage insertions. The hardware must in turn guarantee that these primitive operations are atomic with respect to all other memory accesses. FETCH&INCREMENT(*address*) adds one to the value at an address and returns the previous value. COMPARE&SWAP(*address, old, new*) compares the value at an address with an expected value, old. If the two values are equal, the operation writes a third value, new, into the address and returns TRUE. Otherwise, CAS returns FALSE. Pseudo-code for these primitives appears in Figure 3.4.

As a message moves through a message packet, the packet cycles through three states, as depicted by Figure 3.5. To enqueue a short message, a sender claims a packet in the destination queue structure with CLAIMPACKET, changing the state of the returned packet from FREE to CLAIMED, then fills in the packet. Claiming a packet involves concurrent access to the queue and must be performed atomically with respect to other claims, but only one sender accesses a packet while filling it. Once a packet is full, the sender changes its state to READY. For bulk data transfers, a sender uses CLAIMBULK to

```
ClaimPacket(^q)
    index ← Fetch&Increment(q^.tail) mod Q_LENGTH
    while TRUE
        if Compare&Swap(q^.packet[index].state, FREE, CLAIMED)
            return index
        (back off exponentially and poll)

ClaimBulk(^q)
    block ← Fetch&Increment(q^.bulk_tail) mod BULK_LENGTH
    while TRUE
        if Compare&Swap(q^.bulk[block].claimed, FREE, CLAIMED)
            index ← ClaimPacket(q)
            q^.packet[index].bulk_index ← block
            return index
        (back off exponentially and poll)
```

Figure 3.6: Pseudo-code for our lock-free approach to claiming a packet from a queue q. ClaimBulk claims both a packet and a bulk data block.

claim both a packet and a bulk data block, fills in both, and changes the packet state to READY-BULK. As described earlier, the receiver's poll operation detects when a message at the head of a packet queue is ready for delivery and invokes the appropriate handler. After this call returns, both the packet and the data block are marked as FREE, completing the cycle of states.

Pseudo-code for the claim operations appears in Figure 3.6. The analogue of the critical section in ClaimPacket consists of two steps. First, a sender obtains a packet assignment by atomically incrementing the queue tail using F&I. Next, the sender claims the assigned packet by changing its state from FREE to CLAIMED with CAS. The number of assigned packets may exceed the queue size, in which case multiple senders compete for a single packet in the second step. ClaimBulk takes a similar approach, obtaining a bulk data block and claiming that block before competing for a packet.

If a sender's claim fails, the queue is full, and the sender backs off exponentially to minimize memory transactions. The backoff strategy starts with a 1 microsecond delay and doubles the delay with each failure to a limit of 255 microseconds. If the backoff strategy reaches the delay limit and the claim continues to fail, the sending process relinquishes its processor to allow another process—perhaps the queue's receiver—to make progress.

The backoff strategy can also benefit from interaction with the operating system

TEST&SET(^$address$)
    $value \leftarrow address$^
    $address$^ $\leftarrow$ LOCKED
    return $value$

FETCH&INCREMENT(^$address$)
    repeat
        $value \leftarrow address$^
        $next \leftarrow (value + 1)$
    until COMPARE&SWAP($address$^, $index, next$)
    return $value$

Figure 3.7: Pseudo-code for TEST&SET and for a version of FETCH&INCREMENT based on COMPARE&SWAP. This form of FETCH&INCREMENT admits starvation.

scheduler. Rather than simply yielding its processor, a process can sleep until an event occurs. This approach informs the scheduler that the process currently has no useful work and does not wish to compete with the processes responsible for providing it with work. When signaling an event, a process must notify the sleeping process through the kernel. In keeping with active message semantics, event notification is synchronous with respect to control flow within the program. As Arpaci-Dusseau *et al.* found for uniprocessor clusters [ADCM98], an application must utilize this method at all levels in order for the strategy to be fully effective. Due to subtle issues with AM-II kernel support, we did not integrate event notification between the protocols, and use it only with the shared memory protocol in isolation. Events such as waiting for a concurrent queue to drain permitted no obvious signaling criteria, hence we did not use events in the case of full queue backoff. As apparent from the performance data in Chapters 6 and 7, events do improve multiprogramming performance, but the overhead of the additional interaction with the kernel is also non-trivial and can degrade performance on dedicated machines.

The code shown in the figure requires that both Q_LENGTH and BULK_LENGTH be powers of two, but removing this restriction requires only slight modifications.

3.3.2 Hardware support

The lock-free algorithm depends logically on the availability of the F&I and CAS synchronization primitives, but can be implemented with others. Consider, for example, the CAS-based version of F&I shown in Figure 3.7. As the Sparc instruction set lacks the

state	lock	substate
FREE	UNLOCKED	FREE
CLAIMED	LOCKED	FREE
READY	LOCKED	READY
READY-BULK	LOCKED	READY-BULK

Table 3.1: Extended state requirements for a TEST&SET-based variant of the lock-free algorithm. Recall that only the transition from FREE to CLAIMED need be atomic.

F&I primitive, that architecture requires the use of this alternative solution. The approach shown decouples the read and modify components of the F&I from the write component, yet guarantees atomicity by performing the write only when the value remains unchanged. Such constructions have an interesting theoretical history, culminating in Herlihy's definition of universality for synchronization primitives in [Her88]. A *universal* primitive (*e.g.*, CAS) can be used to implement any other primitive in such a way that no process impedes the progress of any other. Although based on CAS, the F&I shown in the figure does not prevent such interference. A successful F&I by one process can cause an F&I attempt by a second process to fail, even to the point of starving the second process. In return for a weaker guarantee, this version of F&I provides much better performance in the common case. We present a version that prevents starvation in Chapter 9.

The specific requirements of the lock-free algorithm also admit replacement of the CAS in the claim step with a simple TEST&SET (T&S) primitive. TEST&SET(*address*) replaces the value at an address with a LOCKED value and returns the previous value, and was perhaps the first synchronization primitive ever suggested. Pseudo-code for T&S appears in Figure 3.7. Use of T&S with the lock-free algorithm requires that the packet state be split into lock and substate values, as shown in Table 3.1. The table lists equivalent values for each possible state. Using T&S, the EMPTY to CLAIMED transition remains atomic: only one process can succeed. The sender implicitly hands the "lock" to the receiver by setting the state to READY or READY-BULK, and the sender must clear the lock after changing the substate to FREE to complete the transition back to the EMPTY state.

3.3.3 Performance notions

Careful scrutiny of the lock-free algorithm allows us to predict its performance relative to locking algorithms for both high and low levels of contention. The key to both predictions is the separation between assigning a packet and claiming it. This separation increases the number of synchronization primitives performed and results in slightly worse performance in the absence of contention. The same separation, however, results in a much shorter window of vulnerability to contention. The bottleneck step is packet assignment, for which the Enterprise 5000 uses the CAS construction of F&I. This approach is vulnerable to failure only between the completion of the queue tail load and the execution of the CAS, a period covering roughly a tenth of a microsecond on that machine. In contrast, the critical section in a locking algorithm spans at least two cache misses (the queue tail and the packet state) and totals roughly a microsecond. Hence we expect the lock-free algorithm to outperform locking algorithms under contention.

In practice, our lock-free algorithm demonstrates performance superior to both spin locks and preemption-safe locks. The degree of robustness afforded by eliminating locks reduces the impact of interactions with the scheduler yet avoids the high overhead inherent to operating system support. As shown in the detailed performance results in Chapter 6, the lock-free algorithm outperforms other algorithms on both dedicated and multi-programmed machines.

3.4 Theoretical Analysis

In this section, we address several theoretical aspects of our lock-free algorithm. We begin with two proofs of correctness. After making a few basic assumptions about the nature of the system and application program, we prove that the algorithm neither loses messages nor admits deadlock. A discussion of the queue semantics provided by the algorithm follows the proofs, and we conclude the section with commentary on the relative disadvantages of pointer-based algorithms.

3.4.1 Message loss

We first prove that messages do not get lost in a queue—that the receiver cannot wait indefinitely for a message to arrive at the head of the queue while ready messages are present elsewhere. Equivalently, a message inserted atomically into an empty queue must always appear at the head of the queue. In order for this property to hold, the process scheduler must not allow any process to starve. A process that claims the packet at the head of a queue and is subsequently starved by the scheduler blocks all further message reception from that queue.

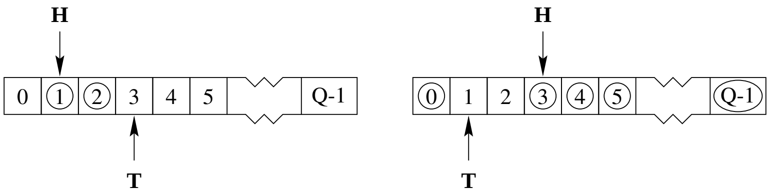

Figure 3.8: Cyclic message queue illustration. Numbers of active packets are circled.

Before formally stating this theorem, we make two definitions. First, recall that when a sender calls ClaimPacket, the algorithm assigns the sender one packet from the queue. Such an assignment is *outstanding* from its time of inception until the receiver frees the assigned packet after handling the message associated with the assignment. Note that we use the term assignment as an abbreviation for packet assignment and do not mean to include the assignment of bulk data blocks. We address bulk data transfers as a corollary to the theorem for short messages. Second, a process scheduler is *starvation-free* if there exists a time quantum $\tau \in \mathbb{R}$, $\tau > 0$ such that a process competing with $P-1$ other processes is guaranteed to execute at least one instruction in any time period of length $P\tau$. In these terms, we state our theorem:

Theorem 3.1 *If governed by a starvation-free scheduler, the packet at the head of a queue with outstanding assignments becomes ready for delivery in bounded time.*

Intuitively, we prove this theorem by establishing that the distribution of outstanding assignments over the queue is fairly even and that the packet at the head of a non-empty queue always has an outstanding assignment. The starvation-free property then guarantees that the head packet becomes ready for delivery in bounded time. We prepare for the formal proof by establishing three useful lemmas.

Consider a cyclic queue consisting of Q message packets numbered 0 through $Q-1$. Let H be the number of the packet at the head of the queue, and define T to be the number of the packet at the tail. A packet i is termed *active* if it falls between the head and the tail of the queue, as illustrated by Figure 3.8. In particular, packet i is active when it satisfies the inequality

$$(i - H) \bmod Q \quad < \quad (T - H) \bmod Q \tag{3.1}$$

Let A be the set of active packets:

$$A \equiv \{i \in \mathbb{Z},\ 0 \le i < Q \quad | \quad \text{packet } i \text{ is active}\}$$

and note that $|A| = (T - H) \bmod Q$. Our first lemma concerns properties of the set A.

Lemma 3.2 *The head and tail of the queue have certain properties with respect to A. In particular,*

(i) $T \notin A$

(ii) $H = T \ \rightarrow \ A = \emptyset$

(iii) $H \neq T \ \rightarrow \ H \in A$

(iv) $A \neq \emptyset \ \rightarrow \ H \in A$

Proof of Lemma 3.2: Proof of the first three clauses requires only the replacement of variables in (3.1). For (i), replace i with T. (ii) follows from replacement of H with T. To see (iii), replace i with H and recognize that T can differ from H by at most $Q - 1$. (iv) follows directly from (ii) and (iii): $A \neq \emptyset \ \rightarrow \ H \neq T \ \rightarrow \ H \in A$.

Having established these properties, we are ready to state the lemma central to the theorem. Let C_i be the number of outstanding assignments to packet i.

Lemma 3.3 *Outstanding assignments are distributed evenly over a queue, with active packets assigned one time more than packets that are not active. In particular,*

$$\exists n \in \mathbb{Z},\ n \ge 0 \text{ such that}$$
$$\forall i \in \mathbb{Z},\ 0 \le i < Q,\quad (i \in A \ \wedge \ C_i = n + 1) \ \vee \ (i \notin A \ \wedge \ C_i = n) \qquad (3.2)$$

Proof of Lemma 3.3: In a queue's initial state, the C_i are uniformly 0, and $T = H = 0$. By clause (ii) of Lemma 3.2, $A = \emptyset$, thus $n = 0$ satisfies (3.2). We prove the lemma by showing that (3.2) remains true under the operations of packet assignment and message reception. Let unprimed variables represent the state of the queue before an operation and primed variables represent the state after the operation. Furthermore, let n satisfy (3.2) before an operation. The proof then requires that we find a value of n' that satisfies (3.2) after the operation.

First consider packet assignment. By definition, assignments are made to the tail of the queue, advancing the tail cyclicly:

$$\begin{aligned} H' &= H \\ T' &= (T+1) \bmod Q \\ C'_T &= C_T + 1 \\ \forall i \in \mathbb{Z},\ 0 \le i < Q,\ i \ne T, \quad C'_i &= C_i \end{aligned}$$

Notice that by clause (i) of Lemma 3.2, $T \notin A$, thus $C_T = n$ and $C'_T = n+1$.

Two cases are possible. If the assignment does not bring the tail to the head, $T' \ne H$, and we can write

$$\begin{aligned} & (T' - H) \bmod Q = [(T - H) \bmod Q] + 1 \\ \text{hence} \quad & A' = A \cup \{T\} \end{aligned}$$

In this case, packet T becomes active due to the assignment. $C'_T = n+1$ thus implies that $n' = n$ satisfies (3.2).

Next consider the case in which the assignment brings the tail to the head, $T' = H$. By clause (ii) of Lemma 3.2, $A' = \emptyset$, and all of the C'_i must equal n' to satisfy (3.2). A choice of $n' = n+1$ results in $C'_T = n'$, but the same must also hold for the other C_i, *i.e.*, all other packets must be active prior to the assignment. Replacing T with $T'-1$ in (3.1), we find:

$$\begin{aligned} & (i - H) \bmod Q \quad < \quad (T' - 1 - H) \bmod Q \quad = \quad (-1) \bmod Q \quad = \quad Q - 1 \\ \text{hence} \quad & A = \{i \in \mathbb{Z},\ 0 \le i < Q \quad | \quad i \ne T\} \\ \text{and} \quad & \forall i \in \mathbb{Z},\ 0 \le i < Q,\ i \ne T, \quad C'_i = C_i = n + 1 = n' \end{aligned}$$

as desired, proving that n' satisfies (3.2) and thus that (3.2) remains true under packet assignment.

The proof of invariance under message reception is analogous. By definition, messages are received from the head of the queue, advancing the head cyclicly:

$$\begin{aligned} H' &= (H+1) \bmod Q \\ T' &= T \\ C'_H &= C_H - 1 \\ \forall i \in \mathbb{Z},\ 0 \le i < Q,\ i \ne H, \quad C'_i &= C_i \end{aligned}$$

Notice that $(H - H') \bmod Q = Q - 1$, which implies $H \notin A'$; a packet is never active immediately after a message has been received from it.

Two cases are again possible. If the head does not match the tail prior to reception, $H \neq T$, and clause (iii) of Lemma 3.2 implies that $H \in A$. Thus $C_H = n + 1$ and $C'_H = n$. $H \neq T$ also implies

$$\begin{aligned} & (T - H') \bmod Q = [(T - H) \bmod Q] - 1 \\ \text{hence} \qquad & A' = A \setminus \{H\} \end{aligned}$$

In this case, the head packet ceases to be active due to the reception. $C'_H = n$ thus implies that $n' = n$ satisfies (3.2).

Otherwise, the head and the tail match prior to reception, $H = T$, which by clause (ii) of Lemma 3.2 implies that $A = \emptyset$. Hence $C'_H = n - 1$ and all other C'_i are equal to n. We require $n > 0$ for message reception to occur, as $n = 0$ in this case implies an empty queue. As $H \notin A'$, a choice of $n' = n - 1 \geq 0$ satisfies (3.2) provided that all packets except H are active after the reception. Writing (3.1) in primed form and replacing H' with $H + 1$, we find:

$$\begin{aligned} & (i - H - 1) \bmod Q \quad < \quad (T - H - 1) \bmod Q \quad = \quad (-1) \bmod Q \quad = \quad Q - 1 \\ \text{hence} \qquad & A' = \{i \in \mathbb{Z},\ 0 \leq i < Q \quad | \quad i \neq H\} \end{aligned}$$

as desired, proving that n' satisfies (3.2) and completing the proof of the lemma.

Given an even distribution, we need now merely show that messages appear at the head of the queue first.

Lemma 3.4 *If a queue has outstanding assignments, the packet at the head of the queue has outstanding assignments. In particular,*

$$\forall i \in \mathbb{Z},\ 0 \leq i < Q, \quad C_i > 0 \ \rightarrow \ C_H > 0$$

Proof of Lemma 3.4: Pick $i \in \mathbb{Z}$, $0 \leq i < Q$ such that $C_i > 0$. If $i \in A$, $H \in A$ by clause (iv) of Lemma 3.2 and $C_H = C_i > 0$ by Lemma 3.3. Otherwise, $i \notin A$, and Lemma 3.3 implies that $C_H \geq C_i > 0$, completing the proof.

We are now ready to prove the theorem. Two facts are essential to the final proof: one process always wins the competition to claim a particular packet, and the code for claiming and filling a packet requires a finite number of instructions when successful.

Proof of Theorem 3.1: Assume that a queue has outstanding assignments at some time t and that the packet H at the head of the queue is not ready for delivery at that time. By Lemma 3.4, we know that H has at least one outstanding assignment, $C_H > 0$. Consider the code path for inserting a message into H after the assignment step. At that point, bulk data transfers have already obtained a bulk data block and are inside of ClaimPacket, as are short messages. A sender must back off from any previous claim failure, claim H, fill H and possibly a bulk data block, and change H's state. If the packet claim succeeds, as it must for exactly one sender when H is free, the code does not branch indefinitely (recall that the backoff period is bounded). Let N denote the maximum number of instructions executed by the successful process. Let P be the maximum number of processes that the system supports, and let $P\tau$ be the length of the time period in which a process is guaranteed to execute at least one instruction. As the scheduler is starvation-free, $P\tau$ is finite. The sender that succeeds in claiming H thus marks H as ready for delivery no later than time $t + NP\tau$, proving the theorem.

Theorem 3.1 holds for all messages, including bulk data transfers, once they have been assigned packets. Proving that a bulk data transfer waiting to claim a bulk data block also implies the arrival of a message takes a small additional effort. A bulk data block assignment is outstanding from its time of inception until the receiver frees the assigned block after handling the message associated with the bulk data block assignment.

Corollary 3.5 *If governed by a starvation-free scheduler, the packet at the head of a queue with outstanding bulk data block assignments becomes ready for delivery in bounded time.*

Proof of Corollary 3.5: Let p be a process with an outstanding bulk data block assignment. If p has yet to claim its assigned bulk data block, it attempts to do so in bounded time under a starvation-free scheduler. If the claim fails, let p' be the process that last successfully claimed the block assigned to p. Otherwise, let $p' = p$. If p' has already been assigned a packet, the claim must be outstanding since the bulk data block associated with p' has not been received (and hence neither has the assigned packet), and the proof is done.

Otherwise, a packet is assigned to p' in bounded time under a starvation-free scheduler, and the packet at the head of the queue becomes ready in bounded time by Theorem 3.1, completing the proof.

3.4.2 Communication deadlock

We next prove that the lock-free algorithm cannot result in deadlock within the communication layer. Intuitively, deadlock implies that a program reaches a state in which none of its processes is capable of making forward progress. A synchronous communication layer, however, can guarantee only that the processes do not deadlock within the layer. Avoiding deadlock in general also requires that each process remain responsive to communication. For the lock-free algorithm, we must demonstrate that processes can not all block indefinitely within the algorithm without sending or receiving messages. As processes only block within the algorithm when waiting to insert into a full queue, we use the full queue condition as the basis for a more formal construction.

Let $F(t)$ be the set of processes waiting to insert messages into full queues at time t, and let $T_f(t)$ be the target process for $f \in F$—the process that receives messages from the full queue on which f is blocked trying to insert a message. Define the insertion graph $G(V(t), E(t))$ to be the directed graph with vertices consisting of $F(t)$ and the set of target processes $T(t)$ and with edges from each member of $F(t)$ to its target process:

$$\begin{aligned} T(t) &\equiv \{f \in F(t) \quad | \quad T_f(t)\} \\ V(t) &\equiv F(t) \quad \cup \quad T(t) \\ E(t) &\equiv \{(f,g) \in F(t) \times T(t) \quad | \quad g = T_f(t)\} \end{aligned}$$

In light of this graph construction, we can provide a meaningful definition of deadlock within the message layer: a *deadlock* occurs when no process in $V(t) \neq \emptyset$ can send or receive a message in bounded time. We define deadlock in terms of $G(V(t), E(t))$ to avoid speculation about higher levels of software; processes outside of $V(t)$ may be capable of progress, but we cannot assume that such progress is possible.

Processes must remain responsive to the communication layer. Formally, a process is *responsive* if there exists a $\tau \in \mathbb{R}$, $\tau > 0$ such that the process sends a request or checks for message arrival on each endpoint at least once during any time period in which execution time outside of the communication layer totals at least τ. The actual time required depends on the time allocated to the process by the scheduler, but is bounded under a starvation-free scheduler. Note that sending a request causes the communication layer to check for incoming messages. We are now ready to state our theorem.

Theorem 3.6 *If all processes are responsive, the lock-free algorithm cannot result in deadlock.*

Note that the theorem does not require that the scheduler be starvation-free. If a scheduler starves a process and thereby prevents all processes from making progress, the result is starvation, not deadlock. However, in proving this theorem, we first show that some process does make progress under a starvation-free scheduler, which in turn implies that the process can make progress, independent of whether or not it actually does so.

The lock-free algorithm avoids deadlock by polling while backing off from a full queue, polling for all messages when sending a request, but polling only for replies when sending a reply. Although this additional splitting of the network substantially complicates the proof, it is also necessary to avoid deadlock, as mentioned in Section 3.1.3. The added difficulty revolves upon the case in which a process sending a request targets the full queue of a process that blocked while sending a reply. The latter process polls only for replies, but we cannot guarantee that a reply becomes available in bounded time without considering longer chains of dependencies. We begin the proof by establishing an evolutionary property of the target process relationship: a blocked process and its target process either make progress or reach the send-request, poll-reply state in bounded time.

Lemma 3.7 *Let $f \in F(t)$ be a process blocked on a full queue at time t, and let the responsive process $g = T_f(t)$ be f's target process. If governed by a starvation-free scheduler, one of the following holds:*

(i) Some process in $V(t)$ makes progress in bounded time by either sending or receiving a message.

(ii) f is blocked on g's request queue at time t, and g blocks on a full reply queue in bounded time.

Proof of Lemma 3.7: First consider the case in which $g \notin F(t)$, and assume for now that g does not block on a full queue before receiving a message. As f has an outstanding bulk data block or packet assignment to a queue owned by g and the scheduler is starvation-free, Theorem 3.1 and Corollary 3.5 imply that a message becomes ready for g to receive in bounded time. As g is responsive and the scheduler is starvation-free, g receives the message within bounded time after it becomes ready, resulting in case (i). However, g can block on a full queue before receiving a message, *i.e.*, also in bounded time. If any other process in $V(t)$ sends or receives a message before g blocks, the result is case (i). We can thus assume without loss of generality that G does not change until g blocks at a time t' with $V(t') = V(t) \cup \{T_g(t')\}$ and $E(t') = E(t) \cup \{(g, T_g(t'))\}$.

Next consider the case in which g is blocked (at either time t or time t') on a full request queue. Assume for now that g neither succeeds in inserting its request nor blocks on a full reply queue (while handling a request) before receiving a message. Given the assumptions, a message becomes ready for g to receive in bounded time. While blocked on a full request queue, g polls for messages and hence receives the message within bounded time after it becomes ready, resulting in case (i). If g succeeds in inserting its request before receiving a message, the result is also case (i). Otherwise, g blocks on a full reply queue in bounded time, and we make the same assumptions as before about the progress of other processes.

Thus all cases evolve in bounded time to the one in which g is blocked on a full reply queue. If f is blocked on g's request queue, case (ii) results immediately, hence assume instead that f is blocked on g's reply queue. Also assume for now that g does not succeed in inserting its reply before receiving a reply. Given the assumptions, a reply becomes ready for g to receive in bounded time. While blocked on a full reply queue, g polls for replies and hence receives the reply within bounded time after it becomes ready, resulting in case (i). If g succeeds in inserting its reply before receiving a reply, the result is the same, thus completing the proof.

The lemma is the framework for the rest of the proof, the essence of which lies in applying the lemma twice to locate a pair of processes that cannot enter the send-request, poll-reply relationship. We now prove the theorem.

Proof of Theorem 3.6: Assume without loss of generality that the scheduler is starvation-free; a scheduler that starves processes does not change the fact that a process can make progress (when scheduled). Let $G(V(t), E(t))$ be the insertion graph at some time t. If

$F(t) = \emptyset$, $V(t) = \emptyset$, and we are done. Otherwise, pick $f \in F(t)$ and let g denote its target process, $g = T_f(t)$. If f is blocked on g's reply queue, Lemma 3.7 guarantees that some process in $V(t)$ makes progress in bounded time, and we are done. Thus assume that f is blocked on g's request queue. Lemma 3.7 then guarantees either that some process in $V(t)$ makes progress or that g blocks on a full reply queue in bounded time. Assuming that the latter occurs and that it occurs at a time t', let $h = T_g(t')$. As g is blocked on h's reply queue at time t', Lemma 3.7 guarantees that some process in $V(t')$ makes progress in bounded time from t', which in turn occurs in bounded time from t, completing the proof.

As a final comment on deadlock issues, we note that the order defined by CLAIM-BULK—reserving space in the bulk data queue before competing for space in the packet queue—circumvents a deadlock scenario that arises with the reverse order. Consider a process performing a bulk data transfer and holding the only outstanding assignment to the packet at the head of a packet queue. If the associated bulk data queue is full, the process cannot complete its message insertion, but neither can the queues' owner receive any messages holding bulk data blocks until it receives the message at the head of the packet queue. In terms of the proofs, this reversed order of operations violates the assertion in the proof of Theorem 3.1 that a process with an outstanding assignment cannot delay indefinitely before marking its packet as ready; in this case, the process must wait for a bulk data block to become available, but that event never happens. Once messages can be lost in the queue, deadlock is straightforward.

3.4.3 Queue semantics

The lock-free algorithm gives rise to several subtle issues of semantics. The astute reader may have noticed that we avoided a detailed discussion of starvation due to the algorithm itself. In fact, the algorithm does not prevent such scenarios; rather it guarantees only that some process makes progress. A process that consistently loses the competition for its assigned packet, for example, may never insert its message into a queue. In practice, an individual packet is rarely assigned to more than one process simultaneously, and starvation does not occur.

Starvation is nevertheless an inherent aspect of all algorithms that compete for central control information using synchronization primitives that provide no guarantee of fairness. The prevalence of universal primitives on modern machines thus has a subtle drawback in that it has kept primitives that guarantee fairness from appearing in some instruction set architectures (*e.g.*, the Sparc). As with simpler non-universal primitives such as T&S, the number of times that a process can fail a universal CAS operation is not generally bounded by hardware. Whereas a primitive such as F&I typically guarantees fairness, a version of F&I constructed from CAS admits starvation. Although not a significant problem for today's machines, the level of contention and the potential for starvation grow with the number of processors in an SMP, and the problem might become significant in the future.

A second interesting aspect of the lock-free algorithm is its lack of a first-in, first-out (FIFO) property as defined in the literature. According to Gottlieb *et al.* [GLR83], a FIFO concurrent queue must ensure the following: "If insertion of a data item p is completed before insertion of another data item q is started, then it must not be possible for a deletion yielding q to complete before a deletion yielding p has started." Herlihy and Wing's notion of linearizability [HW90] leads to a similar definition. Our lock-free algorithm does not obey this rule. Consider processes A and B simultaneously inserting messages into a two-element queue. Assign the first packet to process A and the second to process B, then assume that process A is descheduled before claiming the packet. Process B fills the second packet and begins another insertion. The queue wraps and assigns the first packet again, and B fills this packet as well. At this point, the first packet contains B's second message and the second packet contains B's first message. When the receiver looks for incoming messages, the second message is delivered first, violating the FIFO property. As active messages do not guarantee in-order delivery, the lack of a FIFO property does not make the lock-free algorithm incorrect. Nevertheless, a simple extension considered in Chapter 6 suffices to achieve FIFO semantics.

Finally, one might question whether the lock-free algorithm is truly free of locks. The three-state handshake between sender and receiver can be viewed as a form of fine-grained locking on individual message packets. Despite this fact, we chose to use the term "lock-free" to highlight the avoidance of explicit critical sections and the significantly reduced likelihood of contention between processes.

3.4.4 Pointer-based comparison

The previous section discussed the subtleties of the lock-free algorithm. We now discuss a few hazards that the lock-free algorithm avoids by design, in particular by manipulating values rather than names, *i.e.*, pointers. Pointers can lead to problems with improper success of operations, increased complexity for potentially interleaved operations, and difficulties with reusing memory. The lock-free algorithm operates on a tail index and on packet states, using no pointers or other indirect references to data. We discuss problems with pointers and their solutions to highlight the advantages of the lock-free algorithm in terms of programming complexity.

The ABA Problem

Pointer-based algorithms that make use of the CAS primitive must be aware of the ABA problem, which arises because of a separation between the memory value manipulated by CAS, often the name of a datum, and the object semantically intended for manipulation, the datum itself or an entire data structure. In the ABA problem, a process reads the name A and proceeds to operate on that value. Before the process can perform its CAS operation, a second process replaces A with B, recycles the datum referenced by A, and replaces B with A. The meaning of A at this point is clearly different from its original meaning, but the CAS operation recognizes only the name A and the first process' CAS operation "succeeds."

Figure 3.9 demonstrates the result for a linked list. In section (1), the list contains the elements A and B. A process reads A and it's *next* pointer, B. In section (2), a second process has removed A, leaving only B in the list. In section (3), further operations have moved B from the list to a free list of cells, while A has been recycled and returned to the list. At this point, the first process performs its CAS. The CAS compares the head of the list with A and "succeeds," placing a long string of garbage beginning with B onto the list, as shown in section (4).

Apart from avoiding the use of indirection, which is not always practical, two solutions exist for the ABA problem. The more common of the two is an algorithmic approach: each pointer is extended with an epoch number, and the CAS operation is applied to both the pointer and the epoch number simultaneously, incrementing the epoch with each successful CAS. If the hardware supports CAS on data significantly wider than

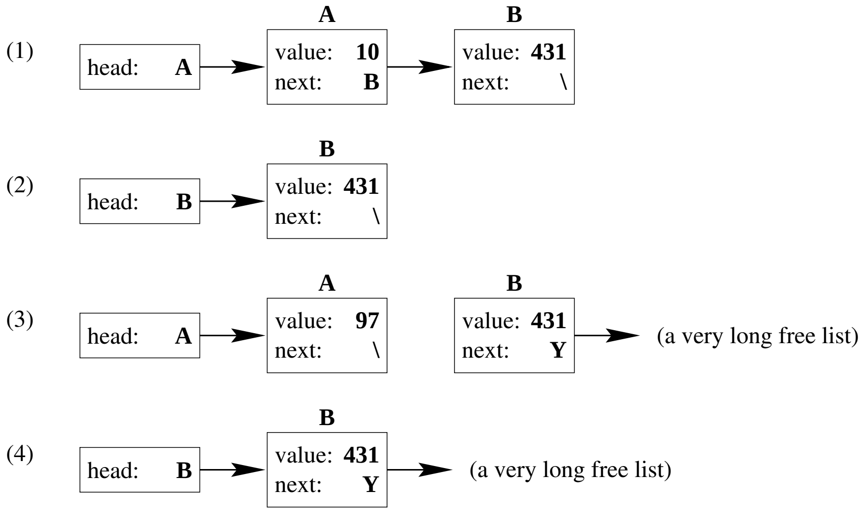

Figure 3.9: Illustration of the ABA problem on a list. See the text for details.

a pointer, then this approach is very unlikely to fail in practice. On machines without such support, however, an algorithm must make use of reduced pointers (either indices or truncated pointers) while retaining enough bits in the epoch number to make the ABA problem effectively impossible.

The second approach to the ABA problem makes use of a slightly different set of hardware primitives known as LoadLinked (LL) and StoreConditional (SC). The initial read operation in an algorithm is replaced with an LL primitive, and the CAS operation is replaced with an SC primitive. SC is guaranteed to fail if any other processor has changed the value at the linked address, so that success of the SC operation is in many cases equivalent to an absence of intervening changes to the data structure (and certainly to the location being changed).

Inconsistent Versions

A second problem arises when a process holds a stale reference to an object that is no longer an active part of the data structure. For correctness, pointer-based lock-free algorithms generally avoid non-atomic changes to the data in a concurrent structure, but such restrictions do not apply to objects once they have been retired from use. In fact, keeping old objects around for reuse is a fairly common method of avoiding dynamic memory overhead (and a significant contributor to the likelihood of the ABA problem).

Herlihy provides one solution to this problem in [Her93]. In his solution, algorithms maintain consistency information with each object. Before using an object, an operation makes a copy of the object, then checks the copy for consistency. Operation must also mark objects as inconsistent before manipulating them. The object becomes consistent again after the operation completes.

Typed Memory

The vagaries of schedulers and virtual memory can impose much more significant inconsistencies on a pointer-based algorithm. Once used in a concurrent data structure, a pointer to an object X of type Y may persist indefinitely in a register of some descheduled process. If the memory used by X is later reclaimed and recast as an object of type Z, the process can return to find the object Y completely incomprehensible. Dynamically-typed objects provide one possible solution to this problem, although concurrent operations must check the type very frequently in such a system, possibly after every memory read, and must also verify the type atomically with each write. Alternatively, Greenwald and Cheriton suggest in [GC96] that memory used for concurrent objects be "type-stable," meaning that the memory used for object X can only be reclaimed if the program can guarantee that no outstanding references to X exist.

Data Manipulation

None of the problems just discussed applies to our lock-free algorithm. The packet queue remains fixed in a single memory location, allowing the lock-free algorithm to treat packet assignments as data rather than pointers. The packet states also have a small number of fixed values, and the lock-free algorithm merely performs transitions within the state diagram. Many lock-free and non-blocking algorithms manipulate linked data structures; the data structures move from point to point in memory, leading to problems with detecting changes (the ABA problem), recognizing stale data (inconsistent versions), and avoiding operations on random values (typed memory).

Chapter 4

The Clump Polling Architecture

We now consider the third and final design challenge for multi-protocol communication, the development of a strategy to minimize the impact of polling each protocol upon messages passed through the other. The abstraction of uniform communication supported by our multi-protocol system must extend to communication performance as well, providing good results on a range of underlying platforms with only a single binary. Obtaining such results in general requires that we address the performance of the protocols both in isolation and in combination. The possibility that a programmer has addressed the hierarchy to improve performance by separating an application into phases of local and remote communication strengthens the dynamic nature of the problem. Our layer must tune communication to the Clump actually supporting the application.

In the previous chapter, we addressed the design of our shared memory protocol in isolation, considering only those issues pertinent to use within a single SMP. Chun *et al.* addressed the design of the network protocol in [CMC98]. In this chapter, we investigate the issues for a communication layer that supports both shared memory and network protocols. The central problem arises from the disparate costs of polling for incoming messages on the two protocols, and our solution takes the form of an adaptive strategy for polling. We begin the chapter with a more detailed description of the problem and possible solutions, then develop a parametrized framework for our adaptive strategy. The parameters allow us to tune the strategy with application performance data across a range of Clumps. We conclude the chapter with a discussion of the parameters and their impact on performance.

4.1 Multi-Protocol Poll Overhead

Message polling operations are ubiquitous in active message layers. Responsiveness demands that a layer poll for incoming messages when sending a message [BK94]. In the multi-protocol implementation, the interaction between the lightweight shared memory protocol and the more expensive network protocol can have a significant impact on the performance of the former. The problem stems from the cost of checking empty message queues; only successful message receptions perform useful work. An empty packet at the head of a shared memory queue remains cache-resident during periods of communication and costs only a handful of cycles to read. Network packets, however, reside in uncacheable NIC memory. Reads from this memory incur roughly a microsecond of overhead, leading to an order of magnitude difference between polling costs for the two substrates. We must carefully address this difference to obtain good performance for programs that send mostly local messages.

The network polling overhead breaks into two components, the number of reads from empty network queues and the cost of each read. One method for reducing the latter component involves moving the network queues into main memory, as do several other fast communication layers [WBvE97, PLC95, PT98, VIA97]. However, such a move requires an extensive redesign of the network protocol to avoid penalizing remote messages. We preferred to avoid making broad changes to the network protocol. Further, with the Myrinet SBUS cards used in our Sparc-based Clump, the cost of moving data into main memory with the NIC can be prohibitive for small transfers. We instead focus on reducing the first component, the number of reads from empty network queues.

4.1.1 Acceptance counts

A poll operation typically ends after reading an empty packet, but can also end after accepting a predefined number of messages. By reducing this number, we also reduce the number of reads from empty queues. Accepting too few messages, however, allows queues to overflow frequently and incurs substantial flow control delays.

The acceptance count touches on fairly complex issues. A naive argument leads one to believe that accepting a single message is sufficient. The communication layer pairs requests and replies on a one-to-one basis—each request message generates one reply message. A process receives one reply for each request that it sends. Similarly, a process sends one reply for each request that it receives. As an implicit poll operation occurs whenever a process sends a message, these equalities imply that the process polls at least once for every message received. In other words, a poll operation receives an average of at most one message.

The average-case argument fails to account for potential differences between the types of traffic, however. A process checks only for incoming replies when sending a reply, hence the number of polls for incoming requests does not necessarily exceed the number of requests sent to a process. The poll operation must acknowledge the possibility of an imbalance between messages sent by a process and messages sent to the process to avoid starving the latter. Furthermore, messages sent to a process may arrive later or in different time distributions from those sent by the process, and extending the queue delay typically degrades performance.

While providing for the possibility of traffic imbalances, the poll operation must also consider the issue of progress. If a poll accepts too many messages, it slows or halts progress by forcing the process that owns the queue to focus constantly on the demands of other processes.

We chose an acceptance count of four for our system. This value is large enough to accommodate incoming traffic but small enough to avoid starvation by other processes. We address empty packet reads through our adaptive polling strategy, which maintains our acceptance policy by increasing the number of messages accepted whenever it reduces the effective number of polls.

4.1.2 Alternative data structures

Changes to the network data structures also reduce the number of empty remote packet reads. Currently, a poll operation must walk through the packets in a network queue, examining each packet's type. The simplest change involves using a non-modular queue tail, which a process can then compare with the queue head to determine the number of messages in the queue. This change results in only a single network read to check for available messages. However, a substantial fraction of polls find their network queues completely empty, and the change has no effect in this situation. The altered data structure might improve network performance, but cannot solve the polling interaction problem completely.

A more substantial reworking of the network protocol to use pools of packets rather than queues has similar results for polling interaction. With pools, we dissolve the notion of ordering inherent to a queue by using bit masks to indicate each packet's status. For the network protocol, pools provide a much cleaner abstraction than queues, as they eliminate head-of-line blocking and couple flow control to message reception. They also require only a single network read, a bit mask, to check for available messages, but again they do not solve the problem completely.

As both of these options required changes to the network protocol, while neither promised a satisfactory solution, we performed neither.

4.1.3 Fractional strategies

Another approach to eliminating empty remote reads is to poll for remote messages less frequently, perhaps on every other poll operation, or on every tenth poll operation. We call this approach a *fractional strategy*. In order to remain responsive to remote traffic, a fractional strategy must accept more messages when checking the network. A fractional strategy that checks the network on every fourth poll operation and yet only receives four messages, for example, effectively limits remote message reception to one message per poll operation, with the same potential for starving incoming traffic as discussed earlier. To balance the reduced polling frequency, a fractional strategy accepts a correspondingly larger number of messages when checking for remote messages. A strategy that polls the network only once in every four calls to poll then accepts up to four times as many network messages in a single network poll as it does shared memory messages in a shared memory poll.

Although fractional strategies are fairly effective in reducing the performance impact of multi-protocol polling, they require a compromise between performance on Clumps at opposite extremes. An SMP, for example, works best with a very small fraction, perhaps one in sixty-four or smaller. A NOW, on the other hand, requires a reasonably large fraction, certainly no less than one in eight. No single fraction performs well on all platforms.

4.1.4 Adaptive strategies

An *adaptive strategy* overcomes the limitations of fractional strategies by adjusting polling rates dynamically in response to traffic patterns. A general adaptive strategy selects a subset of protocols for examination by each poll operation, but we consider a slightly simpler set of strategies that always poll for local messages. The cost of deciding whether to skip a queue in shared memory is comparable to the cost of polling the queue. Our adaptive strategy thus varies a fractional polling rate for the network between minimum and maximum values based on a history of recent remote message arrivals.

4.2 Adaptive Polling Framework

In this section, we develop a parametrized framework for adaptive polling strategies. As the critical path for sending a message includes a poll, we must carefully optimize any code necessary to the strategy. Our goal is thus to construct cheaply calculated estimates of message traffic based upon the number of messages received during each poll, and then to use those estimates to make polling decisions. Our construction includes a number of parameters that allow us to explore tradeoffs for which solutions are not intuitively obvious. We examine empirical data for selecting parameter values in Chapter 7.

We begin the section with a simple form of traffic estimation: given counts n_i of the number of messages received by the i^{th} poll operation, produce an estimate t_i of the number of messages that the $(i+1)^{th}$ poll will receive. The subscripts serve to differentiate between the individual poll operations, which we number beginning with 1 from the start of program execution. Similarly, superscripts differentiate between local and remote traffic. The text describes all other symbols, and in particular the parameters of the adaptive strategy. After addressing the simple form, we extend the equations to handle the case in which the n_i^{remote} have been artificially altered by ignoring remote messages during some polls. We close the section with a formulation of our poll skipping strategy in terms of the traffic estimates.

4.2.1 Single-poll estimation

We estimate traffic for each protocol using an exponential moving average (EMA) of the number of message arrivals during each poll operation. An EMA smooths discrete sequences of values by balancing each new value with all previous values, weighted with a decreasing geometric series. The resulting function decays exponentially in response to a step function in its input, giving the EMA its name. The factor used for weighting the values determines the degree of damping, or the rate at which the EMA tracks changes in the measured traffic. A high damping factor—one close to unity—works well for sparse but uniform arrival rates, while a low damping factor allows the EMA to respond quickly to changes. Formally, we define t_i as follows:

$$\begin{aligned} t_i &\equiv \alpha t_{i-1} + (1-\alpha) n_i \\ \text{or} \qquad t_i &= (1-\alpha)\left(\alpha^i \frac{t_0}{1-\alpha} + \alpha^{i-1} n_1 + \cdots + \alpha n_{i-1} + n_i\right) \end{aligned} \tag{4.1}$$

where $\alpha \in \mathbb{R}$ is a damping factor in the interval $[0, 1)$. The second form illustrates the geometric series of weights.

Two aspects of (4.1) help to reduce the time necessary to compute traffic estimates. First, an endpoint need maintain only a single datum, the traffic estimate for the current poll, in order to calculate a new estimate after the next poll operation. Second, we base the EMA on message arrivals at discrete poll operations rather than on actual arrival rates (messages per second, for example). Incorporating time into the EMA entails calculating powers of the damping factor and hence requires at least an additional table lookup. Clock routine calls can also be quite expensive. Except at points of synchronization, we expect the impact of ignoring time to be minimal. Polls occur reasonably uniformly in time, particularly at the scale implied by a high damping factor.

We further optimize the estimate calculation by restricting the damping factor α to be a rational number, and in particular to take the form $(d-1)/d$ for some $c \in \mathbb{Z}$, $c \geq 0$, $d = 2^c$. Rewriting (4.1) with the damping parameter d, and assuming standard integer arithmetic, we find:

$$t_i = \left\lfloor \frac{(d-1)t_{i-1} + n_i}{d} \right\rfloor \tag{4.2}$$

The multiplication by $(d-1)$ requires only a shift and a subtraction, and the division by d requires only a shift. However, the loss of accuracy due to the integer operations in (4.2) is unacceptably severe. A single poll operation rarely receives more than a few messages, yet some n_i must exceed d in order to raise the estimate above zero. We thus maintain a fixed-point rather than an integral value for traffic estimates. For b-bit precision, we increase n_i in (4.2) by a factor of $a = 2^b$ (with $d < a$) to obtain the fixed-point form:

$$t_i = \left\lfloor \frac{(d-1)t_{i-1} + a n_i}{d} \right\rfloor \tag{4.3}$$

The multiplication by the accuracy parameter a again requires only a shift. In the text, we continue to discuss the actual, integral values of traffic estimates rather than the values represented; although rarely relevant to the discussion, the values represented are a factor of a smaller. Using (4.3), we expect our traffic estimate to settle near $a\bar{n}$ when each poll receives, on average, $\bar{n}$ messages.

4.2.2 Multiple-poll estimation

The traffic estimate calculated by (4.3) relies on the availability of n, the number of messages received by each poll operation. However, by skipping the check for remote messages during some polls, we artificially shift messages to the poll that next checks the network, forcing the n^{remote} to zero for skipped polls. We must hence extend our definitions to handle the case in which the number of message arrivals is known only as an average over many polls. Consider the reception of n messages during the i^{th} poll, and assume that the previous $s-1$ polls did not check for arrival. If we assume that messages arrived at a uniform rate of n/s with respect to the poll operations, we can replace α with $(d-1)/d$ in (4.1) and apply the result s times to obtain:

$$\begin{aligned} t_i &= \frac{1}{d}\frac{n}{s} + \frac{d-1}{d}\left(\frac{1}{d}\frac{n}{s} + \cdots + \frac{d-1}{d}\, t_{i-s}\right) \\ \text{or} \qquad t_i &= \left(\frac{d-1}{d}\right)^s t_{i-s} + \left[1 - \left(\frac{d-1}{d}\right)^s\right]\frac{n}{s} \end{aligned}$$

The result has the same form as the original equation, but the damping factor has been raised to the s^{th} power. A similar form arises with time-based EMA's.

A table lookup can approximate the s-poll damping factor fairly well, but such precision is unnecessary. We avoid adding indexing and a memory reference to the critical path by approximating the damping factor with a linear function of s: $1-s/(2d)$. For large d, the linear approximation is greater than the exact function in the interval $(0, 1.5936d)$. Our system skips at most d polls, effectively increasing damping for the network protocol. As the time scale of remote traffic is longer than that of local traffic, the additional smoothing has little adverse effect.

Applying optimizations similar to those used for the single-poll case, we arrive at the following equation for traffic estimates:

$$t_i = \left\lfloor \frac{(2d-s)t_{i-s} + a n_i}{2d} \right\rfloor \tag{4.4}$$

This equation requires an integer multiplication, but need be performed only after a poll operation that checked for remote messages.

4.2.3 Poll skip selection

We have now defined the traffic estimate equations for each protocol and can turn to the heart of the adaptive strategy, determining the number of polls s to skip between checks for remote messages. As a first attempt, we choose s such that the system polls when the traffic estimates predict the arrival of a message. Recall that the number of messages accepted cannot exceed the number of polls. The use of two protocols does not change this fact, hence $t^{local} + t^{remote} \leq a$. After checking the network for messages, the expected number of polls before another remote message arrives is thus a/t^{remote}, and we write:

$$s = \frac{a}{t^{remote}} \tag{4.5}$$

Calculation of (4.5) requires an integer division after each poll that examines the network. We can elide a check for division by 0 in calculating s if we ensure that t^{remote} converges to a small but non-zero value when the network remains quiescent for an extended period. Let t_∞ be a value of t that remains stable across iterations of (4.4) when no messages arrive. Then t_∞ must satisfy:

$$\begin{aligned} t_\infty &= \left\lfloor \frac{(2d-s)t_\infty}{2d} \right\rfloor \\ 2d\ t_\infty \leq\ t_\infty(2d-s) &< 2d(t_\infty + 1) \\ -\frac{2d}{s} < t_\infty &\leq 0 \end{aligned}$$

t is thus stable in the interval $(-2d/s, 0]$ when no messages arrive. If $s \leq 2d$, all terms in the numerator on the right hand side of (4.4) are positive, and t stops at 0 when approaching from above. By adding s to the numerator, we shift this point of convergence to 1. Thus, we obtain as the final form of the network traffic estimate:

$$t_i^{remote} = \left\lfloor \frac{(2d-s)t_{i-s}^{remote} + s + an_i^{remote}}{2d} \right\rfloor \tag{4.6}$$

The strategy defined by (4.5) ignores the estimate of local traffic, an approach that works well when local traffic is frequent. When both protocols are quiescent, however, the strategy still selects large values for s, unnecessarily penalizing responsiveness to remote traffic. We adjust the approach by adding a factor of t^{local}/a, with the result:

$$s = \frac{t^{local}}{t^{remote}} \tag{4.7}$$

This factor couples the aggressiveness of remote message polling to the local traffic estimate. When local traffic is infrequent, the adjusted strategy polls the network aggressively. But when local traffic is frequent, the new strategy retains nearly the same character as the original strategy, polling only when a message is expected. In the latter case, the remaining difference between the two strategies, a minor reduction in the number of skips from the old to the new, reduces the lag time when remote traffic increases by checking for messages before they might otherwise be expected.

As a final adjustment to the strategy, we might wish to poll the network less frequently than shared memory even when traffic levels are similar. This adjustment trades overhead between the protocols, reducing overhead for local messages at the expense of remote messages. Using an equality parameter k to define the number of network polls to skip when observing equal levels of traffic, we write the final form of our local traffic estimate:

$$t_i^{local} = \left\lfloor \frac{(d-1)t_{i-1}^{local} + kan_i^{local}}{d} \right\rfloor \tag{4.8}$$

If we choose k to be a power of 2, we need merely shift n_i by a different amount when estimating local traffic.

4.2.4 Poll skip restriction

As the last step in defining an adaptive strategy, we bound the number of network polls skipped at both ends:

$$s_{min} \leq s \leq s_{max}$$

The parameter s_{min} must be small enough so as not to degrade performance on a NOW, yet large enough to prevent overly frequent network polls from degrading local message performance on a Clump. Analogously, the parameter s_{max} must be large enough to so as not to degrade performance on an SMP, yet small enough to prevent overly infrequent network polls from degrading network performance on a Clump.

4.3 Parameter Tradeoffs

The adaptive polling framework optimizes the process of deciding when to poll for remote messages but parametrizes values for which an intuitive choice might not be correct.

Parameter	Symbol	Value	Description
acceptance count	N/A	4	default number of messages accepted by a poll operation
accuracy	a	4,096	fractional component of traffic estimates (12 bits)
damping	d	256	damping factor for inclusion of new message arrival counts into traffic estimates
equality	k	4	number of network polls skipped when observing equal levels of traffic on the two protocols
minimum skip	s_{min}	4	lower bound on the number of polls skipped before checking the network
maximum skip	s_{max}	64	upper bound on the number of polls skipped before checking the network

Table 4.1: Polling strategy parameters and values. We chose the first two parameters without measurement. We selected all others based on application performance data.

In this section, we discuss the tradeoffs associated with each parameter, establishing our rationale for selecting the actual values through empirical performance studies.

An adaptive strategy extends the naive poll operation with traffic calculations and a remote polling decision. In particular, the revised poll operation begins by counting down from the last skip count s to decide whether or not to check for remote messages. The operation next handles up to four local messages, then either skips network polling or handles up to $4s$ remote messages. The poll operation keeps track of the number of messages received with each protocol for use in revising the traffic estimates. At the end of the poll operation, a process uses (4.8) to update the local traffic estimate. If the poll checked the network, the process also updates the remote traffic estimate using (4.6) and selects a new value of s with (4.7), adjusting the new value to fall in the interval $[s_{min}, s_{max}]$.

Including the acceptance count, an adaptive strategy is determined by a total of six parameters, as shown in Table 4.1. The first two parameters, acceptance count and accuracy, admit reasonably intuitive choices, whereas the other four require measurement. We discussed the issues for acceptance count in detail in Section 4.1.1 and mentioned our choice of four messages per poll. Accuracy needs to be both large and small enough to prevent the loss of information through roundoff and overflow, respectively. We chose 12-bit accuracy for our system.

The damping parameter d determines the degree of smoothing in the traffic estimates. Equivalently, d determines the scale at which we estimate traffic. At $d = 16$, for example, the last 35 polls contribute at least 10% as much to an estimate as does the current poll. At $d = 128$, the last 293 polls meet this criterion. A high damping parameter thus smooths bursty arrival rates and reduces the discrepancy between arrivals measured in time and arrivals measured by the number of intervening poll operations. A overly large damping parameter, however, prevents the layer from reacting quickly to actual changes in the communication pattern: when local traffic transitions from quiescent to a uniform arrival rate at $d = 1024$, the system requires $2,357$ poll operations to bring the traffic estimate to 90% of the arrival rate.

The equality parameter k determines the frequency of network polling when the traffic estimates are roughly equal. A low value of k is fair in the sense that the layer is equally responsive to both protocols. However, equal levels of traffic do not imply that most polls receive messages, and a low value of k may still adversely affect performance by checking the network too frequently. A high value of k circumvents this problem by eliminating network polls, but may result in a layer that needlessly delays remote messages whenever any traffic uses the shared memory protocol.

The minimum skip parameter s_{min} places an upper bound on the frequency of network polling. In a manner similar to the equality parameter, a high value of s_{min} prevents the communication layer from performing too many empty network reads, but does so without regard to estimated levels of traffic. When s_{min} is too high, however, this insensitivity to traffic degrades remote message performance.

The maximum skip parameter s_{max} places a lower bound on the frequency of network polling. A high value of s_{max} allows the communication layer to ignore the network under appropriate traffic conditions. When s_{min} is too high, however, the layer checks the networks so rarely that it may fail to recognize changes in the traffic patterns.

4.4 Summary

We have outlined an adaptive strategy that dynamically adapts the frequency of polling within a communication layer to changing patterns of message traffic. This strategy allows our multi-protocol layer to provide high levels of performance on a range of Clump architectures. Included in our explanation of the strategy are six parameters designed to expose the effects of the underlying hardware and of the operating system on a communication layer's performance. For a general Clump built from a common hardware and software base, the strategy need be tuned only once to provide adequate performance on all platforms.

This chapter completes our description of the multi-protocol communication architecture. In the next few chapters, we address performance, beginning with our methodology and hardware parameters in Chapter 5 and continuing with shared memory protocol performance in Chapter 6. We return to the adaptive strategy and tuning its parameters for Clump performance in Chapter 7.

Chapter 5

Experimental Methodology

We have completed our description of the architecture of our multi-protocol communication layer and now investigate its performance. In this chapter, we present our methodology, including the details of each experiment, the strategy for collecting results, and the style of reporting data. The common methodology provides a necessary framework for interpreting the measurements reported in the next two chapters. This chapter begins with an explanation of our approach for aggregating and reporting data, then introduces the Clump architecture used to measure performance. We report the important hardware parameters for our experimental platform, which aid in evaluating and interpreting the level of performance delivered by our layer. We next describe our suite of performance benchmarks for passing messages through shared memory. These benchmarks range from simple pairwise exchanges of data between two dedicated processors to multiple copies of full applications competing for cycles. A subset of these benchmarks also serves to measure performance on a Clump.

The next two chapters present the results of our experiments and our interpretation of the data. In Chapter 6, we focus on the shared memory protocol, comparing the lock-free algorithm with alternatives and exploring the performance of communication across a memory interconnect. Chapter 7 addresses performance on Clumps, including the tuning of the adaptive polling strategy and a comparison with performance on a NOW.

5.1 Standard Methodology

Measurements of computer systems are rarely precise. Seemingly random interactions with complex hardware and software systems couple with synchronization races to produce timings that vary widely between executions. Researchers often address this problem by averaging over a few runs, but a simple mean is not in general adequate to allow comparisons between experiments or to explain performance effects. Without some knowledge of variability, the fact that the measured mean of A is greater than the measured mean of B implies only that B is probably faster than A on average. The difference between that implication and an unbiased coin flip may be vanishingly small, however. We employ more effective approaches to gathering data and include measures of variability when reporting results. In the majority of our experiments, we employ a common methodology. This section begins with a short discussion of interpreting units, then describes our *standard methodology* and comments on its appropriateness for our measurements.

Nearly all of our results include units of time or data. All time measurements utilize wall-clock time, as given by the Solaris `gethrtime()` call. This function has a granularity of 16 cycles on our machines and an overhead of roughly 0.7 microseconds when separated by at least one 16-cycle tick. We report units of data in terms of bytes (B), kilobytes (kB), or megabytes (MB), depending on the magnitude of the measurement. In all reported values, the relationship between these units is defined in terms of standard computer notation, *not* metric notation. For example, 1 MB is equivalent to 1,048,576 B, not 1,000,000 B.

When using our standard methodology, we repeat an experiment one hundred times to minimize the impact of random error on our results, then report three values: the arithmetic mean of the repeated experiments, the 95% confidence interval on the reported mean, and the standard deviation of the repeated experiments. For measurements of a variable V, with experiments ranging from 1 to $N = 100$, we calculate the mean $\bar{V}$, the standard deviation σ_V, and the confidence interval Δ_V as follows:

$$\begin{aligned}
\bar{V} &\equiv \frac{1}{N}\sum_{i=1}^{N} V_i \\
\sigma_V &\equiv \left[\frac{\sum_{i=1}^{N}\left(V_i - \bar{V}\right)^2}{N-1}\right]^{\frac{1}{2}} \\
\Delta_V &\equiv \frac{2\sigma_V}{\sqrt{N}} \qquad (5.1)
\end{aligned}$$

We select a multiplier of 2 for simplicity, hence our confidence intervals are slightly higher than 95% when V is normally distributed. After calculating these values at the precision returned by the experiment, we round the confidence interval up to the nearest value with one significant digit. We then report the mean and standard deviation to the precision specified by the confidence interval, rounding off the values beyond that point. We do not report confidence intervals on standard deviation. We typically present results in the following format:

$$\begin{array}{cc} V \pm \Delta_V & 3.065 \pm 0.008 \\ \sigma = \sigma_V & \sigma = 0.036 \end{array}$$

Although our definitions reflect methods typically employed in the statistical analysis of experimental data, we must be careful in interpreting our results in that fashion. The traditional notion of a confidence interval applies only when each measurement is independent of all others and all measurements are drawn from identical probability distributions. With computer systems, neither property necessarily holds.

Separate measurements often interact with common software subsystems, making later measurements dependent on the state changes induced by earlier measurements. Common sources of dependency include physical memory layout, filesystem cache contents, and other operating system resources. Hardware cache alignment and contents also play a role for multiple measurements made within the context of a single job.

A measurement's probability distribution also depends fairly heavily on the state of the machine. Certainly an unrelated process on the same machine can change the result, either directly by claiming a processor for some amount of time or indirectly through cache pollution or contention for memory or network access. Other computers can also affect the results by interacting with the measured machine through the network or by generating network traffic that contends with measured traffic inside the network.

Scrubbing the machine of unrelated processes and eliminating the possibility of external interference reduces or removes most of these effects, but some persist. In particular, certain effects are random across reboots but persistent after any given boot. These effects cannot be treated as random factors without rebooting the system for every measurement. Their magnitudes—a few tens of cycles, or a fraction of a microsecond—tend to be smaller than the magnitudes of random effects for many measurements, but their systematic nature limits the accuracy of our results. We note the impact of these effects when discussing our data in Chapters 6 and 7.

5.2 Hardware Architecture

A prototypical Clump consists of symmetric multiprocessors (SMP's) connected by a fast network. In studying communication, we focus on the interconnection hierarchy between processors. Within each SMP, a backplane connects processors and main memory. Between SMP's, processors communicate via fast network devices attached to I/O buses on the SMP backplanes. We begin this section with a description of the Clump used in our performance study, then introduce a useful tool for low-level measurements: a cycle-exact timing loop. We next discuss the system parameters that play important roles in both inter- and intra-SMP communication and present the values of these parameters for our Clump architecture. The section concludes with a summary of the important aspects of the architecture.

We obtained empirical results for a cluster of four a Sun Enterprise 5000 servers, as shown in Figure 5.1. Each server contains eight 167 MHz UltraSPARC processors and 2 GB of main memory. Sun's Gigaplane Interconnect [SBC+96] connects the processors and memory. This high-performance bus is representative of many modern cache-coherent system interconnects; its most unusual characteristic is support for more than one outstanding transaction on a single cache line, effectively pipelining concurrent memory traffic between processors. The Gigaplane implements an extended version of the MESI invalidation protocol [PP84] to maintain coherence between processor caches and main memory. For atomic transactions, the Sparc V9 instruction set supports both the universal [Her88] COMPARE&SWAP primitive (CAS) and the less powerful TEST&SET (T&S) and SWAP primitives. The interconnection between SMP's consists of a Myrinet network [BCF+95] with four independent links to each SMP. The links support 160 MB/sec of bidirectional bandwidth, but the SBUS I/O bus and network communication software limit the bandwidth realized by applications to around 31 MB/sec. Four Myricom Lanai 4.1 M2F eight-way crossbar switches comprise the network fabric, which thus has a diameter of three hops and a bisection bandwidth of 640 MB/sec.

For comparison, we also obtained a subset of our performance measurements on a cluster of thirty-two UltraSPARC Model 170 workstations, each with 128 MB of main memory. This NOW uses the same Myrinet network cards for interprocessor communication, but each workstation employs only one SBUS and one Myrinet network interface card. The network fabric is also the same, but the topology is similar to a fat-tree [MCSW97].

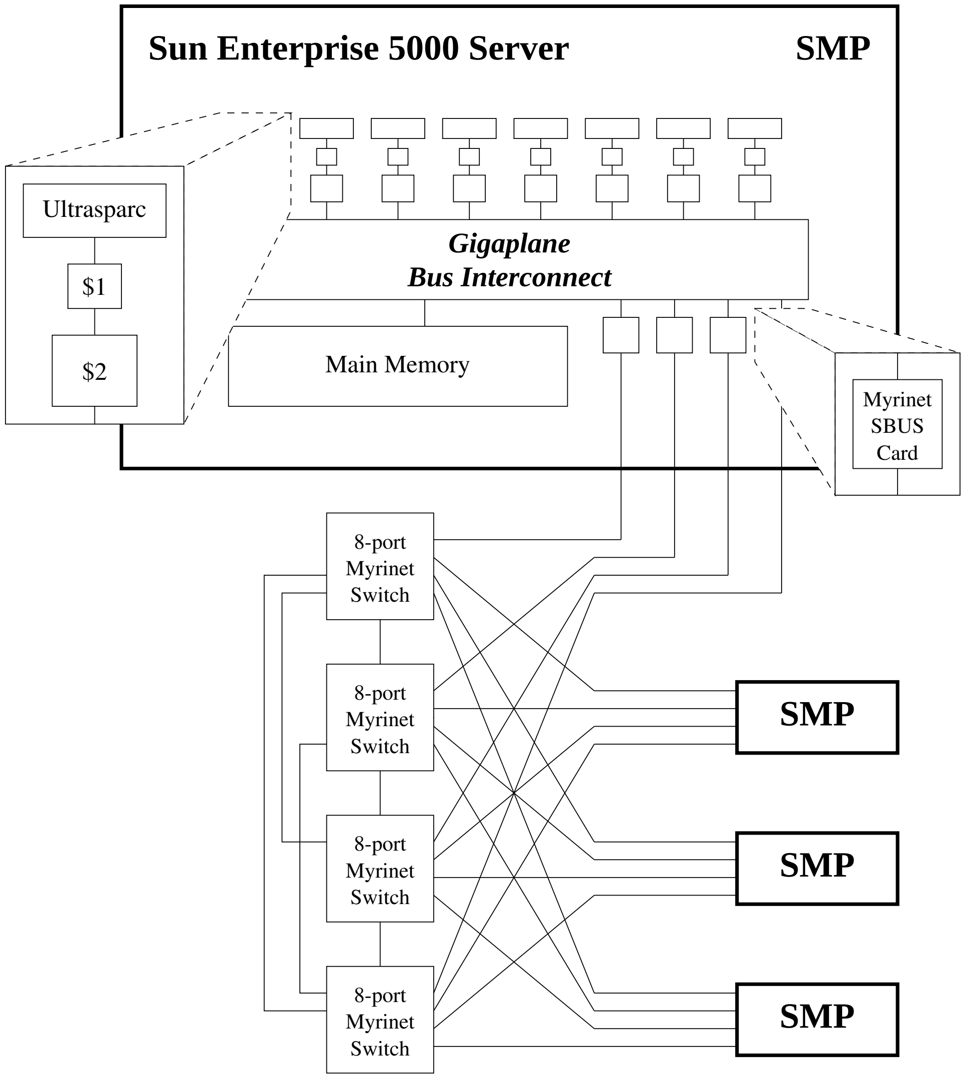

Figure 5.1: Target Clump architecture. Four Sun Enterprise 5000 servers with eight 167 MHz UltraSPARC processors. Each SMP uses four independent SBUS's to communicate through a Myricom Myrinet network.

```
1: cmp   %0,0
   nop
   bg    1b
   dec   %0
```

Figure 5.2: Two cycle per iteration loop for Sparc V9. The C compiler selects a register, %0, in which to pass the cycle count (divided by two) into the loop. The label, "1:," is local to each instance of the loop, and the branch target, "1b," refers to the last instance of the label 1 in the backwards direction.

5.2.1 Timing loops

As a convenient mechanism for generating accurate delays for benchmarks and backoff operations, we developed a loop that executes in an exact number of machine cycles. Although such a loop at first appears trivial to construct, we must recognize some subtle effects of instruction cache alignment and superscalar issue.

The simplest approach involves a register decrement loop, and indeed these loops can suit our purpose, as shown in Figure 5.2. On a Sparc, optimizing compilers produce loops that usually execute in one cycle per iteration. However, if such a loop falls across a cache line boundary, the instructions can not all be issued in the same cycle, and the loop executes more slowly. We considered aligning the loop, but the Sparc assembler pads intervening words with illegal trap instructions (opcode 0) and thus requires that we branch to aligned code. By inserting an additional nop instruction, we create a loop that requires two cycles per iteration regardless of cache alignment, as shown in the figure.

The cycle count, or TICK, register presents an alternative approach, but the relationship between the TICK register on distinct processors is undefined, and migrating from one processor to another can produce odd and incorrect results, *e.g.*, negative timings. Also, TICK accesses are privileged by default, and user-level accesses generate exceptions. These considerations make a TICK-based approach less attractive than our register-based approach.

5.2.2 Cache parameters

Two aspects of hardware performance are critical to shared memory message-passing: the memory hierarchy and the cost of synchronization primitives. The memory hierarchy, and in particular the parameters of L2 cache, governs the rate at which data moves from one processor to another; synchronization primitives impose the basic overhead for concurrent access to data by multiple processes. We first investigate three aspects of the memory hierarchy: the access times for single processors, the cost of cache-to-cache transfers, and the bandwidth achieved in copying memory between buffers. We time the relevant synchronization primitives in Section 5.2.3.

Sequential access times

We measure the relevant aspects of the memory hierarchy with microbenchmark techniques pioneered by Saavedra-Barrera [Saa93]. The approach is fairly simple: for an array of a given size, we loop repeatedly through the array, reading elements at a given stride. We time a total of sixteen million accesses and calculate the time per access to within a cycle. We then graph these access times for a range of array sizes and strides, as shown in Figures 5.3 and 5.4. The first of these figures represents the results on an Enterprise 5000 server, while the second represents the results on an UltraSPARC Model 170 workstation. Our main goal in presenting these results is to determine the L2 cache parameters that limit the performance of the shared memory protocol. However, we present a more complete evaluation of the two memory hierarchies to illustrate the similarity between the architectures, an aspect that lends weight to our use of the Model 170 as the building block for the NOW against which we compare Clump performance.

The features of Figure 5.3 provide information about the memory hierarchy in the Enterprise 5000. The lowest group of lines (8 and 16 kB arrays) lies at 2 cycles and represents 1 cycle of L1 hit time and 1 cycle of overhead from the inner loop. Only for L1 hits does this inner loop overhead appear; in all other data, it is hidden by the access time. The lines begin to rise past 256 B due to overhead from the outer loop, which includes inner loop initialization. The reduced access time when the stride equals the array size probably represents a combination of better branch prediction for the inner loop and relaxation of dynamic dependencies between accesses, which are all destined for the same register. The separation between the lines for 16 and 32 kB arrays also indicates the size of the L1 cache: 16 kB.

The next group of lines includes arrays from 32 to 256 kB. The exponential rise from 4 to 16 B strides reflects the number of L1 cache misses; the access times stop rising at 16 B, when each access incurs a miss. The L1 line size is actually 32 B, but the cache uses

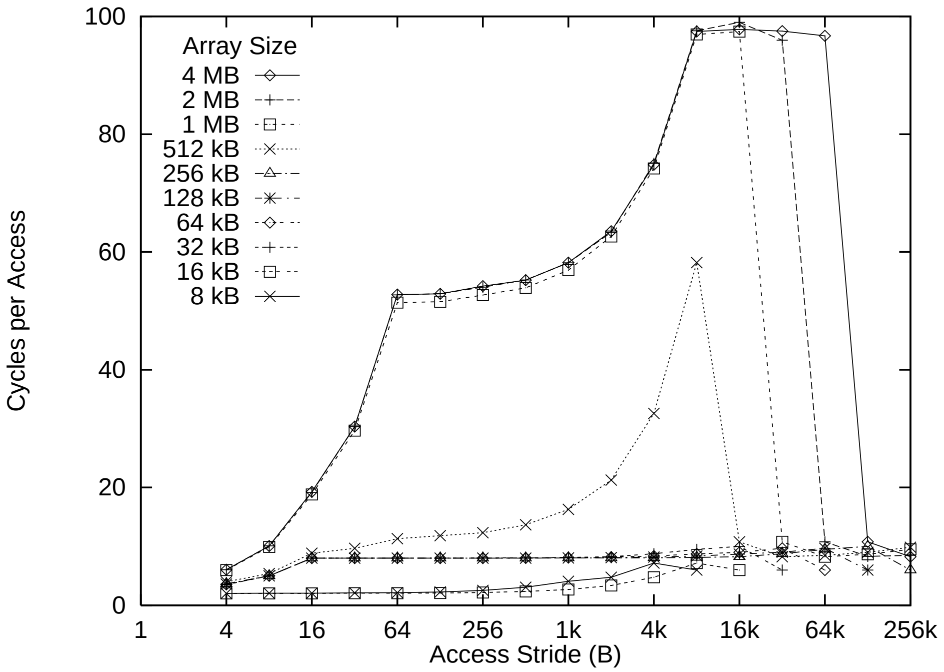

Figure 5.3: Enterprise 5000 server memory access graph.

16 B subblocks. The Sparc V9 can pipeline data from the L2 cache into the L1 cache, but our use of a single destination register in the loop prevents this capability from affecting the graph. The plateau at 8 cycles thus reflects the L2 cache hit time. Note again the effects of outer loop overhead on the right side of the figure.

Skipping the line representing a 512 kB array, we next consider the group of lines for 1 to 4 MB arrays. In a manner similar to that displayed by the previous group, these lines rise exponentially from 4 to 64 B as more accesses incur L2 cache misses. The times stop rising at 64 B, the L2 line size, and we read the main memory access time as 51 cycles. Past 64 B, the lines immediately begin a similar increase due to translation lookaside buffer (TLB) misses. The latter trend stops at a stride of 8 kB, the virtual memory page size, incurring a 97 cycle delay to handle both misses. For very large strides, and in particular when the loop accesses no more than 32 elements, the access times for these arrays drops back down to the L2 miss time. Such a drop generally reflects some kind of associativity, probably in this case through a fully-associative, 32-entry victim cache. Both the L1 and the L2 caches in this architecture are direct-mapped.

The last line, which represents a 512 kB array, mingles the character of the previous

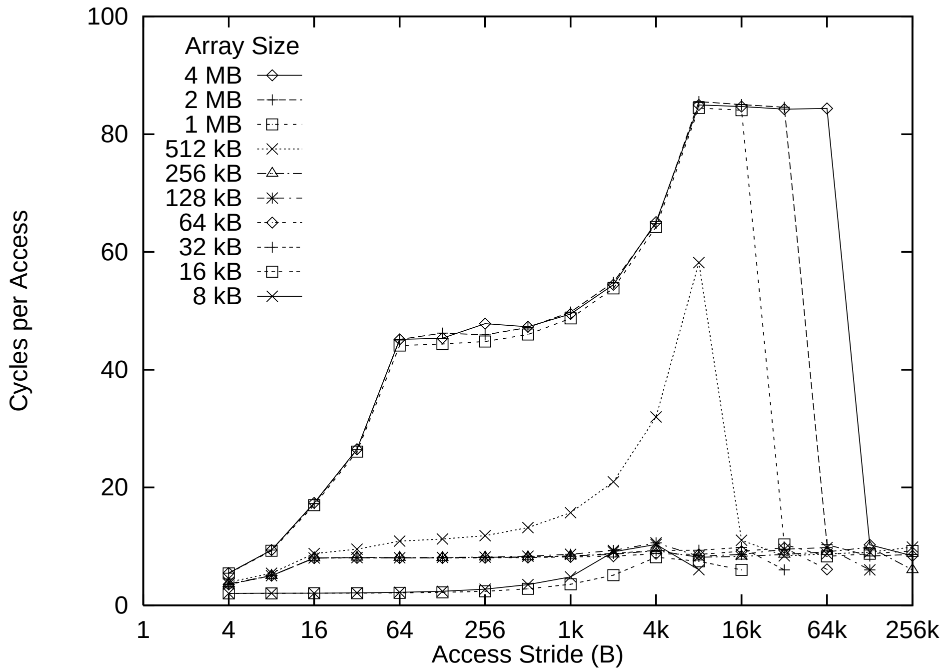

Figure 5.4: UltraSPARC Model 170 workstation memory access graph.

two groups to include TLB misses without L2 misses. Recalling the page size of 8 kB, we thus conclude that the data TLB has 32 entries and that the L2 size is 512 kB. The processor's data TLB actually has 64 entries [SME97], but the software TLB miss handler reserves some of the entries for the operating system.

Now consider Figure 5.4, which presents the results of the same experiment on an UltraSPARC Model 170 workstation. As reflected by the lower portions of the figures, the upper levels of the hierarchy are identical to that of an Enterprise 5000 (with the same processor). This similarity allows a more meaningful evaluation of the performance of our Clump compared with a NOW than is possible with a more disparate workstation architecture. Memory accesses are slightly faster, requiring only 44 cycles, and combined L2/TLB misses require a total of 84 cycles.

Parallel access times

A similar benchmark enabled us to time the cost of moving a cache line between caches. Two processors take turns walking through a shared array the size of the L2 cache and writing one word to each cache line. The last word in the array serves as the handshake

	Enterprise 5000	UltraSPARC Model 170
L2 size	512 kB	512 kB
L2 line size	64 B	64 B
L2 miss (memory)	51 cycles	44 cycles
L2 miss (other L2)	84 cycles	N/A
memcpy bandwidth	184 MB/sec	168 MB/sec

Table 5.1: Level 2 cache parameters.

signal between the processors. One processor fills the array with ones and waits for a flag value of zero before beginning again, and the second processor fills the array with zeroes and waits for a flag value of one. We time ten thousand alternating array walks, subtract the loop overhead, and calculate the cost of bringing a single cache line from a remote cache into a modifiable state in the local cache. Two boundary effects arise—the latency-hiding ability of the L2 write buffer and the potential for extra coherence transactions on the flag value—but these effects oppose each other and are amortized over the number of cache lines (8,192 in our system). On the Enterprise 5000, the cache-to-cache transfer takes about half of a microsecond, or 84 cycles of the 167 MHz processors.

Memory copy bandwidth

Lastly, we measured the memory copy bandwidth available on each platform. Sparc V9 implementations provide a block-transfer engine to support multimedia applications through the VIS instruction set extension. This engine, which supports transfers with arbitrary alignment, is also very useful for memory copy operations. We measured individual 8 kB memory copies to reflect the expected performance of memory copies for bulk transfers within our communication layer. We carefully measured and subtracted timer overhead to obtain more accurate results, but still observed variations of two to three megabytes per second. The results appear in Table 5.1 along with a summary of the cache hierarchy information relevant to interprocessor communication. The Enterprise 5000 delivers roughly 184 MB/sec of memory copy bandwidth, significantly more than the 168 MB/sec delivered by the workstation.

Primitive	Enterprise 5000 & UltraSPARC Model 170	
	Completion	Stall
TEST&SET	15 cycles (90 nsec)	2 cycles (12 nsec)
SWAP	15 cycles (90 nsec)	2 cycles (12 nsec)
COMPARE&SWAP (success)	16 cycles (96 nsec)	3 cycles (18 nsec)
COMPARE&SWAP (failure)	14 cycles (84 nsec)	3 cycles (18 nsec)

Table 5.2: In-cache synchronization primitive timings. Stall time cannot be overlapped with other instructions. Data become available after the completion time.

5.2.3 Synchronization primitives

The cycle loop described earlier provides an effective mechanism for isolating the execution time of single instructions, and in particular an architecture's synchronization primitives. The Sparc V9 instruction set provides three synchronization primitives: CAS, T&S, and SWAP. We insert each primitive into a loop of ten million iterations, then time the loop with high-resolution timers. Overhead from the timer call, scheduler interactions, and other interrupts never comes close to ten million cycles (a one second loop incurs roughly ten thousand cycles of overhead), hence this approach allows us to time loop iterations with cycle accuracy. By adding `nop`'s and timing the loop with and without the synchronization primitive, we can further determine what portion of the completion time for a synchronization primitive can be overlapped with other operations. The remaining time is the stall time for the primitive. We report the measured values in Table 5.2. Completion times for all primitives are roughly twice the L2 hit time; these instructions are likely processed in the L2 cache. CAS requires two extra cycles (one bus cycle) when successful. Stall times are two or three cycles.

5.2.4 NIC parameters

We also use the cycle loop to isolate the cost of loading from and storing to memory on the network interface card. These parameters govern the cost of programmed I/O to the NIC and are the base costs for the network protocol. In particular, the network poll operation performs two reads when polling an empty queue. The techniques are the same as those used for measuring the synchronization primitive timings. In an otherwise empty loop, reads destined for the same register block at the next read until the data arrive. Writes typically go into the write buffer and do not delay the processor unless the write

Operation	Enterprise 5000	UltraSPARC Model 170
32-bit read	153 cycles (918 nsec)	114 cycles (684 nsec)
32-bit write	53 cycles (318 nsec)	33 cycles (198 nsec)
64-bit read	160 cycles (960 nsec)	122 cycles (732 nsec)
64-bit write	62 cycles (372 nsec)	40 cycles (240 nsec)
I/O bus bandwidth	4x31 MB/sec	31 MB/sec

Table 5.3: NIC memory access timings. Read latencies are typically exposed, while write latencies are typically hidden by the write buffer. Bandwidths reflect transfer rates achieved by AM-II.

	Enterprise 5000	UltraSPARC Model 170
Number of processors	8	1
Clock speed	167 MHz	167 MHz
Processor type	UltraSPARC V9	UltraSPARC V9
L2 size	512 kB	512 kB
L2 line size	64 B	64 B
L2 miss (memory)	51 cycles	44 cycles
L2 miss (other L2)	84 cycles	N/A
memcpy bandwidth	184 MB/sec	168 MB/sec
NIC 32-bit read	153 cycles	114 cycles
NIC 32-bit write	53 cycles	33 cycles
SBUS bandwidth	4x31 MB/sec	31 MB/sec
COMPARE&SWAP	15 cycles	15 cycles

Table 5.4: Summary of memory, network, and synchronization primitive parameters for our Enterprise 5000 servers and UltraSPARC Model 170 workstations.

buffer overflows. To measure writes, we perform a write followed by a read, each dependent on the other, then subtract the cost of the read alone.

Both the read and the write latency can be fully hidden by other work, so long as that work does not introduce dependencies or access the memory system. Surprisingly, the cost of traversing the more powerful interconnect is much more prominent in NIC access times that it is for memory accesses. An Enterprise requires 153 cycles to complete a 32-bit read, whereas the Model 170 delivers the data in 114 cycles. The absolute difference for 32-bit writes is about 20 cycles, and 64-bit operations have roughly the same absolute differences, and shown in Table 5.3. Bus bandwidth, as measured in one direction with AM-II, is roughly 31 MB/sec on both platforms.

5.2.5 Summary

We have presented those parameters of our experimental platform that directly impact issues of communication. Table 5.4 summarizes the important parameters. Both the design and the performance of the shared memory protocol depend on the memory hierarchy and synchronization primitives, and NIC memory access times play a major role in network protocol performance. The Model 170 parameters in themselves provide an interesting example of the impact of using a more powerful memory interconnect on access times. One must be careful not to read too much into the numbers in this sense, however. Temporal and spatial locality amortize the 16% increase in memory latency, and, in our experience, application performance rarely reflects this difference. More importantly, the workstation parameters support our efforts to compare the performance of Clumps and NOW's by demonstrating the similarities between our experimental platforms.

5.3 Performance Benchmarks

We are now ready to discuss our metrics of performance, which we cast as a benchmark suite for shared memory message-passing. Concurrent queue algorithms are often tested in isolation, with enqueue-dequeue pairs in a tight loop, for example. Although such microbenchmarks can be instructive for comparing the dynamic length of mutually exclusive critical sections, they tend to artificially inflate per-processor contention and fail to capture the performance of the algorithms as used in practice. A queue serves abstractly to move items from a group of producers to a group of consumers. These items typically take time to produce and time to consume, whereas an enqueue-dequeue pair loop examines only the limit in which both production and consumption times go to zero. Applications intended to scale beyond a handful of processors neither use single, centralized queues nor enqueue items that require trivial amounts of work to consume. Furthermore, higher-level abstractions such as a message-passing system typically add work to the encapsulated enqueue and dequeue operations without extending the length of critical sections.

In this section, we adapt notions from the literature to the problem of evaluating the performance of message-passing based on concurrent queues in shared memory. We develop a benchmark suite that covers the range of contention, from uncontended access using a single sender-receiver pair to maximal contention using many-to-one communica-

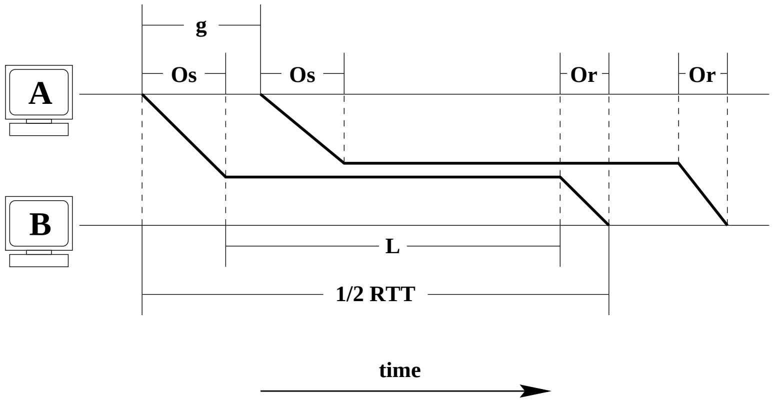

Figure 5.5: LogP communication model. Send overhead o_s and receive overhead o_r require processor cycles, whereas latency L across the interconnect does not. Round-trip time RTT is twice the sum of those three parameters. The gap g shown in the figure represents the per-message average for long bursts; in most actual systems, a burst of two messages requires only $2o_s$.

tion. We also present a small set of applications with varying communication patterns to measure the performance of a queue algorithm in actual use. Finally, we describe tests using the applications to measure the ability of a given algorithm to operate on a multiprogrammed machine. In Chapter 6, we compare the performance of a variety of concurrent queue algorithms with that achieved by our lock-free algorithm. We also utilize a subset of these benchmarks to measure performance on a Clump, and present the results of those experiments in Chapter 7.

5.3.1 Communication parameters

The simplest parametrization of a communication system describes the time required for communicating between a single sender-receiver pair. The LogP model [CKP+93] and its successor, LogGP [AISS95], capture the components of fast, user-level communication quite effectively. Assuming a small, fixed message length, the LogP model characterizes communication networks as a set of four parameters: L, an upper bound on the network latency (wire time) between processors; o, the processor busy-time required to insert a message into the network or to accept one; g, the minimum time between message insertions for large numbers of messages; and P, the number of processors. Descriptions of actual communication systems often separate the overhead o into send and receive overheads, as depicted in Figure 5.5. The send overhead, o_s, is the processor busy-time for message insertion, and

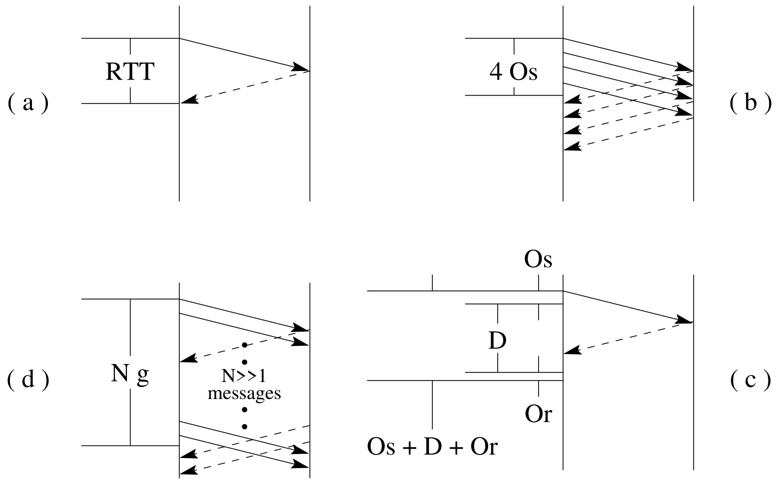

Figure 5.6: LogP microbenchmark methodology. Descriptions in the text proceed clockwise from the upper left. Round-trip time RTT, send overhead o_s, and gap g are measured directly. Receive overhead o_r is found by measuring the sum of o_s, o_r, and a fixed delay D, as shown in (c). Latency (L) is calculated as $\frac{RTT}{2} - o_s - o_r$.

the receive overhead, o_r, is the processor busy-time for message reception. LogGP extends the LogP model with a characterization of long message performance, as captured by the G parameter, which gives the cost per byte for very long messages (*i.e.*, in the limit of infinite length).

We measure and report the LogP parameters for both communication protocols using a microbenchmark suite developed by Culler *et al.* [CLMY96]. We modified and extended this suite to improve the precision of the reported values and to measure the value of G. In general, each microbenchmark makes use of a single sender-receiver pair and sends bursts of varying length from sender to receiver. A burst is simply a set of messages sent back-to-back from the application level. We apply multiple stages of averaging to provide more accurate results by transforming the probability distribution for single measurements into a normal distribution. Typically, each microbenchmark run averages times from 100 bursts of 16,384 messages. Individual burst results are calculated by subtracting out timer overhead, as measured with an empty loop body, then dividing the resulting time by the number of messages sent to find an average time per message. We then average 100 microbenchmark runs to obtain our reported mean and confidence interval. We do not report standard deviations for these measurements, as they represent only the variance of the operations when aggregated into our fairly arbitrary loop lengths and averaging groups.

Several of the LogGP parameters allow direct measurement: round-trip time, send overhead, and gap. We measure the other parameters, including receive overhead and latency, indirectly. We now describe the exact methodology used to perform these measurements and comment on our rationale and on the accuracy achieved through our approach. Figure 5.6 illustrates each microbenchmark graphically.

Round-trip time

Round-trip time RTT represents the best-case time to send a request and to receive the corresponding reply, or twice the end-to-end, application-level latency. Measuring round-trip time is straightforward. We send 16,384 messages, waiting for the reply between each send operation as shown in section (a) of the figure, then apply two-stage averaging to obtain our reported value and confidence interval. After the first stage of averaging, the distribution measured by this microbenchmark depends on the ordered pair of processors being used: the mean round-trip between A and B differed by roughly 1% from the round-trip time between B and A, although each distribution appeared to be normal. The distribution of the sum of these two times, however, also appears to be normal. We thus average between the two directions and present results based on 50 trials rather than 100. We use a multiplier of 2.01 in Equation (5.1) to adjust for the reduced number of trials.

Send overhead

The send overhead o_s represents the amount of time that a processor must spend in the communication layer when sending a short message. For reasonably short bursts of request messages, neither flow control nor acceptance of the implied reply messages play a role in execution time for a sender: the total time is simply the number of messages in the burst multiplied by the send overhead, as shown in section (b) of Figure 5.6. We can thus determine send overhead by measuring the cost of inserting a relatively short burst of messages. During the message send loop, we delay the receiver so as to prevent receipt of replies by the sender.

To minimize the impact of errors in timer overhead measurement and random fluctuations, the sender enqueues as many messages as possible, generally bounded by the size of the receive queue. For the shared memory protocol, we shorten the queue to 1,024 entries to avoid covering the L2 cache with communication data. With the standard length of 4,096 entries, delaying the receiver has the side effect of flushing significant amounts of communication data from the receiver's L2 cache between timed insertion loops. As we desired to capture the difference between packets supplied from memory and packets supplied from the receiver's L2 cache on the send overhead, we decided to reduce the impact of this effect by using a shorter queue. The send overhead microbenchmark then fills these shorter queues using bursts of 1,024 messages. For the network protocol measurements, the queues are by design much shorter, and we send bursts of only 16 messages.

Two-stage averaging proves ineffective for send overhead with the shared memory protocol. The distribution for microbenchmark runs is bimodal and clearly not independent, as a sequence of timings within the same process shows a clear pattern of motion between the two modes. The difference between these modes—roughly 6 cycles—can probably be attributed to cache effects, but one must consider this non-independence in interpreting the results.

Receive overhead

The receive overhead o_r measures the time that a processor must spend in the communication layer when accepting a short message. We define receive overhead to be the cost of calling the poll function and receiving a single message. We measure this cost indirectly by measuring the sum of send overhead, receive overhead, and a known delay, then subtracting the two additional terms. As shown in section (c) of Figure 5.6, we measure the sum as follows: the sender sends a single message, delays for some time longer than the round-trip time, and calls poll. We then apply standard error propagation formulae to obtain our reported values. Denoting the measured sum by S and the delay by D, we calculate:

$$\begin{aligned} o_r &= S - D - o_s \\ \Delta o_r &= \left[(\Delta S)^2 + (\Delta o_s)^2\right]^{1/2} \end{aligned} \tag{5.2}$$

We have assumed in (5.2) that the random error on the delay D is negligible relative to the other terms. The cycle loop described earlier serves as the basis for the delay, taking the form of an aligned function with a correction factor for call overhead. Although the distribution of run timings appears normal, the delay incurred by the cycle loop function call fluctuates between two modes separated by 3 cycles for unknown reasons.

As these fluctuations tend to be correlated in time, we must treat them as systematic error, which limits the accuracy of our results. We report the mean at the mid-point between the modes and add 1.5 cycles into the confidence interval quadratically. The non-independence of the send overhead distribution must also be considered when interpreting the results of this microbenchmark for the shared memory protocol.

Gap

The gap g represents the average time required to insert a message for a long burst of messages, as shown in section (d) of Figure 5.6. It captures the difference between the cost of sending a few messages and the cost of sending many messages in rapid succession. The gap is generally at least as large as the maximum of the send and receive overheads. A sender cannot begin inserting a message until it has finished inserting the previous message, and the finite nature of a receiver's resources prevent a sender from exceeding the reception rate for very long bursts. The gap can exceed both overheads for other reasons as well. If flow control does not allow a sender to saturate the network, the gap rises. For communication systems with request-reply semantics, such as AM-II, a sender must alternate between sending requests and handling replies, and the gap is close to the sum of those two overheads. The distribution of run times for this microbenchmark appears normal, as desired.

Gap per byte

The gap per byte G represents the average time required per byte to insert a long message. G captures the effect of hardware support such as DMA for sending long messages; short messages typically make use of programmed I/O to avoid DMA startup costs, but long messages amortize these costs to achieve lower overheads and latencies than are usually possible with programmed I/O. We calculate G as the inverse of realized bandwidth for long messages. In particular, a sender fragments a 512 kB message into 8 kB blocks and passes them to a receiver in pipelined fashion. The receiver copies the messages into a receive buffer and acknowledges receipt of each block. We time the sender from sending the first block until receipt of the last acknowledgement and calculate realized bandwidth B by dividing 512 kB by the measured time. G is then given by:

$$
\begin{aligned}
G &= \frac{1}{B} \\
\Delta G &= \frac{\Delta B}{B^2}
\end{aligned} \tag{5.3}
$$

The realized bandwidth distribution appears roughly normal, but we incur a systematic error of roughly 0.7 microseconds by not accounting for timer overhead, which limits our accuracy for G to 0.7 μsec/512 kB, or, rounding up, 2 picoseconds/B. We add this error quadratically into the confidence interval before reporting results. For the shared memory protocol, the new term is typically dominated by the effects of random error, but it has the same magnitude as the random error for the network protocol.

For the shared memory protocol, we also present data on realized bandwidth as a function of message length, generalizing the gap per byte microbenchmark to serve this purpose. By varying the total message length, we measure the bandwidth achieved when sending messages of a given size and waiting for acknowledgement of receipt.

Latency

The latency L is roughly the time spent on the "wire," including any protocol processing overhead on a network interface card. In practice, we define L as the difference between half of the round-trip time and the sum of the send and receive overheads for a system. This definition implies that overlap in time between send and receive overheads, usually in the form of the poll operation on the receiver, reduces the effective value of L. We calculate L from our previous measurements of round-trip time and the sum S of overhead and known delay D:

$$
\begin{aligned}
L &= \frac{RTT}{2} - o_s - o_r \quad = \quad \frac{RTT}{2} - (S - D) \\
\Delta o_r &= \left[\frac{(\Delta RTT)^2}{4} + (\Delta S)^2 \right]^{1/2}
\end{aligned} \tag{5.4}
$$

As with our calculation of receive overhead, we have assumed in (5.4) that the random error on the delay D is negligible, and our accuracy is again limited by a systematic error of 3 cycles in the delay loop, which is included in reported results.

Send overhead breakdown

We gain a deeper understanding of the overhead required to send a message by breaking that cost down into components, as advocated in [SC87]. By repeatedly eliminating functionality through modification of the library code, we are able to determine the approximate cost of each piece that we remove. In particular, we iteratively remove a single component of functionality and perform the send overhead measurement described earlier, reporting the difference between two successive measurements as the cost of the component eliminated between those measurements.

Our approach to measuring the component costs requires that we define a canonical order for the components. As a compiler optimizes these components in tandem, the cost of two components A and C may not be the same as the cost of A plus the cost of ABC minus the cost of AB. This transformation from source to object code is fairly complex and hard to predict, and minor source modifications can result in significant changes in performance through the effects of architectural constraints. Register allocation heuristics and the resulting memory accesses can lead to such effects, for example.

Another approach to determining the relative costs of the components is to sample the program counter randomly and to compare the frequency of instructions associated with each component. For a large enough sample, this approach is more effective than ours in capturing the dynamic cost of each component in the actual system, and one is tempted to argue that sampling is in some sense independent of the ordering of components. However, sampling gives no more insight into the source to object translation than does a canonical ordering, and therefore provides no more information as to the advantages of eliminating some aspect of functionality. Also, with a superscalar architecture, program counter sampling may not provide adequate information to obtain accurate frequency information at the instruction level, particularly when instructions from separate source lines are interleaved at granularities below the superscalar issue rate. Finally, sampling perturbs the system by polluting the caches and altering the timing of operations. These perturbations can change the balance between components. We chose our method over sampling because it allowed a simple implementation and a clear interpretation of the results. Our canonical order and rationale for that order appear with the results of these measurements in Chapter 6.

Summary

We described in this section an array of microbenchmarks for measuring low-level parameters of a communication system. These tests explore the performance of message-passing based on concurrent queues in the absence of contention and allow us to investigate the costs of individual components of functionality. However, they do not provide much insight on the effects of contention. In the following sections, we extend our microbenchmark suite with tests that help us to understand those effects.

5.3.2 Stress test

We next explore performance at the extreme of high contention through many-to-one communication, a difficult scenario for most message-passing systems. With one process per physical processor, all but one process send a total of one million messages to the remaining process. Although each process occupies a processor for the duration of this communication stress test, the processes do not constantly contend for the queue, as they must also perform a significant amount of uncontended work. In particular, writing data into message packets, accepting reply messages and invoking the appropriate handler function, and polling for incoming messages operate on implicitly private data structures. Performing a substantial amount of work between queue operations separates our approach from the majority of concurrent algorithm literature, yet this functionality is simply a part of the message-passing layer. The test described generates the highest contention observable by an application using our Active Message layer, and may be representative of more reasonable workloads on larger machines. The distribution of execution times for the stress test appears normal, and we apply our standard methodology. We only run the stress test on one SMP, as processors outside of an SMP do not use the shared memory protocol and can only reduce contention by increasing the uncontended workload.

5.3.3 Application suite

Although performance at the extremes of contention is helpful for exploring the possible behavior of a communication layer, contention in applications generally falls somewhere in the middle of the range. Applications are the natural metric for performance, and their communication patterns are intrinsically interesting. In the previous sections, we used two communication patterns to explore the extremes of contention. Few applications adhere to either of those patterns, however; they typically utilize fairly complex and data-dependent communication, often employing a variety of such patterns during their execution.

We use several applications written in Split-C, which extends C with a global address space abstraction, support for simple distributed arrays, and global communication and synchronization operations. Issues of remote data consistency are left to the programmer. Split-C adopts an SPMD (single program, multiple data) style of programming, assigning a virtual processor number to each process in a given execution. The programs are written in a phase-structured style—processors proceed through a sequence of coarse-grained phases, performing a global synchronization between each phase. They use a wide variety of communication patterns, which also vary from phase to phase.

We perform one or more runs for each application, varying the input parameters to cover the range of possibilities for the application. We also use multiple runs to highlight the potential for improving performance by optimizing the layout of virtual processors into the physical SMP's that make up a Clump. Our measurements include only the main sections of the applications as outlined below, ignoring time and communication aspects of most startup and initialization sequences.

Distributions of application execution times typically have one peak that is nearly normal, but are truncated towards the low side and stretched towards the high side. We nevertheless chose to apply our standard methodology for all application tests, keeping total testing time to a reasonable value (approximately eight to ten days of dedicated Clump access, with full utilization of the processors). Figure 5.7 shows the distribution for SAMPLE on the Clump. In addition to the measured distribution, the figure shows two normal distributions, one calculated from the left peak of the measured data and the second calculated from the full data set. The truncation on the left depends on the amount of actual work that the application must perform. The long tail to the right illustrates the combined effects of synchronization dependencies and race conditions in message acceptance. Synchronization constraints can cause delays to accumulate, an effect well-known in the parallel scheduling literature [ADV+95].

In the remainder of this section, we discuss the algorithms and implementations for the applications, then cover the details of the individual runs and traffic patterns.

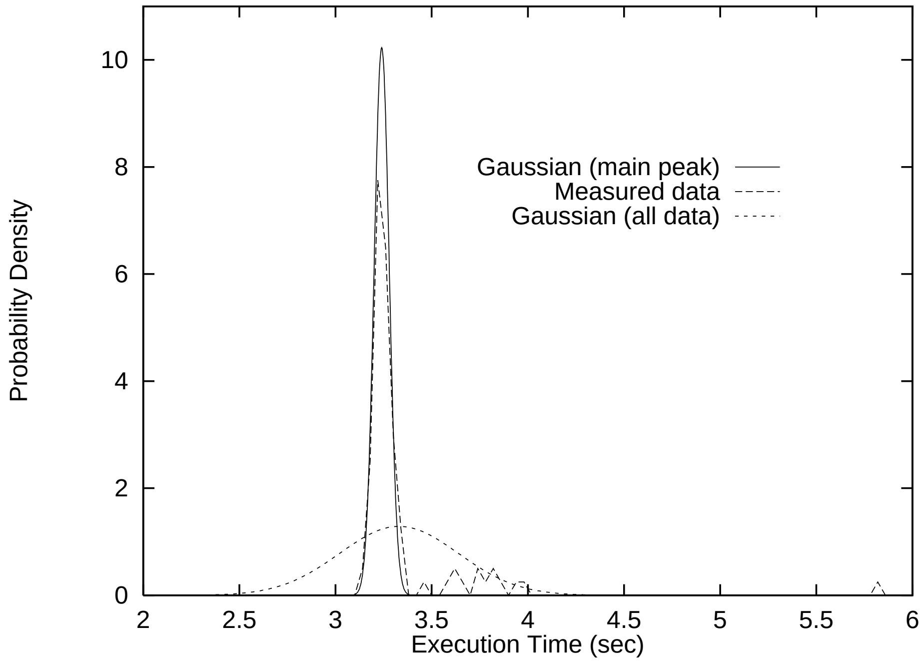

Figure 5.7: Sample sort execution time distribution on our Clump. The two Gaussians shown are calculated from the main peak of the measured data and from the full data set.

Sample sort

SAMPLE sorts 32-bit integers using sample sort [BLM91]. A sample sort uses a sorted random sample of *splitter* keys (integers) to distribute the full set of keys to processors; each processor receives the subset of keys that falls between two consecutive splitter keys. By oversampling to obtain the set of splitters, sample sort ensures that each processor receives roughly the same number of keys, generally within a factor of two of all other processors. The processors then sort their subsets independently and in parallel to obtain the final sorted order.

Our implementation of sample sort is a variant of that described in [CDMS93]. The application uses one initialization phase and four main phases. In the initialization phase, processors trade pointers to their local subsets to allow other processors to access the arrays directly through the Split-C global address space abstraction. The four main phases correspond to the description of the algorithm above. As certain portions of the algorithm are performed sequentially, we distinguish one processor as the *lead processor* for the sort; this term implies nothing beyond the fact that the lead processor performs the sequential portions. In the first phase, all processors sample their local key sets and pass their samples to the lead processor. In the second phase, the lead processor sorts the combined sample, selects the splitter set, and broadcasts that set to the other processors. The processors distribute their local keys sets according to the splitter values in the third phase, aggregating keys into groups of six to make full use of the short message payload (Split-C claims the remaining two words). The processors simultaneously collect all keys sent to them into a new local set. Finally, the processors sort their new local key sets, completing the sort.

Although our sample sort implementation does perform some many-to-one and some one-to-many communication, the dominant pattern by far is the all-to-all communication in the key distribution phase. SAMPLE thus represents applications that perform fine-grained, all-to-all communication.

Connected graph components

CON finds the connected components of a distributed graph [KLCY97]. Given a graph consisting of a set of nodes and an undirected set of edges between nodes, we say that two nodes in the graph are connected if there exists a path between the two nodes consisting only of edges in the graph. A connected component is then a maximal subset of nodes in which every two nodes within the subset are connected. A connected components algorithm identifies all such subsets and labels them with unique identifiers.

Our connected components implementation [LKC95] merges a simple sequential algorithm with an efficient parallel algorithm. We use graphs drawn from Monte Carlo approaches to exploring phase transitions and other critical phenomena as the basis of our study. Such graphs typically correspond to an underlying physical structure, which admits a straightforward partitioning among processors. Processors first find connected components within their local portion of the graph to produce a reduced graph, then apply an iterative, multi-phase parallel algorithm to the reduced graph.

The balance between computation and communication in this application depends strongly on the input parameters. We selected both a computation-bound run, labeled CON/comp, and a second, communication-bound run, labeled CON/comm. Due to the underlying physical structure of the graph, processors perform primarily fine-grained communication in a statistically well-defined pattern. However, most nodes in the graphs used in CON/comm merge into a single, very large connected component, resulting in high contention and load imbalance near the end of the execution. These effects can dominate the more regular communication pattern.

Fast Fourier transform

3-D FFT performs a fast Fourier transform (FFT) [PTVF93] in three dimensions. A Fourier transform calculates the coefficients of a given function in a dual basis defined as sine waves with arbitrary phase and frequency over the original basis. Typically, they are used to transform between spatial and frequency data. The FFT is a an efficient algorithm for calculating a discrete Fourier transform on a one-dimensional data set. For data with multiple dimensions, this transformation can be applied independently in each dimension.

The implementation that we use performs FFT's on each dimension of a cubic lattice. It distributes data to processors cyclically in one dimension, handing out two-dimensional slices in round-robin fashion. The application proceeds through four major phases. First, the processors perform FFT's on the cache-aligned dimension of their slices, then transpose those slices locally to align a new dimension of the data with the cache. The second phase breaks into many subphases to implement a parallel transpose of the data. In the subphases, each processor performs FFT's on the second dimension and sends the results to another processor. Processors synchronize globally between subphases to improve communication performance by avoiding contention [BK94]. In the third phase, the processors perform FFT's on the remaining dimension, which is again cache-aligned. The fourth phase repeats the parallel transpose to return the data to their original locations, breaking into subphases but not calculating further FFT's.

3-D FFT typifies regular applications that rely primarily on bulk communication. The communication pattern is all-to-all, but is scheduled into many one-to-one phases.

Electromagnetic wave propagation

EM3D, the last application, propagates electromagnetic radiation in three dimensions on a static, irregular mesh [Mad92]. The algorithm maintains electric field values along the edges of the mesh and magnetic field values along the edges of a dual mesh. Field updates occur in leapfrog fashion, with half of one timestep between alternate electric and magnetic field updates.

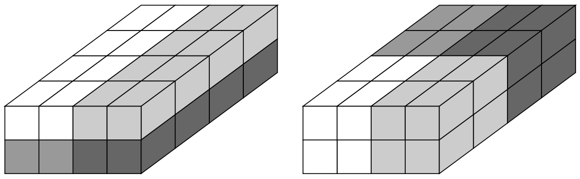

Figure 5.8: EM3D data partitioning. Each small block represents one processors' portion of the physical space. The shading represents the correlation of processors to SMP's. The left distribution produces more inter-SMP traffic than does the right.

Our implementation [CDG+93] generates a random mesh based on statistical characteristics of real meshes, representing the mesh with a bipartite graph of electric and magnetic field nodes. This graph corresponds to an underlying physical space partitioned between processors. Updating the electric field involves sending any magnetic field nodes coupled with electric field nodes on other processors to those processors and, once all data have arrived locally, computing new electric field values. After this update, all processors synchronize globally and begin an analogous update of the magnetic field nodes. We time only the computation of the electric field.

EM3D represents the class of applications that perform irregular, fine-grained communication. We use two distinct runs to highlight the effect of the phase-structured style and the advantage of intelligent virtual processor layout. As shown in Figure 5.8, both runs partition the underlying coordinate space on 32 processors into 4x2x4 blocks. The first run, EM3D/naive, uses a naive layout for virtual processors, placing processors within an SMP into 2x1x4 blocks. In the second run, denoted EM3D/good in the tables, an SMP's processors instead occupy 2x2x2 blocks, reducing the aggregate network traffic.

Application runs

We use six application runs on Clumps and five on individual SMP's. Table 5.5 lists the input parameters and memory requirements for each run. For each platform, memory requirements separate into resident set size, or physical memory use, and the application image size, or virtual memory use. The figures in the table represent peak memory usage over the lifetime of the application. None of the runs approaches the memory limitations on our platforms—2 GB per SMP, 8 GB on the Clump, and half that much on the equivalent NOW's—the parameters were instead tuned to result in roughly one second execution times

	Input Parameters	Memory (MB)			
		SMP		Clump	
		Resident	Image	Resident	Image
SAMPLE	262,144 nodes/processor	64	105	272	417
CON/comp	2D underlying lattice 640,000 nodes/processor 40% edges present	316	356	1,264	1,421
CON/comm	3D underlying lattice 512,000 nodes/processor 25% edges present	272	291	1,095	1,162
3-D FFT	128x128x128 values	74	102	171	312
EM3D	2,500 nodes/processor degree 20, 40% remote one-SMP layout	63	95	—	
EM3D/naive	2,500 nodes/processor degree 20, 40% remote naive Clump layout	—		268	383
EM3D/good	2,500 nodes/processor degree 20, 40% remote good Clump layout	—		265	382

Table 5.5: Input parameters and peak total memory usage for application runs on one SMP and on the Clump. The CON runs differ in the balance between communication and local computation. The EM3D runs differ in the layout of virtual processors.

	Short Messages			Bulk Data
	Mean	Min.	Max.	
SAMPLE	76,517	72,749	81,063	5 x 29 B
CON/comp	11,900	11,518	12,383	18 x 16 B
CON/comm	106,925	102,511	127,227	365 x 16 B
3-D FFT	3,647	3,620	3,692	3,584 x 2 kB
EM3D	141,448	140,940	142,120	none

Table 5.6: Per-processor communication volume for the SMP. Only 3-D FFT uses a significant number of bulk transfers.

on one Enterprise 5000. The memory usage is much larger than the L2 cache, however. With the exception of 3-D FFT, each application run scales with the number of processors.

The SAMPLE run sorts a quarter million integers per processor, for a total of two million keys on our SMP and eight million on our Clump. As mentioned earlier, we perform two distinct runs with the connected components application. CON/comp operates on a weakly connected, two-dimensional lattice with five million nodes on our SMP and twenty million nodes on our Clump. CON/comm does the same on a strongly connected, three-dimensional lattice with four million nodes on our SMP and sixteen million on our Clump. 3-D FFT performs a three-dimensional FFT on a cubic lattice of two million nodes; the problem is the same on both platforms. All EM3D runs propagate electromagnetic waves on a simulated cubic mesh with forty percent of edges linking nodes on two processors. The SMP run, EM3D, uses a total of twenty thousand nodes; the Clump runs use eighty thousand. EM3D/naive distributes virtual processors to SMP's in a naive fashion, while EM3D/good places them more carefully, transforming remote communication into local communication.

Traffic patterns

Communication traffic volumes appear in Tables 5.6, 5.7, and 5.8. The first table shows communication volume for applications on one SMP, relating the mean, minimum, and maximum number of short messages as well as the average number and size of bulk data transfers. All processors send within 6% of the mean number of short messages for every run except CON/comm. For that run, the processor that owns the single, large component sends roughly 19% more short messages than the mean.

Only 3-D FFT uses bulk data transfers for a significant fraction of communication

	Short Messages			Bulk Data	% Remote
	Mean	Min.	Max.		
SAMPLE	65,608	52,852	76,458	4 x 96 B	77
CON/comp	5,991	5,601	6,500	10 x 16 B	50
CON/comm	67,588	52,491	199,747	290 x 16 B	57
3-D FFT	966	768	1164	768 x 2 kB	77
EM3D/naive	60,856	40,380	81,580	none	38
EM3D/good	40,560	0	81,750	none	25

Table 5.7: Per-processor remote communication volume on a Clump of four 8-processor SMP's. 3-D FFT alone uses a significant number of bulk transfers. The communication-bound CON run suffers from load imbalance. Differences in virtual processor layout result in markedly different traffic distributions for the EM3D runs.

	Short Messages			Bulk Data	% Local
	Mean	Min.	Max.		
SAMPLE	19,119	15,622	21,591	3 x 29 B	23
CON/comp	5,889	5,351	6,410	6 x 16 B	50
CON/comm	50,677	43,856	80,666	130 x 16 B	43
3-D FFT	282	224	620	224 x 2 kB	23
EM3D/naive	101,220	79,198	122,670	none	62
EM3D/good	121,432	120,170	122,210	none	75

Table 5.8: Per-processor local communication volume on a Clump of four 8-processor SMP's. 3-D FFT alone uses a significant number of bulk transfers. The communication-bound CON run suffers from load imbalance. Differences in virtual processor layout result in markedly different traffic distributions for the EM3D runs.

traffic; in fact, the majority of short messages sent in 3-D FFT are simply replies to bulk transfer requests. SAMPLE uses a handful of bulk transfers to broadcast array pointers in the initialization phase, and CON passes small structures via bulk transfers. EM3D uses no bulk transfers at all.

Tables 5.7 and 5.8 break communication traffic on the Clump into remote and local components, respectively. The breakdown depends on the configuration of the Clump; the tables represent our experimental platform of four 8-processor SMP's. The tables also include percentages of total traffic sent using each protocol. The balance between processors is poorer than for applications on one SMP, due to a combination of statistical effects (*e.g.*, with SAMPLE) and tree-structured synchronizations (*e.g.*, with 3-D FFT). For SAMPLE and 3-D FFT, roughly three-quarters of the traffic is remote, as we expect from an all-to-all communication pattern on four SMP's with on average the same amount of data moving between every pair of processors.

The underlying physical structures used by the applications further skew the distributions, as the fraction of local communication depends on the location of a processor's neighbors in these structures. For the CON runs, the virtual processor layout is suboptimal, requiring an average of 50% of remote communication. The single, large component in CON/comm raises that average slightly. The naive virtual processor layout in EM3D leads to imbalanced remote and local traffic. EM3D/good transforms some remote traffic into local traffic, balancing the local traffic but causing further imbalance for remote traffic.

5.3.4 Multiprogrammed machines

The performance of a concurrent queue algorithm on a dedicated machine does not provide much information as to its performance on a multiprogrammed machine. Most algorithms choose between simplicity and speed, and limiting the impact of the operating system scheduler on performance. Algorithms based on spin locks, for example, favor the former, and perform well on dedicated machines. Algorithms based on preemption-safe locks favor the latter, trading performance on dedicated machines for robustness under multiprogramming.

We execute several copies of an application simultaneously to measure the performance of an algorithm on a multiprogrammed machine. Each copy, or job, uses one process per physical processor. Unlike many studies in multiprogramming, however, we do not pin processes to processors. Although such pinning can increase performance for controlled environments, it can have the opposite effect when such control is absent. For a statically-balanced parallel job pinned to processors, for example, a sequential job pinned to a processor has the same performance impact as another parallel job, leaving all but one processor idle for half of the execution. Pinning may thus improve benchmark performance without improving real performance.

Neither do unpinned parallel jobs present the harshest environment, however. Such jobs can add overhead through cache pollution, but a job's efforts to coschedule itself through yielding processors can interact positively with competing jobs' efforts to do the same. In contrast, sequential jobs do not cooperate in this manner; a sequential process always has useful work to do, and always holds the processor until its time slice expires. To address this difference, we also investigate the performance of a parallel job when competing with groups of sequential processes. A group is simply one sequential process per physical processor; each process operates independently of all others. We use a simulated application for this purpose: in an infinite loop, a process randomly increments values in an array the size of the L2 cache. This simulation causes significant cache pollution and a fair amount of memory traffic, allowing our measurements to explore the impact of more realistic interactions.

Parallel jobs started with a command line or script interface tend to execute in a very skewed fashion. The first job starts quickly, but the initialization code for a second job competes with the application processes of the first job. The competition slows the initialization, causing significant skew between the jobs. The skew reduces the effects of contention, cache pollution, and synchronization dependencies, producing artificially high levels of performance. We addressed this problem by splitting a single parallel job into multiple virtual jobs, synchronizing the jobs between initialization and execution of the application. We extended the Split-C initialization code to allow a group of processes to partition itself into subsets, each of which executes a Split-C application. After a barrier synchronization to ensure that all processors in a given subset are ready, one process in the subset broadcasts this fact through the filesystem (by creating a file). Each subset of processes then waits for the other subsets' files to appear, at which point the applications begin normal execution. Our global execution layer spawns enough processes for all jobs from a single command, creating twenty-four processes for three subsets of eight, for example.

We measure multiprogramming effects only on a single SMP, primarily due to a lack of integration between the network and shared memory event mechanisms. The multiprogramming tests consist of four specific measurements. We first run two jobs simultaneously, recording the times from both jobs and repeating the experiment 50 times to obtain 100 values. We next run three jobs simultaneously and record the times from all three, repeating the experiment 40 times to obtain 120 values. For sequential competition, we start one group of competing processes and time 100 runs of one parallel job, then start a second group of competing processes and time another 100 runs of one parallel job.

We apply our standard methodology to all of these experiments. Although our method of synchronizing the jobs does for the most part achieve the desired effect of forcing the jobs to compete for the processors, the approach slightly skews the distribution of execution time for multiple parallel jobs towards shorter runs. When the first job finishes, the remaining job or jobs need only compete amongst themselves, and can thus finish the rest of their work in less time. Although the first job is technically drawn from the appropriate distribution, remember that it is in fact the minimum of several variables drawn from that distribution. The job times are also not independent. These problems do not arise in our measurements of a parallel job competing with sequential jobs, as the latter execute continuously over the lifetime of the parallel job. A similar approach might better serve for the measurement of multiple parallel jobs, constantly executing the competing parallel jobs while measuring only the job that runs against such a workload, but we felt that the skew effects were small enough to disregard, particularly since they should have roughly the same impact on different concurrent message queue algorithms and thus not affect comparisons between them.

5.3.5 Summary

The suite of benchmarks described here provides a broad spectrum of measurements for communication through shared memory. They cover the range of contention, yet also highlight performance at more realistic levels and capture the effects of competition between multiple jobs. The parameters for the LogGP model, a well-accepted model in the user-level communication literature, investigate performance in the absence of contention. The stress test uses many-to-one communication to complement these measurements, evaluating performance at the extreme of high contention. A set of Split-C applications examines performance at more reasonable levels and for a range of communication patterns. Lastly, multiple competing parallel and sequential jobs serve as a metric of performance on multi-programmed machines.

We apply a subset of benchmarks to measure performance on a Clump. The stress test does not serve the goal of maximal contention on a Clump, and our communication layer does not integrate the network and shared memory multiprogramming mechanisms. Thus we report Clump performance only in terms of LogGP parameters and application execution times.

Chapter 6

Shared Memory Protocol Performance

In this chapter, we study the performance of the shared memory protocol in isolation, *i.e.*, used only within a single SMP, focusing on the performance of our lock-free concurrent message queue algorithm. We begin by tuning the shared memory protocol to our experimental platform, selecting appropriate backoff yield and queue length parameters. We then introduce a variety of alternative concurrent queue algorithms and use our benchmark suite to compare their performance with that of our own lock-free algorithm. The chapter then presents a detailed breakdown of the cost of message insertion using the shared memory protocol, illustrating the overhead associated with each aspect of functionality. In light of that breakdown, we forecast the impact of trends in architecture on message-passing and suggest a strategy for improving performance with a small set of extensions to a typical SMP architecture. The chapter concludes with a comparison of application performance between an SMP and a comparable NOW, illustrating the impact of the faster protocol at the application level. In the next chapter, we report performance on the Clump.

6.1 Backoff Yield Selection

The performance of our communication architecture depends on the proper selection of the parameters described in previous chapters: the time spent backing off from a full queue before yielding, the packet and bulk data queue lengths, and the parameters that define the adaptive polling strategy. We address the first two parameters in this chapter, delaying polling strategy tuning until we consider multi-protocol performance in Chapter 7. We begin by selecting a good operating point for each parameter through a preliminary study, then tune the communication layer by varying individual parameters around our chosen operating point. A combination of dedicated and multiprogrammed application performance, measured as described in the last chapter, serves as the metric for our studies. Although the parameters may depend strongly on specific aspects of a platform, their dependency on particular applications must be relatively weak for the communication layer to serve as a general-purpose substrate. By reporting the performance of applications with a variety of communication patterns, we establish a degree of independence between the parameters and the specific nature of any application.

We first consider the selection of the backoff yield time. This parameter controls the time that a process waits to insert a message into a full queue before yielding its processor. The communication layer must balance the overhead associated with switching between the processes with the time spent waiting for a descheduled or inattentive process to drain messages from a queue. In this section, we expand on the sources of the switching overhead, then examine the effect of varying backoff yield times on application performance. As mentioned in Chapter 3, we use a backoff yield time of 255 microseconds with our system.

Yielding a processor potentially incurs overhead of three types. First, the process must invoke the operating system scheduler to locate a new process for the processor. This invocation results in at least two context switches and some minor pollution of the data and instruction caches. If the scheduler selects a different process to run, the cache pollution is in general far more extensive, leading to the second type of overhead: reloading cached data. Finally, if the original process migrates to a new processor in its next time quantum, the process must pay the cost of restoring all cached state, including flushing dirty lines in the new cache, moving remaining lines from the original cache, and reloading the remainder of the working set.

We investigate the tradeoff between increased overhead and idle cycles by measuring application performance for both dedicated and multiprogrammed SMP's. The multiprogramming experiments use only parallel jobs, as we expect competition with sequential jobs to produce qualitatively similar results, and the smaller number of experiments simplifies the presentation of the measurements. We normalize the results for each application to execution time in the absence of the processor yield code, then present normalized execution time per job. The bound on the exponential backoff is set to 255 microseconds for

SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
0.697 ± 0.007 $\sigma = 0.032$	1.086 ± 0.006 $\sigma = 0.026$	1.660 ± 0.006 $\sigma = 0.025$	0.676 ± 0.005 $\sigma = 0.023$	0.643 ± 0.005 $\sigma = 0.021$

Table 6.1: Normalization times in seconds for backoff yield selection. These times reflect mean execution times on a dedicated SMP with a layer that spins rather than yielding the processor.

the normalization executions, but processes merely enter another delay cycle rather than yielding their processors after backing off for that period.

The normalization time for each application, obtained and reported using our standard methodology, appears in Table 6.1. Figures 6.1 through 6.5 present the results of the tuning experiment using one figure per application. We varied backoff yield time from 7 to 2047 microseconds to cover an interesting range around our nominal value of 255 microseconds. The upper portion of each figure provides a table of normalized execution times in seconds per job for one, two, and three simultaneous parallel jobs, all reported using our standard methodology. The lower portion of each figure presents these results graphically, using a single scale to simplify cross-application comparisons.

On a dedicated machine, only the context switches and the migration of cached data contribute to the overhead, as the system has no other useful work to perform. For the same reason, however, yielding the process does not convey any performance advantage. The normalized time thus approaches 1 asymptotically as the yield time approaches infinity. This trend holds for all applications, as shown in the figures, although the impact of overly frequent yielding of processors varies from application to application. CON/comm (Figure 6.3) fares the worst in this regard due to the use of many-to-one communication near the end of its execution. In this case, one process owns a giant connected component, and all other processes switch in and out very rapidly while waiting to communicate with the first process. CON/comp (Figure 6.2) and EM3D (Figure 6.5) also suffer fairly substantial penalties as the yield time becomes smaller, perhaps due to dependency chains amongst communicating neighbor processes. SAMPLE (Figure 6.1) and 3-D FFT (Figure 6.4) performance, which use all-to-all communication, degrade slightly below a 255 microsecond yield time, but are stable from that point down to 7 microseconds.

The penalty for including yielding at 255 microseconds ranges from 0 to 14% on a dedicated machine, but yielding results in far superior performance on multiprogrammed

Yield Time (usec)	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
7	1.22 ± 0.02 $\sigma = 0.08$	1.75 ± 0.06 $\sigma = 0.26$	1.76 ± 0.05 $\sigma = 0.26$
31	1.19 ± 0.02 $\sigma = 0.09$	1.62 ± 0.05 $\sigma = 0.24$	1.66 ± 0.05 $\sigma = 0.26$
127	1.21 ± 0.02 $\sigma = 0.08$	1.51 ± 0.05 $\sigma = 0.24$	1.59 ± 0.05 $\sigma = 0.25$
255	1.093 ± 0.005 $\sigma = 0.025$	1.39 ± 0.04 $\sigma = 0.19$	1.45 ± 0.04 $\sigma = 0.19$
511	1.085 ± 0.007 $\sigma = 0.032$	1.38 ± 0.05 $\sigma = 0.20$	1.45 ± 0.04 $\sigma = 0.19$
2,047	1.072 ± 0.005 $\sigma = 0.024$	1.37 ± 0.05 $\sigma = 0.22$	1.40 ± 0.04 $\sigma = 0.18$

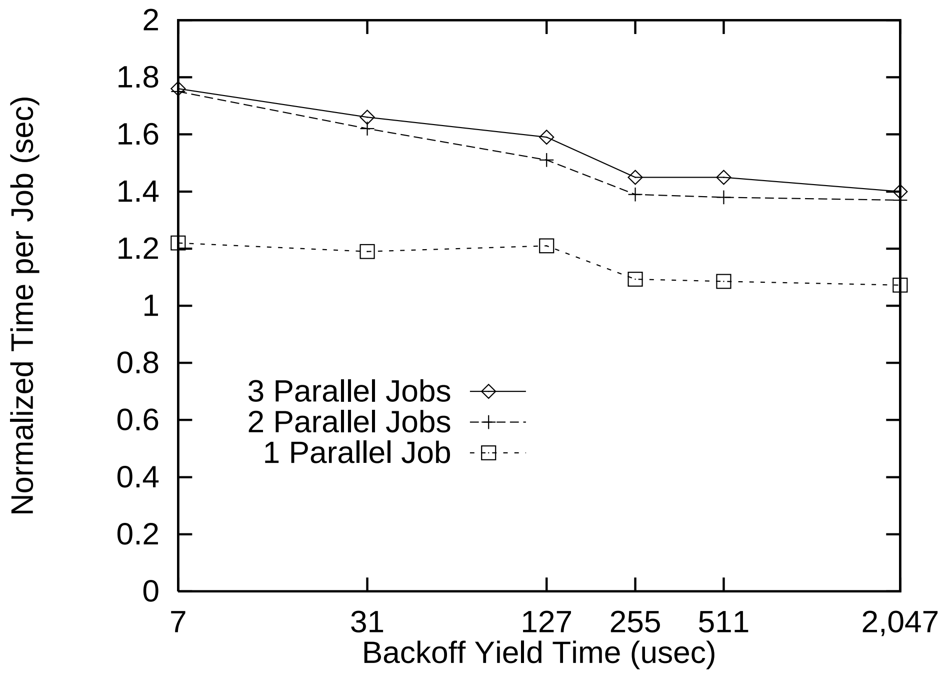

Figure 6.1: Normalized execution time in seconds per job for SAMPLE as a function of backoff yield time in microseconds. Times are normalized to 0.697 seconds.

Yield Time (usec)	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
7	1.87 ± 0.02 $\sigma = 0.06$	1.91 ± 0.03 $\sigma = 0.14$	2.18 ± 0.04 $\sigma = 0.17$
31	1.13 ± 0.02 $\sigma = 0.07$	1.38 ± 0.03 $\sigma = 0.14$	1.74 ± 0.04 $\sigma = 0.17$
127	1.07 ± 0.02 $\sigma = 0.09$	1.12 ± 0.03 $\sigma = 0.14$	1.10 ± 0.03 $\sigma = 0.12$
255	1.02 ± 0.02 $\sigma = 0.05$	1.09 ± 0.03 $\sigma = 0.12$	1.08 ± 0.03 $\sigma = 0.14$
511	1.017 ± 0.008 $\sigma = 0.036$	1.09 ± 0.03 $\sigma = 0.12$	1.09 ± 0.03 $\sigma = 0.13$
2,047	1.013 ± 0.007 $\sigma = 0.035$	1.08 ± 0.04 $\sigma = 0.16$	1.15 ± 0.04 $\sigma = 0.17$

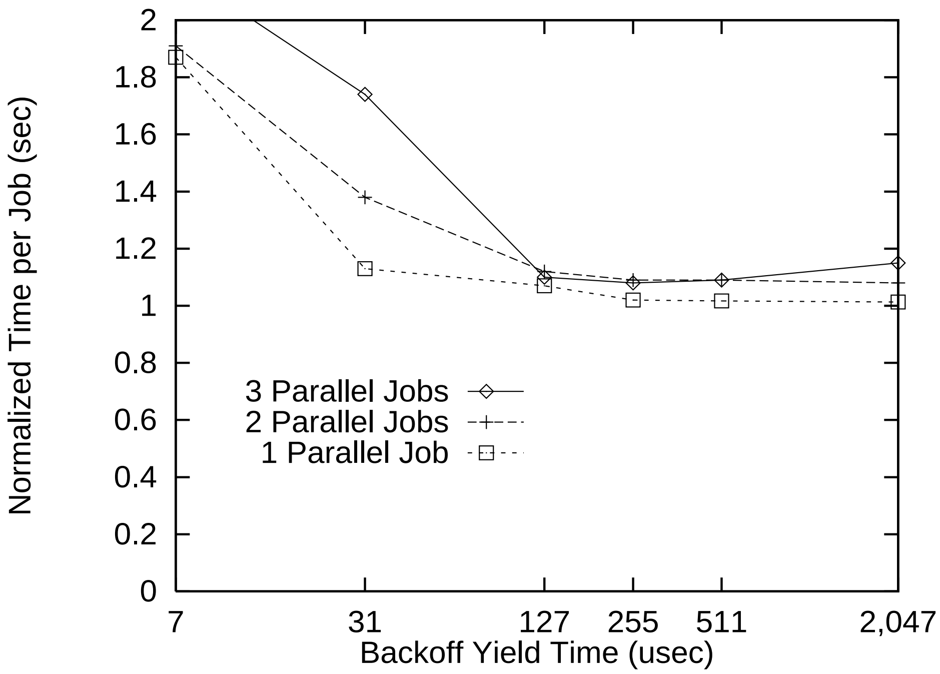

Figure 6.2: Normalized execution time in seconds per job for CON/comp as a function of backoff yield time in microseconds. Reported times are normalized to 1.086 seconds.

Yield Time (usec)	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
7	5.40 ± 0.03 $\sigma = 0.13$	7.32 ± 0.04 $\sigma = 0.17$	8.99 ± 0.05 $\sigma = 0.23$
31	1.81 ± 0.03 $\sigma = 0.11$	3.53 ± 0.04 $\sigma = 0.19$	6.35 ± 0.04 $\sigma = 0.22$
127	1.26 ± 0.03 $\sigma = 0.12$	1.70 ± 0.03 $\sigma = 0.13$	2.20 ± 0.03 $\sigma = 0.16$
255	1.14 ± 0.03 $\sigma = 0.11$	1.58 ± 0.03 $\sigma = 0.14$	1.89 ± 0.03 $\sigma = 0.16$
511	1.12 ± 0.02 $\sigma = 0.10$	1.72 ± 0.04 $\sigma = 0.17$	2.06 ± 0.04 $\sigma = 0.17$
2,047	1.11 ± 0.03 $\sigma = 0.12$	2.01 ± 0.06 $\sigma = 0.25$	2.99 ± 0.05 $\sigma = 0.27$

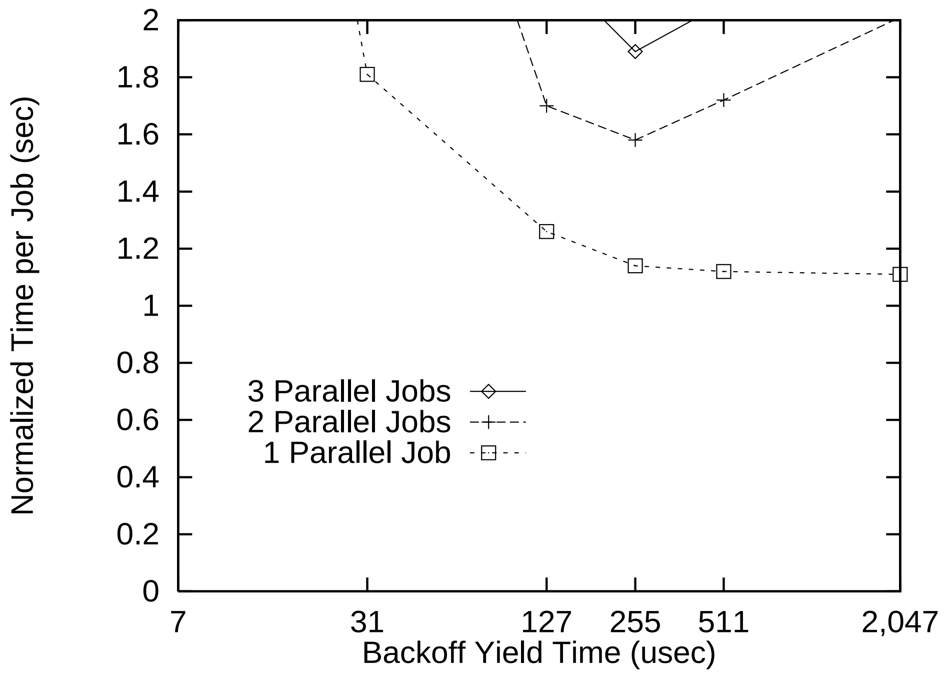

Figure 6.3: Normalized execution time in seconds per job for CON/comm as a function of backoff yield time in microseconds. Reported times are normalized to 1.660 seconds.

Yield Time (usec)	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
7	1.08 ± 0.02 $\sigma = 0.09$	1.26 ± 0.04 $\sigma = 0.15$	1.28 ± 0.04 $\sigma = 0.20$
31	1.05 ± 0.02 $\sigma = 0.07$	1.25 ± 0.03 $\sigma = 0.14$	1.25 ± 0.04 $\sigma = 0.19$
127	1.04 ± 0.02 $\sigma = 0.07$	1.23 ± 0.03 $\sigma = 0.14$	1.25 ± 0.04 $\sigma = 0.17$
255	1.00 ± 0.02 $\sigma = 0.05$	1.23 ± 0.04 $\sigma = 0.16$	1.22 ± 0.03 $\sigma = 0.15$
511	1.00 ± 0.02 $\sigma = 0.05$	1.25 ± 0.04 $\sigma = 0.15$	1.24 ± 0.03 $\sigma = 0.16$
2,047	0.983 ± 0.009 $\sigma = 0.043$	1.27 ± 0.04 $\sigma = 0.16$	1.30 ± 0.03 $\sigma = 0.15$

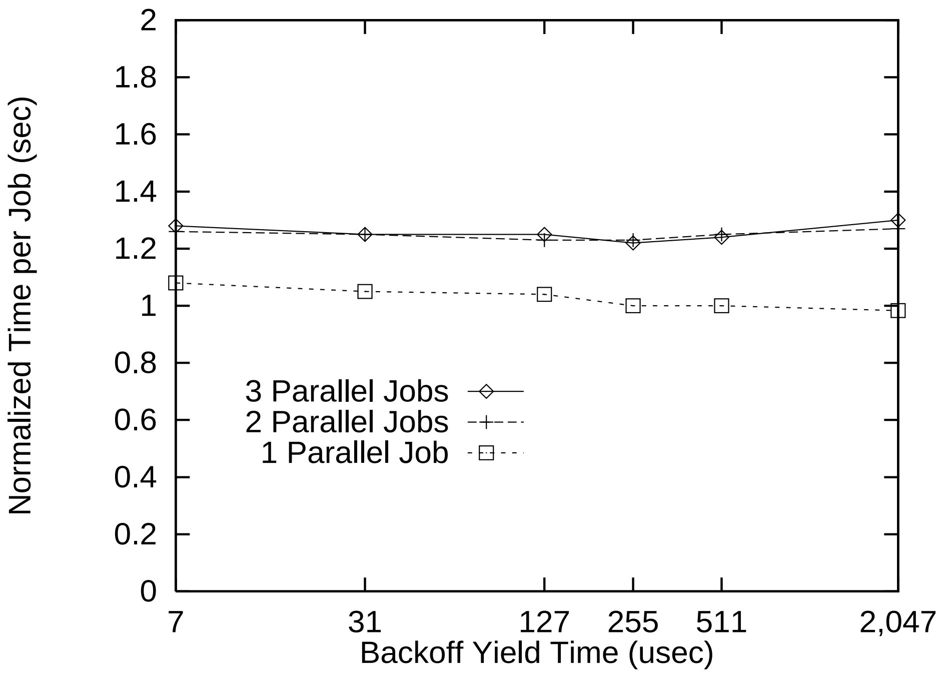

Figure 6.4: Normalized execution time in seconds per job for 3-D FFT as a function of backoff yield time in microseconds. Times are normalized to 0.676 seconds.

Yield Time (usec)	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
7	1.47 ± 0.05 $\sigma = 0.21$	1.60 ± 0.05 $\sigma = 0.21$	1.53 ± 0.05 $\sigma = 0.23$
31	1.15 ± 0.03 $\sigma = 0.12$	1.40 ± 0.04 $\sigma = 0.16$	1.36 ± 0.03 $\sigma = 0.16$
127	1.09 ± 0.02 $\sigma = 0.10$	1.21 ± 0.03 $\sigma = 0.12$	1.15 ± 0.03 $\sigma = 0.12$
255	1.06 ± 0.02 $\sigma = 0.07$	1.18 ± 0.03 $\sigma = 0.13$	1.11 ± 0.03 $\sigma = 0.13$
511	1.04 ± 0.02 $\sigma = 0.06$	1.18 ± 0.03 $\sigma = 0.14$	1.08 ± 0.03 $\sigma = 0.13$
2,047	1.03 ± 0.02 $\sigma = 0.06$	1.20 ± 0.03 $\sigma = 0.14$	1.12 ± 0.03 $\sigma = 0.13$

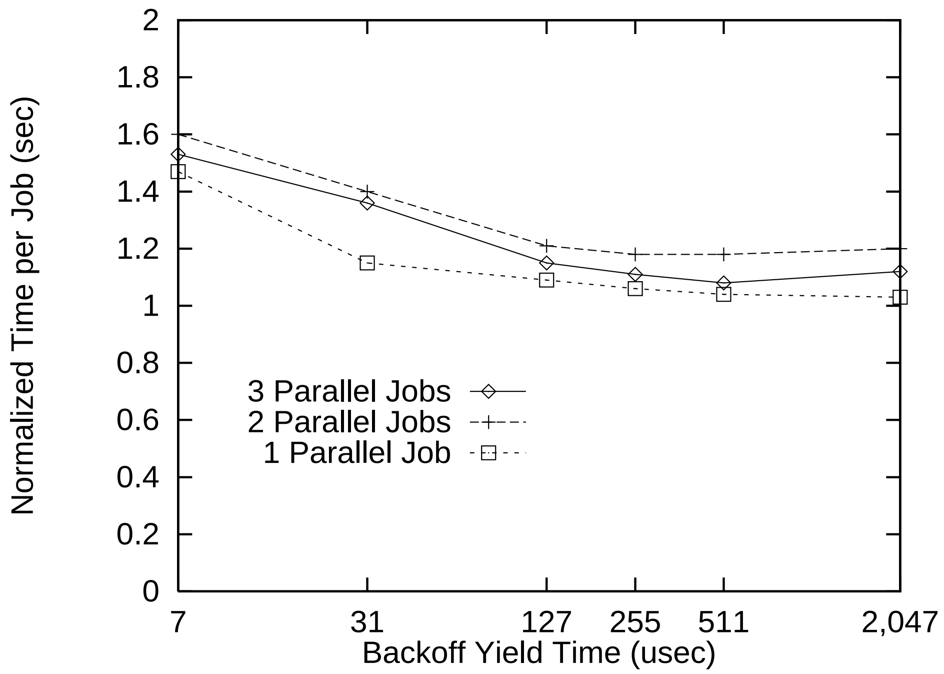

Figure 6.5: Normalized execution time in seconds per job for EM3D as a function of backoff yield time in microseconds. Times are normalized to 0.643 seconds.

machines. The multiple job data in the figures is similar to the single job data with two additional features arising from cache pollution and frequent yielding.

Cache pollution decreases overall performance under multiprogramming. The impact of this problem varies from application to application, but for most applications is the same with two or three simultaneous jobs, as we expect if a process' cached data are completely removed or migrated when another process claims its processor. The exception to this rule is CON/comm (Figure 6.8). The performance bottleneck in the final phase of CON/comm is the process that owns the single, large connected component. The remaining processes leave much of the cache untouched as they attempt to communicate with the bottleneck process. With more simultaneous jobs, however, more processes intervene between two time slices of a single process, exacerbating the cache pollution and leading to the performance shown in the figure.

Multiprogrammed applications also suffer performance degradation when the yield time grows too large, as processes spend more time waiting for their descheduled communication partners to accept messages. This effect is most prominent with COM/comm, again due to the use of many-to-one communication.

In light of the data, we select 255 microseconds as our backoff yield time. Smaller values unnecessarily penalize both dedicated and multiprogrammed performance by incurring the overheads associated with yielding too frequently. Larger values penalize multiprogrammed performance for some applications by wasting cycles waiting for descheduled processes.

6.2 Queue Length Selection

We now consider the selection of the queue lengths, the number of entries in the packet and bulk data queues used by the shared memory protocol. The communication layer must balance between the queues' ability to buffer incoming messages and their memory footprint. In this section, we discuss the tradeoff in slightly more detail, then present and explain data on the impact of the queue lengths on performance.

The factors affecting queue length arise from the ability of a queue to buffer incoming messages for a descheduled or unresponsive process. Consider a simple, two-stage cyclic model of skewed processes. In the first stage, a receiver process occupies a processor, drains its queue, and is eventually descheduled. In the second stage, a sender process occu-

SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
0.762 ± 0.004	1.11 ± 0.02	1.89 ± 0.04	0.676 ± 0.008	0.682 ± 0.009
$\sigma = 0.017$	$\sigma = 0.05$	$\sigma = 0.19$	$\sigma = 0.036$	$\sigma = 0.043$

Table 6.2: Normalization times in seconds for packet queue length selection. These times reflect mean execution times on a dedicated SMP with a backoff yield time of 255 microseconds, a queue length of 4,096, and a bulk data queue length of 16.

pies a processor, sends messages to the first until its queue is full, and immediately yields the processor. The cycle then repeats. Filling the queue takes time, and can be considered progress for the sender, but such progress must be balanced against the corresponding overhead, the cost of two context switches.

The model just described implicitly assumes that the processor yielded by the sender has other useful work to perform. We might instead ask that a sender be allowed to make progress for some fraction of a scheduling time-slice, a time typically four orders of magnitude longer than a context switch overhead. Queues meeting that criterion can withstand a reasonable amount of skew between communicating processes without incurring significant amounts of additional overhead. Long queues can adversely impact cache performance, however, bumping application data out of the the data cache to make space for communication data. The appropriate queue length depends on the relative importance of these two factors on a particular system.

The application performance measurements used to investigate the backoff yield parameter also serve for queue length. We again present execution time per job normalized to the application execution time when using our final parameter values: 255 microsecond backoff yield, 4,096-entry packet queues, and 16-entry bulk data queues. The normalization time for each application, obtained and reported using our standard methodology, appears in Table 6.2. Figures 6.6 through 6.10 present the results of the packet queue length investigation using one figure per application. We used only 3-D FFT in the bulk data queue length investigation, as only that application makes significant use of bulk transfer messages; these data appear in Figure 6.11. We varied the packet queue length from 256 to 32,768 entries to cover an interesting range around our nominal value; towards the same end, we varied the bulk data queue length from 2 to 128 entries. The figure format is identical to the format used in the previous section on backoff yield time. The upper portion of each figure provides a table of normalized execution times in seconds per job for one, two, and three simultaneous parallel jobs, all reported using our standard methodology. The lower portion of each figure presents these results graphically, using a single scale to simplify cross-application comparisons.

The data on a dedicated machine have a similar character for all applications, showing a slight rise for very long queues, presumably due to pollution of data caches and TLB's with communication data. EM3D (Figure 6.10) also shows a decrease in performance below queue lengths of 2,048, a side-effect of an optimization that allows processes that have received any necessary remote magnetic field information to begin computing new electric field values for their own nodes. During the computing portion of the phase, processes ignore the communication layer, potentially allowing their queues to fill and stalling other processes. The EM3D programmer might reasonably expect this behavior not to occur, as the program guarantees that a process has both sent and received all necessary data before starting to compute. However, the reply messages required by the AM-II specification can exhibit the blocking behavior.

EM3D performance improves as the queue length decreases further, suggesting a second factor such as L1 cache pollution. More likely, the shorter queues reduce the impact of the blocking behavior by coupling the processes' progress; no process can get too far ahead of another without stalling on a full queue.

Multiprogramming again imposes a penalty to performance due to the overhead of switching between individual processes, and again this penalty is the same for two or three parallel jobs except for CON/comm. The overhead incurred due to queue overflow is also prominent under multiprogramming, particularly for communication-intensive applications such as SAMPLE (Figure 6.6) and EM3D. CON/comm is also fairly communication-intensive, but the many-to-one nature of the communication hides the overhead, as the senders are limited by the receiver at all queue lengths studied.

The final feature of interest is the inversion of performance for long queues with SAMPLE, in which multiprogrammed machines outperform dedicated ones. Many possibilities exist, including systematic error due to the skewed distributions measured with multiple parallel jobs. Multiprogramming can improve performance for applications with few interprocess dependencies, however. Effectively, a multiprogrammed machine provides only a fraction of its processors to a single job on average, thereby reducing the concurrent access contention and message arrival rates within that job. Process migration and cache pollution can also have positive impact. A sender might fill a processor's cache with

Queue Length	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
256	1.012 ± 0.009 $\sigma = 0.045$	3.3 ± 0.3 $\sigma = 1.0$	10.9 ± 0.4 $\sigma = 2.0$
512	0.973 ± 0.006 $\sigma = 0.026$	2.9 ± 0.2 $\sigma = 0.9$	5.9 ± 0.3 $\sigma = 1.4$
1,024	1.000 ± 0.009 $\sigma = 0.042$	2.1 ± 0.2 $\sigma = 0.6$	3.3 ± 0.2 $\sigma = 0.6$
2,048	0.992 ± 0.005 $\sigma = 0.023$	1.64 ± 0.08 $\sigma = 0.37$	2.03 ± 0.07 $\sigma = 0.33$
4,096	1.03 ± 0.02 $\sigma = 0.08$	1.29 ± 0.05 $\sigma = 0.23$	1.34 ± 0.04 $\sigma = 0.21$
8,192	1.050 ± 0.009 $\sigma = 0.043$	1.10 ± 0.03 $\sigma = 0.14$	1.07 ± 0.03 $\sigma = 0.16$
16,384	1.112 ± 0.007 $\sigma = 0.031$	1.06 ± 0.03 $\sigma = 0.13$	1.01 ± 0.03 $\sigma = 0.11$
32,768	1.260 ± 0.008 $\sigma = 0.039$	1.12 ± 0.03 $\sigma = 0.12$	1.03 ± 0.02 $\sigma = 0.10$

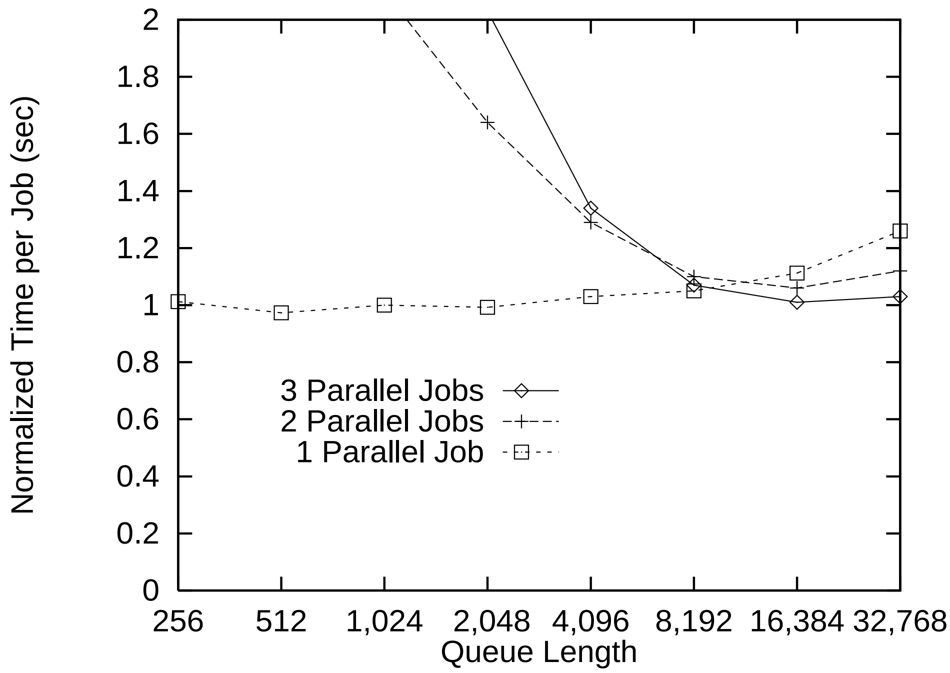

Figure 6.6: Normalized execution time in seconds per job for SAMPLE as a function of packet queue length. Times are normalized to 0.762 seconds.

Queue Length	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
256	0.978 ± 0.008 $\sigma = 0.035$	1.07 ± 0.03 $\sigma = 0.12$	1.05 ± 0.03 $\sigma = 0.11$
512	0.983 ± 0.009 $\sigma = 0.040$	1.04 ± 0.03 $\sigma = 0.10$	1.05 ± 0.03 $\sigma = 0.11$
1,024	0.99 ± 0.02 $\sigma = 0.05$	1.05 ± 0.03 $\sigma = 0.12$	1.02 ± 0.03 $\sigma = 0.11$
2,048	0.99 ± 0.01 $\sigma = 0.05$	1.06 ± 0.03 $\sigma = 0.13$	1.05 ± 0.03 $\sigma = 0.12$
4,096	0.997 ± 0.007 $\sigma = 0.034$	1.07 ± 0.03 $\sigma = 0.13$	1.06 ± 0.03 $\sigma = 0.11$
8,192	0.98 ± 0.02 $\sigma = 0.06$	1.06 ± 0.03 $\sigma = 0.11$	1.04 ± 0.03 $\sigma = 0.12$
16,384	0.979 ± 0.008 $\sigma = 0.038$	1.05 ± 0.03 $\sigma = 0.11$	1.04 ± 0.03 $\sigma = 0.12$
32,768	0.984 ± 0.009 $\sigma = 0.044$	1.07 ± 0.03 $\sigma = 0.13$	1.07 ± 0.03 $\sigma = 0.14$

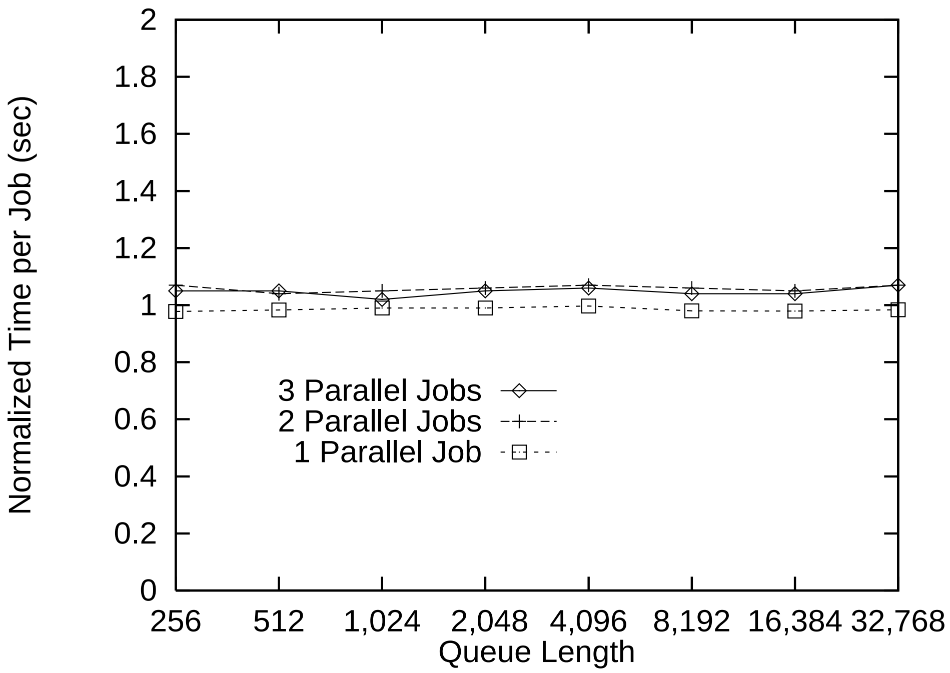

Figure 6.7: Normalized execution time in seconds per job for CON/comp as a function of packet queue length. Times are normalized to 1.11 seconds.

Queue Length	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
256	0.96 ± 0.02 $\sigma = 0.08$	1.41 ± 0.03 $\sigma = 0.11$	1.67 ± 0.03 $\sigma = 0.15$
512	0.95 ± 0.02 $\sigma = 0.07$	1.39 ± 0.03 $\sigma = 0.11$	1.67 ± 0.03 $\sigma = 0.15$
1,024	0.99 ± 0.02 $\sigma = 0.10$	1.40 ± 0.03 $\sigma = 0.11$	1.68 ± 0.03 $\sigma = 0.14$
2,048	0.97 ± 0.02 $\sigma = 0.07$	1.39 ± 0.03 $\sigma = 0.10$	1.66 ± 0.03 $\sigma = 0.13$
4,096	1.01 ± 0.02 $\sigma = 0.10$	1.41 ± 0.03 $\sigma = 0.12$	1.67 ± 0.04 $\sigma = 0.17$
8,192	1.00 ± 0.02 $\sigma = 0.08$	1.40 ± 0.03 $\sigma = 0.11$	1.64 ± 0.03 $\sigma = 0.15$
16,384	1.06 ± 0.03 $\sigma = 0.10$	1.44 ± 0.03 $\sigma = 0.13$	1.71 ± 0.03 $\sigma = 0.14$
32,768	1.10 ± 0.02 $\sigma = 0.10$	1.54 ± 0.03 $\sigma = 0.12$	1.81 ± 0.03 $\sigma = 0.16$

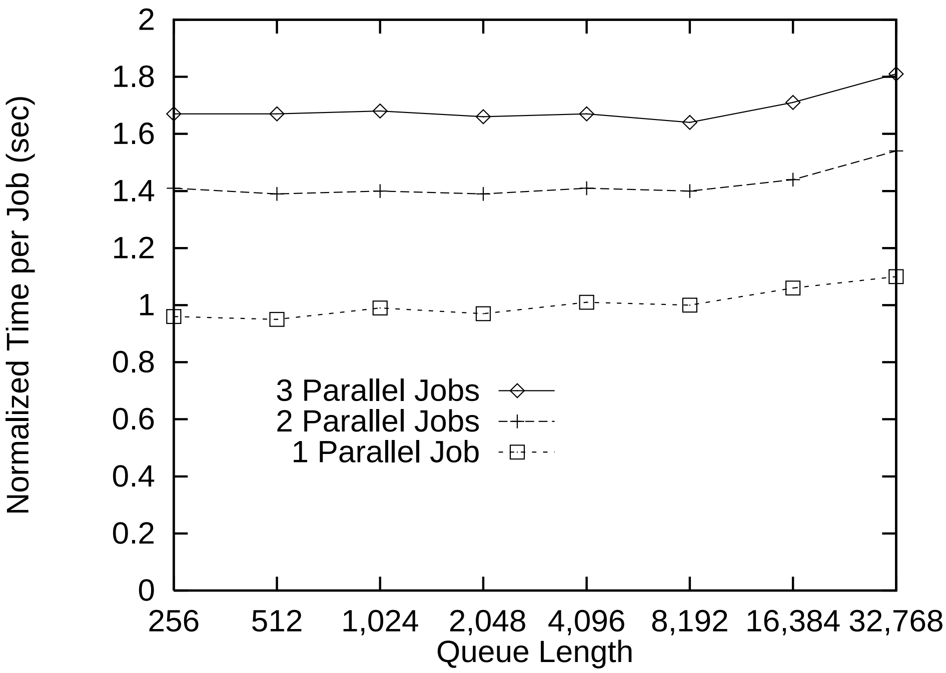

Figure 6.8: Normalized execution time in seconds per job for CON/comm as a function of packet queue length. Times are normalized to 1.89 seconds.

Queue Length	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
256	0.98 ± 0.02 $\sigma = 0.05$	1.23 ± 0.04 $\sigma = 0.16$	1.23 ± 0.03 $\sigma = 0.16$
512	0.979 ± 0.009 $\sigma = 0.043$	1.21 ± 0.03 $\sigma = 0.14$	1.22 ± 0.03 $\sigma = 0.15$
1,024	0.992 ± 0.009 $\sigma = 0.044$	1.22 ± 0.04 $\sigma = 0.15$	1.23 ± 0.03 $\sigma = 0.16$
2,048	0.99 ± 0.02 $\sigma = 0.06$	1.23 ± 0.04 $\sigma = 0.15$	1.22 ± 0.03 $\sigma = 0.15$
4,096	1.00 ± 0.02 $\sigma = 0.06$	1.23 ± 0.03 $\sigma = 0.14$	1.22 ± 0.03 $\sigma = 0.15$
8,192	1.005 ± 0.009 $\sigma = 0.044$	1.25 ± 0.03 $\sigma = 0.15$	1.25 ± 0.04 $\sigma = 0.17$
16,384	1.01 ± 0.02 $\sigma = 0.06$	1.25 ± 0.04 $\sigma = 0.16$	1.25 ± 0.04 $\sigma = 0.17$
32,768	1.00 ± 0.01 $\sigma = 0.05$	1.25 ± 0.04 $\sigma = 0.16$	1.24 ± 0.03 $\sigma = 0.16$

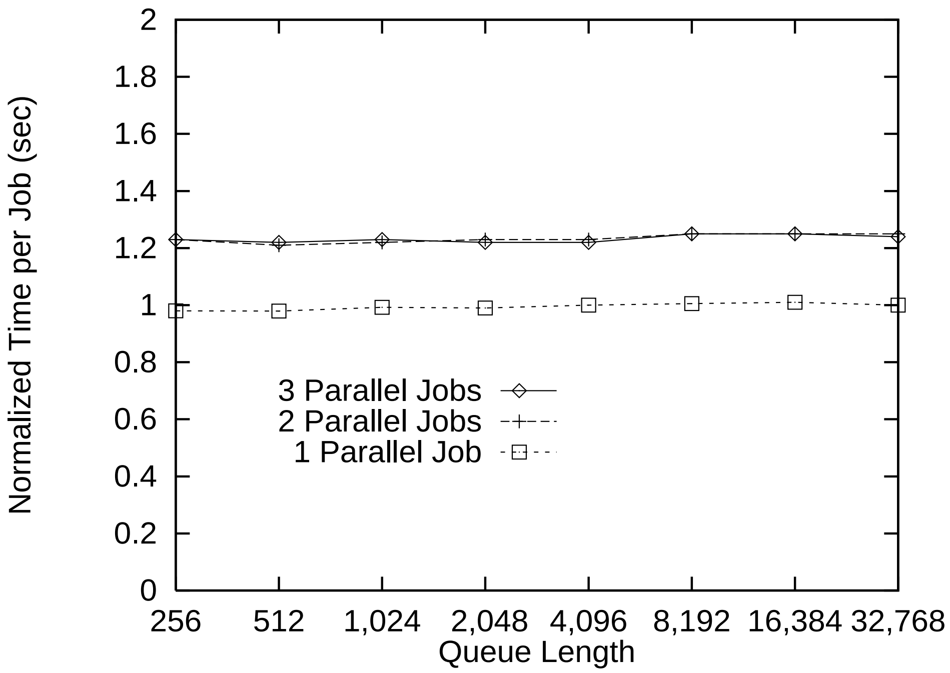

Figure 6.9: Normalized execution time in seconds per job for 3-D FFT as a function of packet queue length. Times are normalized to 0.676 seconds.

Queue Length	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
256	1.13 ± 0.02 $\sigma = 0.07$	1.49 ± 0.04 $\sigma = 0.18$	1.74 ± 0.06 $\sigma = 0.30$
512	1.15 ± 0.02 $\sigma = 0.08$	1.41 ± 0.04 $\sigma = 0.17$	1.57 ± 0.06 $\sigma = 0.29$
1,024	1.20 ± 0.02 $\sigma = 0.09$	1.33 ± 0.03 $\sigma = 0.15$	1.45 ± 0.05 $\sigma = 0.22$
2,048	1.02 ± 0.01 $\sigma = 0.05$	1.23 ± 0.04 $\sigma = 0.16$	1.26 ± 0.03 $\sigma = 0.15$
4,096	0.99 ± 0.02 $\sigma = 0.06$	1.11 ± 0.03 $\sigma = 0.11$	1.06 ± 0.03 $\sigma = 0.14$
8,192	1.02 ± 0.02 $\sigma = 0.07$	1.11 ± 0.03 $\sigma = 0.12$	1.00 ± 0.02 $\sigma = 0.10$
16,384	1.01 ± 0.02 $\sigma = 0.08$	1.11 ± 0.03 $\sigma = 0.11$	1.01 ± 0.03 $\sigma = 0.12$
32,768	1.25 ± 0.02 $\sigma = 0.07$	1.37 ± 0.03 $\sigma = 0.12$	1.24 ± 0.03 $\sigma = 0.13$

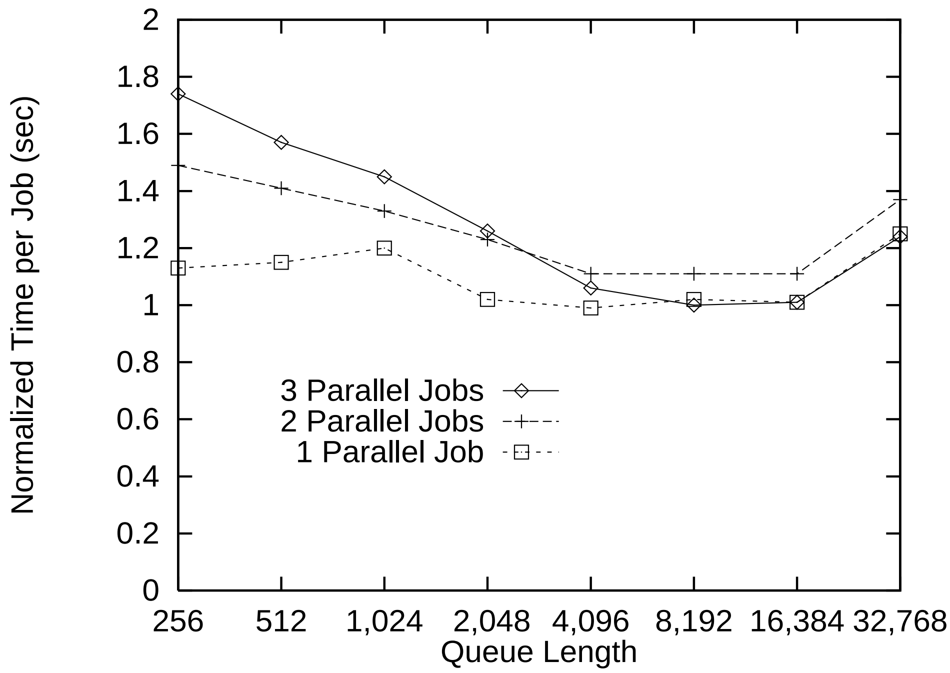

Figure 6.10: Normalized execution time in seconds per job for EM3D as a function of packet queue length. Times are normalized to 0.682 seconds.

Bulk Data Queue Length	1 Parallel Job	2 Parallel Jobs	3 Parallel Jobs
2	1.00 ± 0.01 $\sigma = 0.05$	1.28 ± 0.03 $\sigma = 0.15$	1.32 ± 0.03 $\sigma = 0.16$
4	1.01 ± 0.02 $\sigma = 0.06$	1.27 ± 0.04 $\sigma = 0.17$	1.27 ± 0.03 $\sigma = 0.16$
8	0.987 ± 0.008 $\sigma = 0.039$	1.25 ± 0.03 $\sigma = 0.14$	1.25 ± 0.04 $\sigma = 0.17$
16	1.00 ± 0.01 $\sigma = 0.05$	1.22 ± 0.03 $\sigma = 0.14$	1.23 ± 0.03 $\sigma = 0.16$
32	1.00 ± 0.02 $\sigma = 0.05$	1.25 ± 0.04 $\sigma = 0.16$	1.22 ± 0.03 $\sigma = 0.16$
64	1.03 ± 0.01 $\sigma = 0.05$	1.26 ± 0.04 $\sigma = 0.16$	1.22 ± 0.03 $\sigma = 0.16$
128	1.07 ± 0.02 $\sigma = 0.06$	1.27 ± 0.03 $\sigma = 0.14$	1.23 ± 0.03 $\sigma = 0.16$

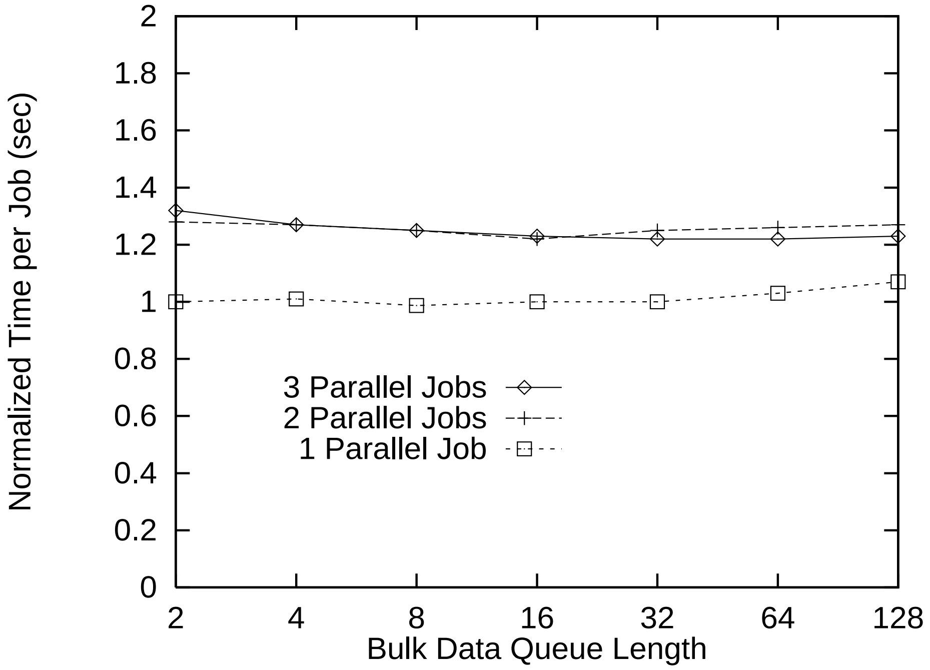

Figure 6.11: Normalized execution time in seconds per job for 3-D FFT as a function of bulk data queue length. Times are normalized to 0.676 seconds.

communication data and then yield the processor to the receiver, "moving" the data without actually crossing the memory interconnect. In a more limited form, a third process might knock the communication data from the cache, speeding retrieval by the receiver from another processor, as memory accesses are faster than cache-to-cache transfers.

We select a packet queue length of 4,096 entries based on these data. Smaller values increase queue overflows on multiprogrammed machines, while larger values increase the impact of cache pollution within each process. Despite a slight performance disadvantage for the multiprogrammed SAMPLE runs, we chose 4,096-entry rather than slightly longer queues to limit the memory footprint of the endpoint to under a megabyte.

We select a bulk data queue length of 16 primarily based on memory footprint. As is apparent from Figure 6.11, 3-D FFT does not provide much information for queue length selection, presumably because the communication pattern is very structured, with global barriers between each exchange of data.

Our selection of queue lengths completes the tuning necessary for the shared memory protocol. As demonstrated by this section and the previous one, tuning these two parameters requires a significant amount of time and effort, and produces a substantial amount of data for analysis. By expending this effort to develop the runtime system, however, we free the application programmer from the fairly onerous task of repeating these experiments for each new application or each new data abstraction developed. In the next sections, we compare the performance of our lock-free algorithm with alternative algorithms from the literature, then delve into our algorithm's performance in more detail.

6.3 Alternative Algorithms

We now compare the performance of our lock-free algorithm with a range of alternatives using the benchmark suite described in Chapter 5. This section first outlines a classification scheme for concurrent access algorithms and introduces the algorithms against which we compare our own. The remainder of the section presents and interprets results for each component of the benchmark suite in turn, studying the extremes of access contention with the LogGP parameters and the stress test, then investigating application performance on both dedicated and multiprogrammed machines. Our algorithm demonstrates competitive or superior performance in all of these trials.

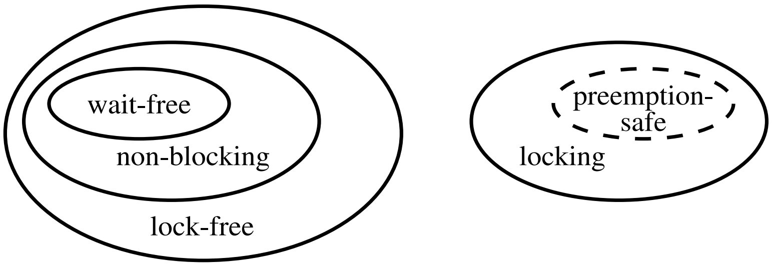

Figure 6.12: Concurrent algorithm hierarchy. Lock-free algorithms avoid mutual exclusion; non-blocking algorithms guarantee that some process makes progress; wait-free algorithms guarantee that all processes make progress. Preemption-safe locking uses operating system support to eliminate adverse interactions between locks and the scheduler.

6.3.1 Algorithm descriptions

The literature separates concurrent access algorithms into four categories. Traditional algorithms [RK79, And90, MCS91] are *locking*: a process must obtain a mutually exclusive lock to enter a critical section, thereby preventing other processes from entering concurrently. When such locks are used, a process stalled inside a critical section can delay all others for an arbitrary amount of time, a behavior termed blocking. *Non-blocking* algorithms [MP91, Her93, GC96, MS96] hence guarantee that some process makes progress in a finite amount of time, which implies that they do not enforce mutual exclusion. The third category, known as *wait-free*, strengthens the guarantee: every process makes progress in a finite number of its own time steps. Machine-level instructions, for example, are generally assumed to be wait-free. The remaining algorithms, such as our own, do not use locks but can still result in blocking behavior [Val94, BCL+95, KC95]. We follow Valois [Val94] in adopting the term *lock-free* for this fourth category. A diagram of this hierarchy appears in Figure 6.12. The diagram further identifies a division within locking algorithms, known as *preemption-safe* locking, with operating system support for reducing or eliminating the adverse affects of preemption [MS97]. Such support does not necessarily imply algorithmic differences.

Non-blocking algorithms are advantageous on multiprogrammed systems, as locks interact poorly with time-sharing. These algorithms follow a common design strategy and are simpler than their optimized locking counterparts. A typical non-blocking operation works as follows. A process reads a value from a data structure, performs all computation based on the value read, and inserts the results atomically into the data structure. If another process has changed the original value during the computation phase, the computed results are discarded and the operation starts again from the beginning. For many operations, the first steps of such an approach are the same as a sequential implementation, making non-blocking algorithms straightforward to construct. However, most modern architectures support only 64-bit synchronization primitives, and atomic insertion of results often necessitates an extra level of indirection in data structures.

On a dedicated system, the overhead of additional indirection and the cost of discarding optimistically completed work make generic non-blocking algorithms slower than locking algorithms. To address this drawback, numerous efforts apply problem-specific information to build more efficient solutions. This optimization process sometimes involves sacrificing true non-blocking behavior in favor of fast common-case performance. The result is a lock-free algorithm. In practice, these algorithms provide many of the advantages of non-blocking algorithms while avoiding most non-blocking overheads.

We compare our lock-free algorithm to a variety of spin locks, a preemption-safe lock, and a FIFO variant of our algorithm. We do not directly compare against non-blocking or wait-free algorithms, both because of the inherent overhead of these algorithms and because of the nature of our system. The use of a request-response communication paradigm makes true non-blocking behavior moot, as an endpoint that fails to respond blocks application progress. Non-blocking behavior also requires dynamic storage to avoid queue overflow, a factor that does not mesh well with our model of pre-allocated shared segments. In Chapter 9, we provide illustrative examples of non-blocking algorithms and a wait-free implementation of COMPARE&SWAP using FETCH&INCREMENT, and outline methods by which comparisons can be made using our system.

Locking algorithms

The concurrent access literature evaluates locking algorithms primarily in terms of the number of coherence transactions generated. Precluding starvation and providing fair access are also attractive qualities. We consider four spin lock algorithms drawn from [MCS91] and one preemption-safe locking algorithm based on the Solaris implementation of Posix mutexes.

We describe the spin lock algorithms in increasing order of complexity, commenting on the traffic generated with an invalidation-based coherence protocol. Each algo-

FETCH&INCREMENT(^*address*)
 value ← *address*^
 address^ ← *value* + 1
 return *value*

FETCH&DECREMENT(^*address*)
 value ← *address*^
 address^ ← *value* − 1
 return *value*

TEST&SET(^*address*)
 value ← *address*^
 address^ ← LOCKED
 return *value*

Figure 6.13: Pseudo-code for synchronization primitives. Each operation is atomic with respect to other memory accesses.

rithm relies on either TEST&SET (T&S) or FETCH&INCREMENT (F&I), which are atomic with respect to all other memory accesses. As mentioned earlier, TEST&SET(*address*) sets the value at an address to LOCKED and returns the previous value at the address. FETCH&INCREMENT(*address*) adds one to the value at an address and returns the previous value at the address. Figure 6.13 provides pseudo-code for these primitives. For the Sparc implementation, F&I is simulated with the non-blocking version of COMPARE&SWAP shown in Figure 3.7.

Message insertion with a locking algorithm is very similar to insertion with our lock-free algorithm. Packets cycle through the same states, using the same transitions. A sender transforms a packet from FREE to CLAIMED using CLAIMPACKET, which enforces atomicity through mutual exclusion. The sender then fills the packet and marks it as READY. The receiver accepts the message, then returns the packet to the FREE state, completing the cycle. For the locking algorithms, this three-state handshake shortens the critical section by extracting packet-filling.

Locking algorithms differ from our lock-free algorithm within CLAIMPACKET and CLAIMBULK. Pseudo-code for the locking procedures appears in Figure 6.14; each algorithm uses a distinct set of LOCK and UNLOCK operations. Under the protection of a queue's lock, CLAIMPACKET checks for a free packet at the tail of the queue. When available, the packet is claimed, the queue tail is advanced, and the packet is returned. The tail packet may

```
ClaimPacket(^q)
    while TRUE
        Lock(q^.mutex)
        index ← q^.tail
        if q^.packet[index].state = FREE
            q^.packet[index].state ← CLAIMED
            q^.tail ← (index + 1) mod Q_LENGTH
            Unlock(q^.mutex)
            return index
        Unlock(q^.mutex)
        (back off exponentially and poll)

ClaimBulk(^q)
    while TRUE
        Lock(q^.mutex)
        index ← q^.tail
        block ← q^.bulk_tail
        if q^.packet[index].state = FREE
            if q^.bulk[block].claimed = FREE
                q^.packet[index].state ← CLAIMED
                q^.bulk[block].claimed ← CLAIMED
                q^.tail ← (index + 1) mod Q_LENGTH
                q^.bulk_tail ← (block + 1) mod BULK_LENGTH
                Unlock(q^.mutex)
                q^.packet[index].bulk_index ← block
                return index
        Unlock(q^.mutex)
        (back off exponentially and poll)
```

Figure 6.14: Pseudo-code for claiming a packet from a queue q using mutual exclusion. ClaimBulk claims both a packet and a bulk data block.

```
LOCK(^mutex)            (Test & Set)
    repeat
    until TEST&SET(mutex) = UNLOCKED

LOCK(^mutex)            (Test & Test & Set)
    while TRUE
        if TEST&SET(mutex) = UNLOCKED
            return
        repeat
        until mutex^ = UNLOCKED

UNLOCK(^mutex)
    mutex^ ← UNLOCKED
```

Figure 6.15: Pseudo-code for the Test & Set and Test & Test & Set algorithms.

```
LOCK(^mutex)
    number ←FETCH&INCREMENT(mutex^.ticket)
    repeat
    until mutex^.service = number

UNLOCK(^mutex)
    mutex^.service ← mutex^.service + 1
```

Figure 6.16: Pseudo-code for the ticket lock algorithm.

already be claimed—when the queue is full, for example—and, in this case, the operation stalls until it becomes available. CLAIMBULK has the same form as CLAIMPACKET but operates on both the packet and bulk data queues inside the critical section. CLAIMBULK also records the index of the associated data block in the packet. As with the lock-free algorithm, the claim operations accept messages when a queue is full to avoid deadlock.

The first locking algorithm, Test & Set, uses a simple, one-bit lock, as shown in Figure 6.15. Waiting processes continuously generate invalidation and read-request coherence transactions. The second algorithm, Test & Test & Set, appears in the same figure. Test & Test & Set waits until a lock is released before making another attempt to obtain it, generating coherence traffic only when a lock is manipulated.

Third, the ticket lock [RK79], further reduces cache-coherence traffic by ordering the processes waiting on a lock. To obtain a lock, a process obtains a ticket and waits for

```
LOCK(^mutex)
    number ←FETCH&INCREMENT(mutex^.index) mod NUM_PROCESSES
    repeat
    until mutex^.held[number] = LOCKED
    mutex^.slot[MY_PROCESS] = number

UNLOCK(^mutex)
    number ← mutex^.slot[MY_PROCESS]
    next ← (number + 1) mod NUM_PROCESSES
    mutex^.held[number] ← UNLOCKED
    mutex^.held[next] ← LOCKED
```

Figure 6.17: Pseudo-code for the Anderson lock algorithm. MY_PROCESS is a process identifier between 0 and (NUM_PROCESSES−1).

a service counter to show its ticket number. The ticket counter is incremented atomically to ensure that each process receives a different ticket. When a process releases a lock, it increments the service counter to allow the next process waiting on the lock to proceed. As with previous algorithms, the release of a ticket lock incurs invalidation and read-request coherence transactions for each waiting process. No further traffic is necessary, however, as the next process acquires the lock when it rereads the service counter. Pseudo-code appears in Figure 6.16. Given hardware support for wait-free F&I, the ticket lock precludes starvation and is strictly fair [MCS91]. Our Sparc implementation, based on a simulated, non-blocking F&I, does not prevent starvation—it is fair only to processes that successfully obtain tickets.

Fourth is the Anderson lock [And90], which improves on the ticket lock by dividing the service counter into a separate flag for each process waiting on the lock, eliminating contention when a lock moves from one process to the next. As shown in Figure 6.17, the lock operation obtains a slot assignment (ticket) with F&I, then waits for the assigned slot to contain a lock indicator. By retaining the form of the ticket counter, the Anderson lock preserves the fairness property of the ticket lock. However, the maximum number of processes must be known in advance to avoid multiple assignments to a slot in the divided service counter. When releasing a lock, a process moves the lock indicator from its assigned slot into the next. As only a single process watches the slot that next receives the lock, Anderson locks generate fewer read-request transactions when a process releases a lock.

These four algorithms, together known as *spin locks*, have several drawbacks in practice. As the name implies, spin locks spin in a tight loop while waiting for a lock, potentially wasting valuable cycles. Spin locks also interact poorly with the operating system scheduler and admit deadlock when used with preemptive threads. A scheduler can artificially extend the length of a critical section by descheduling a process during its execution, thereby preventing progress by other processes for relatively long periods. A high-priority thread that preempts a low-priority thread holding a lock then tries to obtain the lock creates a deadlock. Preemption-safe locking addresses these problems [MS97].

Numerous preemption-safe solutions exist, and many are supported in modern thread packages through integration with the operating system. The Solaris implementation of Posix mutexes, our final locking algorithm, exemplifies these solutions. A process that tries to obtain a lock held by another process enqueues itself on the lock and relinquishes its processor. Priority inheritance allows a low priority process in a critical section to inherit the priority of a high priority process waiting to enter that section. Unfortunately, the same close coupling with the operating system results in performance disadvantages on dedicated systems. For Posix mutexes, the Lock and Unlock calls translate directly into their standard counterparts, `pthread_mutex_lock` and `pthread_mutex_unlock`.

A fair comparison between the locking algorithms and our lock-free algorithm requires that we lay out the lock data in a way that delivers the best performance. We achieve this end by placing the lock on a private cache line for the Test & Set and Test & Test & Set algorithms. We found that the ticket lock delivers slightly better performance when both counters occupy a single cache line, whereas the literature implies the use of separate lines. For the Anderson lock, we placed the ticket counter and each slot on a separate cache line, as the potential advantage of that algorithm lies in the separation of the slots. Finally, we placed the Posix mutex structure on a line of its own.

Lock-free algorithms

A variety of lock-free, array-based queue algorithms appear in the literature, including some very similar to our own. We describe several of those algorithms, then introduce a modified form of our lock-free algorithm that provides FIFO semantics.

One algorithm was developed independently by two groups working with the Cray T3D, as described in Brewer *et al.* [BCL+95] and Karamcheti and Chien [KC95]. The algorithm uses Fetch&Increment to claim queue entries from a static queue, but relies on the receiver to reset the queue after all entries have been claimed and processed. We avoid this boundary condition through the use of an additional synchronization primitive.

A second algorithm, based on work with the NYU Ultracomputer [GLR83], was designed for a many-to-many queue in which overflow and underflow must be detected explicitly. Figure 6.18 presents pseudo-code for the Enqueue and Dequeue operations. These operations maintain a count of full queue entries as a range of two values, *upper* and *lower*. Enqueue begins with a check for queue overflow. The second conditional attempts to reserve one of the remaining queue entries, atomically decrementing the upper bound on full entries if another process has already taken the last one. The first conditional is necessary to prevent the system from entering a perpetual state of false overflow, as might occur if a number of processes repeatedly executed the second conditional on a nearly-full queue. The rest of the Enqueue operation is practically equivalent to that used in our algorithm, although it uses semaphores to manage contention for individual packets and increments the lower bound on full entries after completing an insertion. The form of Dequeue mirrors that of Enqueue. The explicit overflow and underflow support adds overhead to the Ultracomputer algorithm, as does the ability to allow concurrent access for multiple receivers. We needed neither of these features, hence our algorithm obtains higher performance.

Valois [Val94] presents a third algorithm that uses explicit head and tail symbols in the queue. A process enqueues a new element by atomically replacing adjacent array elements, changing a (TAIL, EMPTY) pair to a (<new value>, TAIL) pair. This approach limits queue entries to a size that can be manipulated with Compare&Swap primitives, for our system requiring that the queue be an array of pointers. Locating the TAIL symbol also incurs additional overhead, as the algorithm maintains only a hint as to its location in the array. Most importantly, however, the algorithm requires unaligned Compare&Swap primitives, making it "infeasible on a real machine."

The last lock-free algorithm to be discussed is a modified form of our own. Providing FIFO semantics is not imperative for a correct implementation of the AM-II specification, but might prove useful for higher-level software built to the AM-II interface. To obtain such semantics, we take advantage of the total ordering on message insertions defined by the atomic packet assignment operation. As shown in Figure 6.19, we interpret the upper bits of the queue tail as an epoch number for insertions. Contention for individual packets is then resolved in the order defined by the epoch numbers. After a receiver frees a message

```
ENQUEUE(^q, ^data)
    if q^.upper >= Q_LENGTH
        return OVERFLOW
    if FETCH&INCREMENT(q^.upper) >= Q_LENGTH
        FETCH&DECREMENT(q^.upper)
        return OVERFLOW
    index ← FETCH&INCREMENT(q^.tail) mod Q_LENGTH
    while TRUE
        if q^.packet[index].enq_sema) > 0
            if FETCH&DECREMENT(q^.packet[index].enq_sema) > 0
                q^.packet[index].data ← data^
                FETCH&INCREMENT(q^.packet[index].deq_sema)
                FETCH&INCREMENT(q^.lower)
                return SUCCESS
            else
                FETCH&INCREMENT(q^.packet[index].enq_sema)

DEQUEUE(^q, ^data)
    if q^.lower < 1
        return UNDERFLOW
    if FETCH&DECREMENT(q^.lower) < 1
        FETCH&INCREMENT(q^.lower)
        return UNDERFLOW
    index ← FETCH&INCREMENT(q^.head) mod Q_LENGTH
    while TRUE
        if q^.packet[index].deq_sema) > 0
            if FETCH&DECREMENT(q^.packet[index].deq_sema) > 0
                data^ ← q^.packet[index].data
                FETCH&INCREMENT(q^.packet[index].enq_sema)
                FETCH&DECREMENT(q^.upper)
                return SUCCESS
            else
                FETCH&INCREMENT(q^.packet[index].deq_sema)
```

Figure 6.18: Pseudo-code for Ultracomputer lock-free queue algorithm. The number of items in the queue is in the range $[lower, upper]$. Contention for individual packets is managed through enqueue and dequeue semaphores.

```
ClaimPacket(^q)
    assignment ← Fetch&Increment(q^.tail)
    epoch      ← assignment div Q_LENGTH
    index      ← assignment mod Q_LENGTH
    while TRUE
        if q^.packet[index].epoch = epoch
            return index
        (back off exponentially and poll)

ClaimBulk(^q)
    assignment ← Fetch&Increment(q^.bulk_tail)
    epoch      ← assignment div BULK_LENGTH
    block      ← assignment mod BULK_LENGTH
    while TRUE
        if q^.bulk[block].epoch = epoch
            index ← ClaimPacket(q)
            q^.packet[index].bulk_index ← block
            return index
        (back off exponentially and poll)
```

Figure 6.19: Pseudo-code for a FIFO lock-free queue. The epoch numbers guarantee the FIFO property. The receiver must increment the epoch for each packet and each block received.

packet, it increments the packet's epoch number, allowing the next sender to fill the packet again. The Ultracomputer work [GLR83] suggests a similar alternative for managing packet contention.

6.3.2 Communication parameters

We now apply the benchmarking methodology developed in Chapter 5 to compare the performance of the algorithms just described. We first measure point-to-point communication performance with our LogGP benchmarks. These benchmarks use one process as an RPC server and a second as a client to illustrate performance in the absence of contention. Parameter values and round-trip times in microseconds for all algorithms appear in Table 6.3.

With the exception of the Posix mutex, the various algorithms are roughly equivalent. A comparison between Test & Set and Test & Test & Set provides a reasonable estimate of the actual accuracy of the measurements, as the two algorithms are nearly the same in the absence of contention. The receive overhead provides a second estimate. None

Parameter	Test & Set	Test & Test & Set	Ticket Lock	Anderson Lock
Latency (L)	-0.59 ± 0.01	-0.53 ± 0.01	-0.58 ± 0.01	-0.61 ± 0.01
Send Overhead (o_s)	1.824 ± 0.005	1.714 ± 0.005	1.764 ± 0.004	1.884 ± 0.005
Receive Overhead (o_r)	1.49 ± 0.02	1.52 ± 0.02	1.68 ± 0.02	1.49 ± 0.02
Gap (g)	2.953 ± 0.003	2.937 ± 0.003	3.122 ± 0.003	2.988 ± 0.002
Gap per Byte (G)	5.991 ± 0.008 nsec/B	5.992 ± 0.005 nsec/B	5.986 ± 0.009 nsec/B	6.20 ± 0.02 nsec/B
Bandwidth ($1/G$)	159.2 ± 0.2 MB/sec	159.2 ± 0.2 MB/sec	159.3 ± 0.3 MB/sec	153.7 ± 0.3 MB/sec
Round-trip Time (RTT)	5.439 ± 0.003	5.412 ± 0.002	5.737 ± 0.004	5.510 ± 0.003

Parameter	Lock-Free	Lock-Free FIFO	Posix Mutex
Latency (L)	-0.50 ± 0.01	-0.60 ± 0.01	-0.42 ± 0.01
Send Overhead (o_s)	1.819 ± 0.005	1.883 ± 0.005	2.160 ± 0.005
Receive Overhead (o_r)	1.47 ± 0.02	1.49 ± 0.02	1.57 ± 0.02
Gap (g)	3.005 ± 0.002	3.240 ± 0.004	3.418 ± 0.006
Gap per Byte (G)	6.035 ± 0.006 nsec/B	5.98 ± 0.02 nsec/B	6.044 ± 0.005 nsec/B
Bandwidth ($1/G$)	158.0 ± 0.2 MB/sec	159.5 ± 0.3 MB/sec	157.8 ± 0.2 MB/sec
Round-trip Time (RTT)	5.588 ± 0.004	5.542 ± 0.003	6.634 ± 0.007

Table 6.3: LogGP parameters and round-trip times in microseconds. Negative latencies indicate overlap in time between the send and receive overheads.

of the algorithms alters the code for message reception, thus the receive overhead for each algorithm should be the same. The round-trip times for the more complex spin locks (the ticket lock and the Anderson lock) and for the lock-free algorithms are a few percent larger than those of the simpler locks, but the Posix mutex time is roughly 20% larger. Due to interactions with the operating system, Posix mutex send overhead is also about 20% larger than that of other algorithms, and gap is about 14% larger.

The negative values for latency indicate overlap in time between the send and receive overheads [LC95], in this instance due to the poll operation and to differences between send overhead for request and reply messages. The benchmark that measures the sum of send and receive overheads forcibly separates the two with a large delay. The round-trip

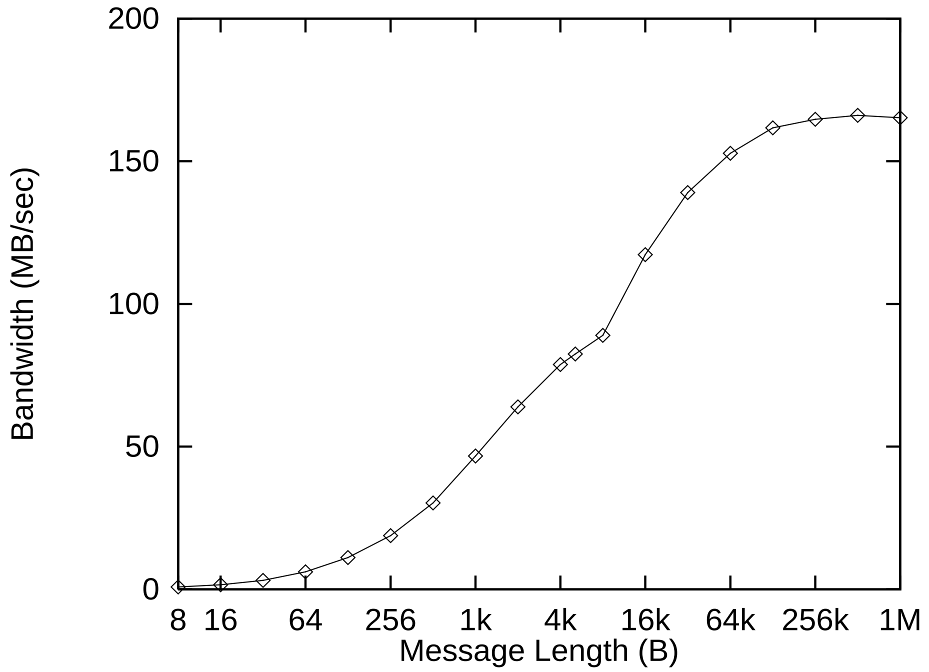

Figure 6.20: Shared memory protocol bandwidth. The cusp occurs at the transition from one- to two-packet messages. The receiver and the sender are not perfectly matched, thus the half-power point occurs before the cusp, around 5.1 kB.

time benchmark allows overlap with both requests and replies; a message recipient can initiate the poll operation before the sender changes the packet state, so long as the state change occurs before the state check in the poll. In fact, a message recipient can even initiate a coherence transaction to bring the packet into its cache before the packet is ready; again the only constraint is that the packet be marked as ready before the sender's cache yields its exclusive access rights. Adding an 84-cycle cache-to-cache transfer to a 51-cycle application-level poll operation (see Chapter 7) and assuming that on average half of each poll is hidden in each direction, we predict an overlap of roughly 0.8 microseconds. Reply send overhead is also likely to be smaller than request send overhead, as replies need not check tags or arguments. Half of this difference, or roughly 0.15 microseconds (see Section 6.4.1 below), appears as decreased latency. A third interaction damps the overlap effect, however: when a recipient initiates the coherence transaction too early, the sender must issue another invalidation before completing its message insertion. The likelihood of this event depends on the time required to finish claiming and filling the packet, leading to the slight differences in latency between the algorithms.

Gap per byte and bandwidth have less variation than the other parameters, as the cost to insert long messages amortizes the concurrent access overhead of the algorithms. Peak realized bandwidths range from about 154 MB/sec (Anderson lock) to 160 MB/sec (lock-free FIFO), or roughly 85% of the memory copy rate.

Figure 6.20 reports realized bandwidth with the lock-free algorithm as a function of message length. The graph assumes a shape typical of fragmented and pipelined communication on a half-log scale: bandwidth first grows exponentially, then inflects and approaches a limit asymptotically, and finally drops down discontinuously and increases slope when the application must generate an additional fragment. For our communication layer, these discontinuities occur at 8 kB intervals; the data points chosen for the figure reveal only the first discontinuity as a cusp. The half-power point occurs at 5.1 kB, before the transition from one to two fragments per message.

The location of the half-power point relative to the first discontinuity depends on the balance between the time spent on each fragment by the sender and the receiver. Denote by t_s the time spent by the sender in sending a single fragment, *i.e.*, a bulk transfer message of 8 kB. Similarly, denote by t_r the time spent by the receiver in receiving a single fragment. Finally, denote by t_c the time spent returning the last acknowledgement, a short message, to the sender. We can then write the realized bandwidth B for a message of N fragments as follows:

$$B = \frac{N\ 8\ \text{kB}}{N\max(t_s, t_r) + \min(t_s, t_r) + t_c}$$

or, in the limit of large N, $$B_{asymp} = \frac{8\ \text{kB}}{\max(t_s, t_r)}$$

A one-fragment message can then achieve more than half of the asymptotic limit whenever t_s differs from t_r by at least t_c. Assuming that $|t_s - t_r| \geq t_c$, we write

$$\frac{8\ \text{kB}}{\max(t_s, t_r) + \min(t_s, t_r) + t_c} \geq \frac{8\ \text{kB}}{2\max(t_s, t_r)} = \frac{1}{2}B_{asymp}$$

Using our lock-free algorithm, a sender incurs significantly more overhead than a receiver for bulk transfers, and the half-power point occurs before the first discontinuity. For the network protocol, t_c is much larger, and the half-power point occurs above 8 kB.

6.3.3 Stress test

We next report the performance of many-to-one communication using our contention stress test. Numerical results for up to eight processors appear in Table 6.4, and

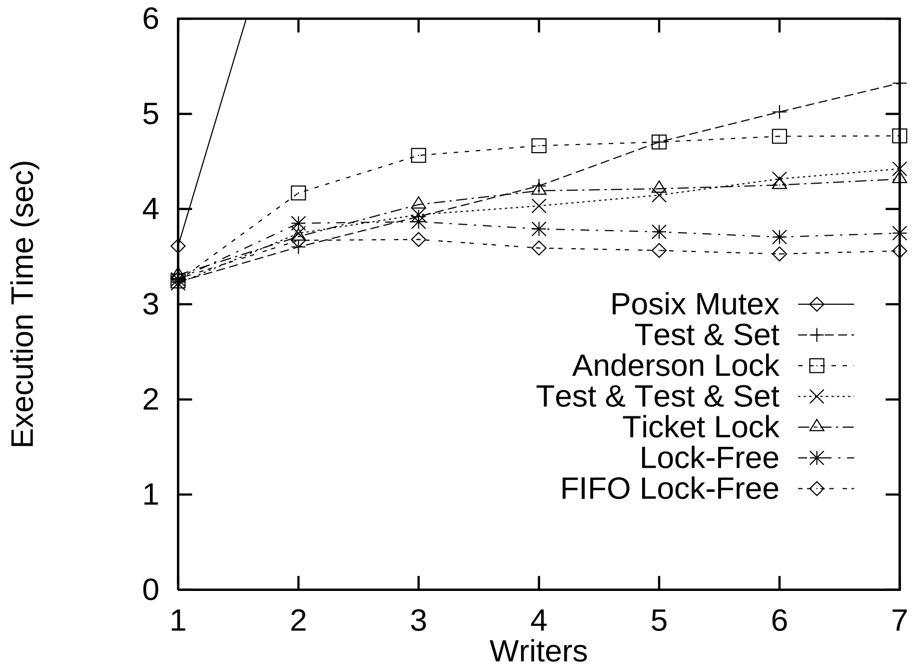

Figure 6.21: Graph of communication stress test execution time in seconds. The values can also be interpreted as microseconds per message. Execution times for the simpler spin locks rise across the range reported. Other algorithms quickly flatten out, with remaining differences due to the number of cache lines that move between processors.

a graphical version appears in Figure 6.21. The vertical axis in the figure represents execution time in seconds, but can equivalently be interpreted as microseconds per message. The test with a single writer generates no contention and is equivalent to the LogP gap measurement; again the algorithms are roughly equal except for the Posix mutex, which incurs higher overhead.

Execution time for all algorithms jumps upward in moving to two writers, as each algorithm moves at least one cache line between processors when multiple writers exist. For larger numbers of writers, the character of stress test performance takes one of two basic shapes. Execution times for Test & Set and Test & Test & Set rise roughly linearly as contention increases due to data thrashing between caches. The other algorithms flatten out after their initial increases.

The ticket lock, for example, rises only slightly due to interference between the ticket and service counters, which reside on the same cache line in our implementation. The Anderson lock provides separate cache lines for each process queued on a lock, resulting in a

	Test & Set	Test & Test & Set	Ticket Lock	Anderson Lock
1 writer	3.232 ± 0.005 $\sigma = 0.024$	3.218 ± 0.006 $\sigma = 0.026$	3.306 ± 0.004 $\sigma = 0.017$	3.245 ± 0.006 $\sigma = 0.029$
2 writers	3.602 ± 0.005 $\sigma = 0.023$	3.744 ± 0.006 $\sigma = 0.026$	3.711 ± 0.005 $\sigma = 0.023$	4.169 ± 0.007 $\sigma = 0.032$
3 writers	3.919 ± 0.006 $\sigma = 0.026$	3.940 ± 0.005 $\sigma = 0.025$	4.046 ± 0.005 $\sigma = 0.024$	4.563 ± 0.008 $\sigma = 0.036$
4 writers	4.242 ± 0.007 $\sigma = 0.030$	4.033 ± 0.007 $\sigma = 0.034$	4.193 ± 0.005 $\sigma = 0.021$	4.665 ± 0.006 $\sigma = 0.028$
5 writers	4.705 ± 0.006 $\sigma = 0.025$	4.146 ± 0.007 $\sigma = 0.034$	4.214 ± 0.005 $\sigma = 0.024$	4.704 ± 0.006 $\sigma = 0.025$
6 writers	5.019 ± 0.005 $\sigma = 0.023$	4.317 ± 0.008 $\sigma = 0.036$	4.251 ± 0.005 $\sigma = 0.020$	4.765 ± 0.004 $\sigma = 0.019$
7 writers	5.323 ± 0.007 $\sigma = 0.033$	4.422 ± 0.007 $\sigma = 0.032$	4.315 ± 0.005 $\sigma = 0.022$	4.768 ± 0.007 $\sigma = 0.033$

	Lock-Free	Lock-Free FIFO	Posix Mutex
1 writer	3.26 ± 0.01 $\sigma = 0.05$	3.268 ± 0.007 $\sigma = 0.030$	3.612 ± 0.005 $\sigma = 0.022$
2 writers	3.851 ± 0.004 $\sigma = 0.017$	3.671 ± 0.004 $\sigma = 0.017$	7.82 ± 0.06 $\sigma = 0.27$
3 writers	3.866 ± 0.004 $\sigma = 0.016$	3.681 ± 0.004 $\sigma = 0.017$	10.55 ± 0.05 $\sigma = 0.22$
4 writers	3.791 ± 0.003 $\sigma = 0.013$	3.591 ± 0.004 $\sigma = 0.017$	11.29 ± 0.04 $\sigma = 0.16$
5 writers	3.762 ± 0.003 $\sigma = 0.013$	3.565 ± 0.004 $\sigma = 0.017$	11.22 ± 0.03 $\sigma = 0.12$
6 writers	3.706 ± 0.004 $\sigma = 0.015$	3.528 ± 0.004 $\sigma = 0.018$	11.25 ± 0.02 $\sigma = 0.09$
7 writers	3.746 ± 0.006 $\sigma = 0.028$	3.560 ± 0.007 $\sigma = 0.033$	11.33 ± 0.02 $\sigma = 0.08$

Table 6.4: Communication stress test times in seconds.

flatter curve but incurring a larger penalty in the transition to multiple writers. Whereas the other spin locks move only the lock and the queue tail between senders, the Anderson lock must move the lock counter, a lock slot, and the queue tail. Posix mutex performance has the same character as that of the Anderson lock, but context switches result in significantly higher absolute execution times. The line for Posix mutexes goes off the scale in Figure 6.21 before reaching 2 writers.

Both forms of our lock-free algorithm flatten out below the locking algorithm lines. For our algorithms, only the queue tail need move between processor caches. We attribute the gap in execution time between the FIFO and non-FIFO forms to the additional synchronization primitive (CAS) used by the non-FIFO form. Execution time in fact decreases slightly as the number of writers increases due to a gap reduction effect found in many message-passing systems. The overhead of a poll operation is amortized over all messages accepted by that poll, making the gap for many-to-one communication lower than that for one-to-one communication. For both forms of our lock-free algorithm, gap reduction dominates the small increase in execution time due to contention.

6.3.4 Application suite

The Split-C applications utilize a variety of communication patterns and generate contention levels between the extremes investigated by the previous two benchmarks. Execution times for five application runs using only the shared memory protocol on one SMP appear in Table 6.5.

Our lock-free algorithm generally demonstrates performance comparable or better than that of the other algorithms. Applications with very structured communication such as 3-D FFT or with nearest-neighbor communication such as CON/comp are independent of the choice of concurrent access algorithm.

CON/comm illustrates minor distinctions due to the use of many-to-one communication. Three of the locking algorithms perform well enough for their confidence intervals to overlap with that of our lock-free algorithm at 1.89 seconds. CON/comm is 10% slower when using the Anderson lock, however, executing in 2.08 seconds. The Posix mutex results in an even slower execution: 2.16 seconds, or 15% slower than our lock-free algorithm.

EM3D, with many cross-processor dependencies, shows a more significant separation. The performance of the algorithms under contention clearly interacts with the

	SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
Test & Set	0.891 ± 0.005 $\sigma = 0.024$	1.12 ± 0.02 $\sigma = 0.07$	1.94 ± 0.04 $\sigma = 0.18$	0.685 ± 0.008 $\sigma = 0.040$	0.79 ± 0.02 $\sigma = 0.06$
Test & Test&Set	0.875 ± 0.006 $\sigma = 0.028$	1.078 ± 0.008 $\sigma = 0.039$	1.94 ± 0.04 $\sigma = 0.19$	0.687 ± 0.007 $\sigma = 0.034$	0.836 ± 0.009 $\sigma = 0.043$
Ticket Lock	0.893 ± 0.007 $\sigma = 0.034$	1.12 ± 0.02 $\sigma = 0.06$	1.93 ± 0.04 $\sigma = 0.17$	0.685 ± 0.006 $\sigma = 0.030$	0.817 ± 0.008 $\sigma = 0.037$
Anderson Lock	1.027 ± 0.007 $\sigma = 0.034$	1.14 ± 0.02 $\sigma = 0.07$	2.08 ± 0.05 $\sigma = 0.20$	0.681 ± 0.007 $\sigma = 0.030$	0.905 ± 0.009 $\sigma = 0.040$
Lock-Free	0.762 ± 0.004 $\sigma = 0.017$	1.11 ± 0.02 $\sigma = 0.05$	1.89 ± 0.04 $\sigma = 0.19$	0.677 ± 0.008 $\sigma = 0.037$	0.682 ± 0.009 $\sigma = 0.043$
FIFO Lock-Free	0.791 ± 0.007 $\sigma = 0.032$	1.09 ± 0.02 $\sigma = 0.07$	1.86 ± 0.04 $\sigma = 0.17$	0.69 ± 0.01 $\sigma = 0.05$	0.68 ± 0.02 $\sigma = 0.05$
Posix Mutex	2.43 ± 0.07 $\sigma = 0.34$	1.12 ± 0.02 $\sigma = 0.06$	2.16 ± 0.04 $\sigma = 0.17$	0.70 ± 0.02 $\sigma = 0.06$	1.284 ± 0.009 $\sigma = 0.042$

Table 6.5: Application execution times in seconds.

application-level blocking effect mentioned earlier, allowing some processes to begin field computations before others receive all of their updates. EM3D executes in 0.682 seconds using our lock-free algorithm. Execution times with the spin lock algorithms range from 0.79 seconds to 0.905 seconds, a slowdown of 16% to 33%. Posix mutexes again result in the worst performance, requiring 1.284 seconds, or 88% more than our lock-free algorithm.

Although the contention for any single queue is necessarily less than that measured by the stress test, the more complex communication patterns employed by applications can result in greater differences in performance. The unstructured all-to-all communication used in the last application run, SAMPLE, demonstrates this effect. The run executes in 0.762 seconds using the lock-free algorithm. Spin lock execution times range from 0.875 seconds to 1.027 seconds, or 15% to 35% slower. SAMPLE requires 2.43 seconds when using Posix mutexes, more than a factor of three slower than with our algorithm. The impact of operating system support in this case goes beyond the overhead of the interaction itself. The resulting context switches increase the potential for process migration and dramatically increase the execution time.

6.3.5 Multiprogrammed machines

Our algorithm is clearly advantageous on a dedicated machine, providing competitive or superior performance for applications with a range of communication patterns. We now conclude our investigation of alternative algorithms with a presentation of application performance on multiprogrammed machines.

The data appear in Figures 6.22 through 6.26, with one figure per application. The upper portion of each figure provides a table of execution times in seconds per job for four combinations of parallel and sequential jobs. As detailed in our methodology in Chapter 5, each parallel job is an eight-process application, and each sequential "job" is a set of eight independent sequential processes designed to stress the memory system. The dedicated data, labeled "1 Parallel Job," are also included for comparison. The lower portion of each figure presents the same data in graphical form. The order of algorithms from left to right in each group of seven bars corresponds to the order used throughout the thesis and replicated in the legend from top to bottom. Unfortunately, absolute execution times between application runs varied too widely to allow the use of a single time scale for all of the charts.

Applications with very structured communication again demonstrate little variation due to the choice of algorithm. 3-D FFT (Figure 6.25), for example, executes in the same time within the confidence intervals for each algorithm. As noted during our discussion of parameter tuning earlier in this chapter, 3-D FFT execution time per job is about 23% larger when multiple parallel jobs compete; execution time with the lock-free algorithm, for example, rises from 0.677 seconds on a dedicated machine to 0.83 seconds on a multiprogrammed one. Sequential jobs have significantly more impact on parallel execution time, as they do not cooperate in yielding processors. Adding one group of sequential processes slows a parallel job using the lock-free algorithm by 49%, and two groups slows a parallel job by 166%.

CON/comp (Figure 6.23) shows some variation across algorithms. In the presence of two groups of sequential processes, execution times with the ticket lock and the Anderson lock are respectively 29% and 36% larger than with the lock-free algorithm. For the other multiprogramming combinations measured, all algorithms are equivalent within the confidence intervals. Multiple parallel jobs have less impact than with 3-D FFT: multiprogrammed execution time per job is 6% larger than the dedicated value. Sequential jobs also have relatively little impact. The execution time of a parallel job increases by 39% when competing with one group of sequential processes and by 26% when competing with two groups.

The many-to-one communication in CON/comm (Figure 6.24) separates the al-

	1 Parallel Job	2 Parallel Jobs	1 Parallel, 1 Sequential Job	3 Parallel Jobs	1 Parallel, 2 Sequential Jobs
Test & Set	0.891 ± 0.005 $\sigma = 0.024$	1.46 ± 0.05 $\sigma = 0.21$	1.58 ± 0.08 $\sigma = 0.36$	1.52 ± 0.04 $\sigma = 0.20$	1.75 ± 0.06 $\sigma = 0.27$
Test & Test & Set	0.875 ± 0.006 $\sigma = 0.028$	1.42 ± 0.05 $\sigma = 0.22$	1.55 ± 0.07 $\sigma = 0.34$	1.54 ± 0.05 $\sigma = 0.23$	1.73 ± 0.05 $\sigma = 0.25$
Ticket Lock	0.893 ± 0.007 $\sigma = 0.034$	took too long to run			
Anderson Lock	1.027 ± 0.007 $\sigma = 0.034$	took too long to run			
Lock-Free	0.762 ± 0.004 $\sigma = 0.017$	0.97 ± 0.03 $\sigma = 0.14$	1.06 ± 0.06 $\sigma = 0.26$	1.01 ± 0.03 $\sigma = 0.13$	1.24 ± 0.05 $\sigma = 0.20$
FIFO Lock-Free	0.791 ± 0.007 $\sigma = 0.032$	1.03 ± 0.04 $\sigma = 0.17$	1.12 ± 0.07 $\sigma = 0.33$	1.08 ± 0.04 $\sigma = 0.17$	1.31 ± 0.05 $\sigma = 0.22$
Posix Mutex	2.43 ± 0.07 $\sigma = 0.34$	1.61 ± 0.05 $\sigma = 0.23$	2.42 ± 0.09 $\sigma = 0.43$	1.04 ± 0.03 $\sigma = 0.14$	1.92 ± 0.06 $\sigma = 0.27$

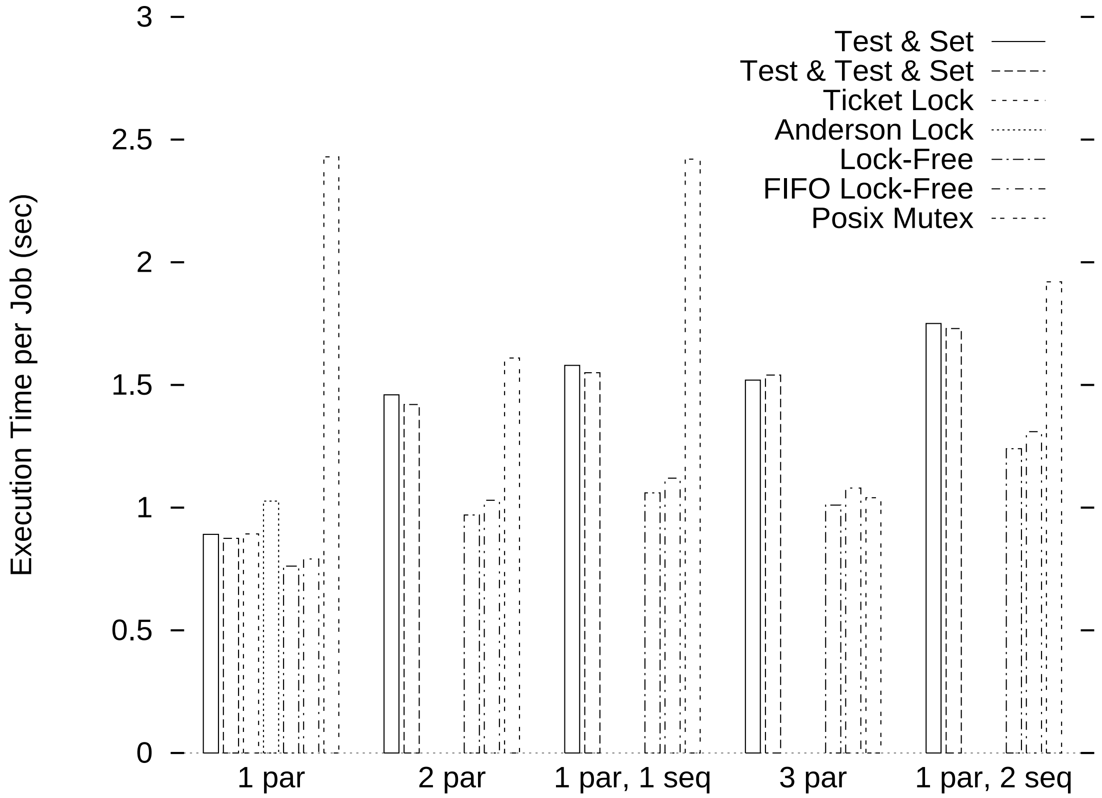

Figure 6.22: SAMPLE execution times in seconds per job.

	1 Parallel Job	2 Parallel Jobs	1 Parallel, 1 Sequential Job	3 Parallel Jobs	1 Parallel, 2 Sequential Jobs
Test & Set	1.12 ± 0.02 $\sigma = 0.07$	1.21 ± 0.03 $\sigma = 0.14$	1.51 ± 0.07 $\sigma = 0.34$	1.21 ± 0.03 $\sigma = 0.14$	1.37 ± 0.04 $\sigma = 0.17$
Test & Test & Set	1.078 ± 0.008 $\sigma = 0.039$	1.17 ± 0.03 $\sigma = 0.12$	1.60 ± 0.08 $\sigma = 0.38$	1.15 ± 0.03 $\sigma = 0.14$	1.46 ± 0.04 $\sigma = 0.19$
Ticket Lock	1.12 ± 0.02 $\sigma = 0.06$	1.21 ± 0.03 $\sigma = 0.14$	1.67 ± 0.08 $\sigma = 0.39$	1.21 ± 0.03 $\sigma = 0.15$	1.8 ± 0.2 $\sigma = 0.6$
Anderson Lock	1.14 ± 0.02 $\sigma = 0.07$	1.23 ± 0.03 $\sigma = 0.15$	1.69 ± 0.08 $\sigma = 0.40$	1.23 ± 0.03 $\sigma = 0.14$	1.9 ± 0.2 $\sigma = 0.8$
Lock-Free	1.11 ± 0.02 $\sigma = 0.05$	1.18 ± 0.03 $\sigma = 0.13$	1.54 ± 0.07 $\sigma = 0.34$	1.18 ± 0.03 $\sigma = 0.15$	1.40 ± 0.04 $\sigma = 0.16$
FIFO Lock-Free	1.09 ± 0.02 $\sigma = 0.07$	1.18 ± 0.03 $\sigma = 0.13$	1.56 ± 0.08 $\sigma = 0.36$	1.13 ± 0.03 $\sigma = 0.12$	1.34 ± 0.04 $\sigma = 0.17$
Posix Mutex	1.12 ± 0.02 $\sigma = 0.06$	1.19 ± 0.03 $\sigma = 0.13$	1.51 ± 0.07 $\sigma = 0.34$	1.18 ± 0.03 $\sigma = 0.12$	1.37 ± 0.04 $\sigma = 0.17$

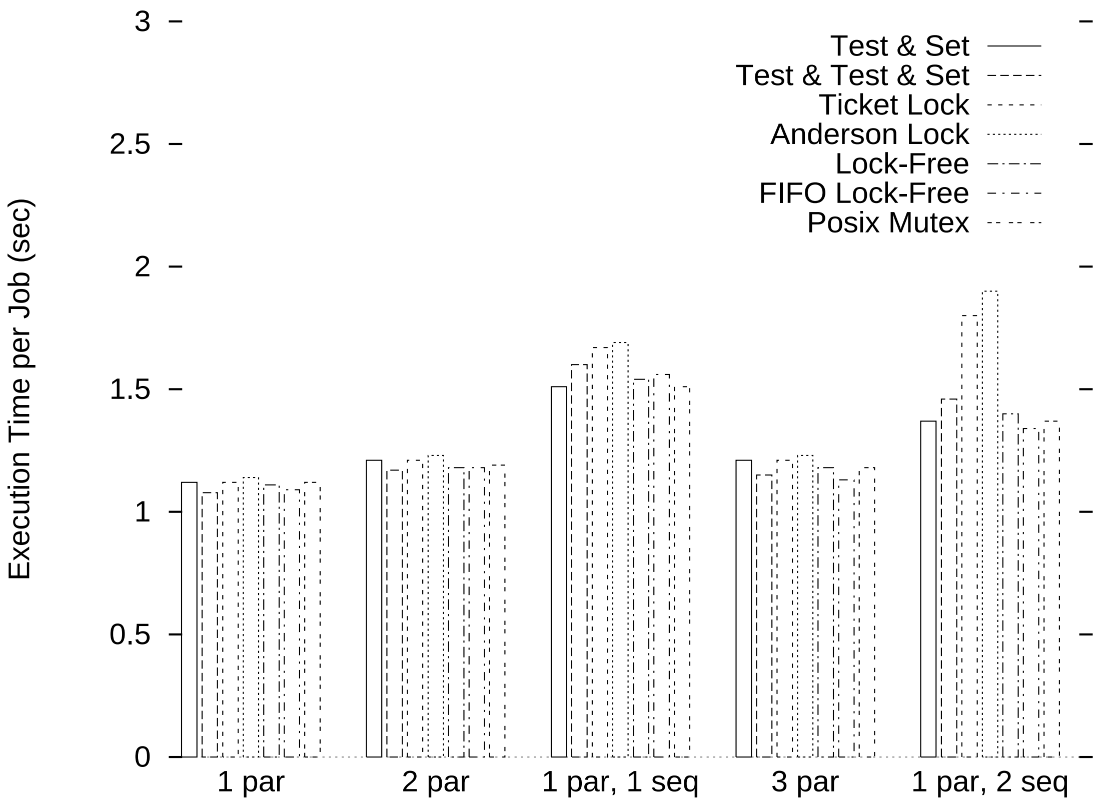

Figure 6.23: CON/comp execution times in seconds per job.

	1 Parallel Job	2 Parallel Jobs	1 Parallel, 1 Sequential Job	3 Parallel Jobs	1 Parallel, 2 Sequential Jobs
Test & Set	1.94 ± 0.04 $\sigma = 0.18$	2.76 ± 0.05 $\sigma = 0.23$	3.5 ± 0.2 $\sigma = 0.9$	3.36 ± 0.06 $\sigma = 0.28$	3.7 ± 0.1 $\sigma = 0.5$
Test & Test & Set	1.94 ± 0.04 $\sigma = 0.19$	2.74 ± 0.05 $\sigma = 0.23$	3.7 ± 0.2 $\sigma = 0.9$	3.29 ± 0.06 $\sigma = 0.30$	3.76 ± 0.09 $\sigma = 0.41$
Ticket Lock	1.93 ± 0.04 $\sigma = 0.17$	2.81 ± 0.06 $\sigma = 0.27$	5.1 ± 0.6 $\sigma = 2.5$	3.62 ± 0.07 $\sigma = 0.34$	8.0 ± 0.9 $\sigma = 4.4$
Anderson Lock	2.08 ± 0.05 $\sigma = 0.20$	2.99 ± 0.06 $\sigma = 0.28$	4.9 ± 0.4 $\sigma = 1.7$	3.87 ± 0.08 $\sigma = 0.42$	8.1 ± 0.9 $\sigma = 4.4$
Lock-Free	1.89 ± 0.04 $\sigma = 0.19$	2.62 ± 0.05 $\sigma = 0.22$	3.6 ± 0.2 $\sigma = 0.9$	3.14 ± 0.05 $\sigma = 0.26$	3.54 ± 0.08 $\sigma = 0.37$
FIFO Lock-Free	1.86 ± 0.04 $\sigma = 0.17$	2.57 ± 0.05 $\sigma = 0.20$	3.6 ± 0.3 $\sigma = 1.0$	3.15 ± 0.05 $\sigma = 0.26$	3.57 ± 0.09 $\sigma = 0.44$
Posix Mutex	2.16 ± 0.04 $\sigma = 0.17$	3.01 ± 0.05 $\sigma = 0.22$	4.0 ± 0.2 $\sigma = 0.8$	3.62 ± 0.05 $\sigma = 0.27$	3.99 ± 0.09 $\sigma = 0.40$

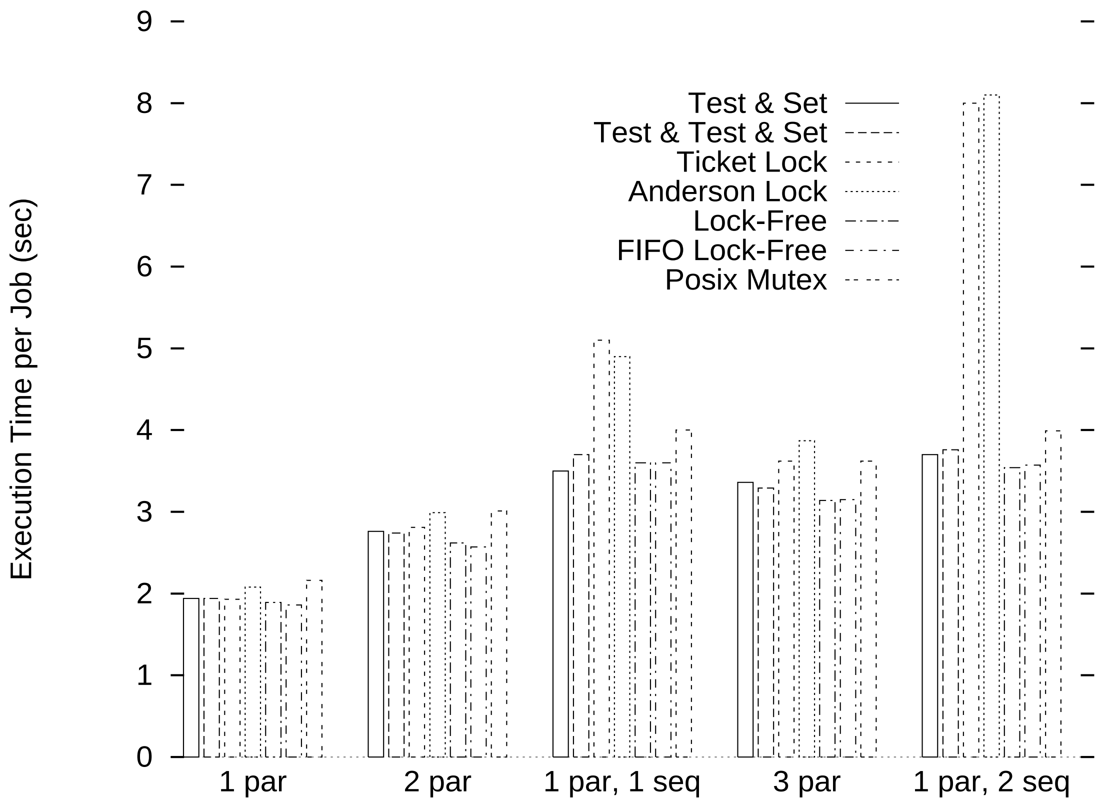

Figure 6.24: CON/comm execution times in seconds per job.

	1 Parallel Job	2 Parallel Jobs	1 Parallel, 1 Sequential Job	3 Parallel Jobs	1 Parallel, 2 Sequential Jobs
Test & Set	0.685 ± 0.008 $\sigma = 0.040$	0.84 ± 0.02 $\sigma = 0.10$	0.95 ± 0.08 $\sigma = 0.35$	0.83 ± 0.03 $\sigma = 0.11$	1.7 ± 0.2 $\sigma = 0.6$
Test & Test & Set	0.687 ± 0.007 $\sigma = 0.034$	0.85 ± 0.03 $\sigma = 0.10$	1.0 ± 0.1 $\sigma = 0.5$	0.86 ± 0.02 $\sigma = 0.11$	1.9 ± 0.2 $\sigma = 0.7$
Ticket Lock	0.685 ± 0.006 $\sigma = 0.030$	0.87 ± 0.03 $\sigma = 0.11$	1.03 ± 0.08 $\sigma = 0.36$	0.85 ± 0.02 $\sigma = 0.10$	1.7 ± 0.2 $\sigma = 0.5$
Anderson Lock	0.681 ± 0.007 $\sigma = 0.030$	0.85 ± 0.03 $\sigma = 0.12$	1.05 ± 0.09 $\sigma = 0.42$	0.85 ± 0.03 $\sigma = 0.11$	1.8 ± 0.2 $\sigma = 0.7$
Lock-Free	0.677 ± 0.008 $\sigma = 0.036$	0.83 ± 0.03 $\sigma = 0.11$	1.01 ± 0.08 $\sigma = 0.39$	0.83 ± 0.02 $\sigma = 0.10$	1.8 ± 0.2 $\sigma = 0.6$
FIFO Lock-Free	0.69 ± 0.01 $\sigma = 0.05$	0.85 ± 0.03 $\sigma = 0.10$	0.98 ± 0.08 $\sigma = 0.37$	0.84 ± 0.02 $\sigma = 0.10$	1.8 ± 0.2 $\sigma = 0.6$
Posix Mutex	0.70 ± 0.02 $\sigma = 0.06$	0.85 ± 0.03 $\sigma = 0.11$	1.01 ± 0.07 $\sigma = 0.33$	0.85 ± 0.02 $\sigma = 0.11$	1.8 ± 0.2 $\sigma = 0.7$

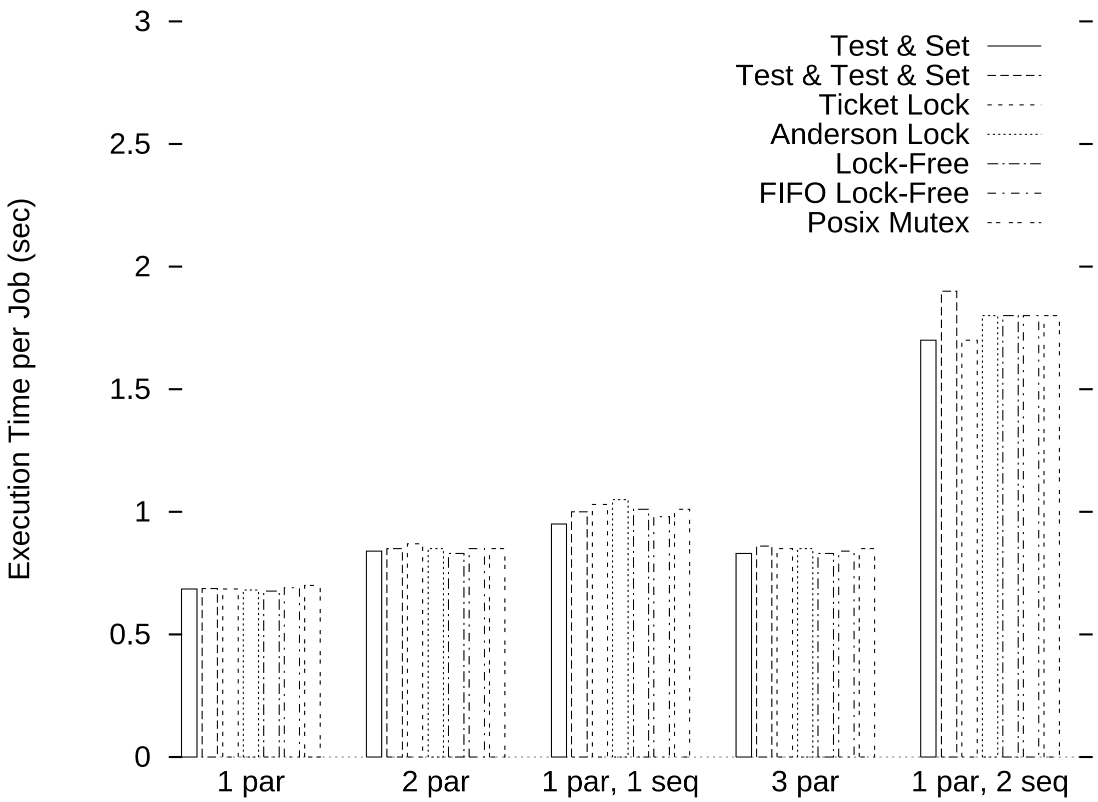

Figure 6.25: 3-D FFT execution times in seconds per job.

	1 Parallel Job	2 Parallel Jobs	1 Parallel, 1 Sequential Job	3 Parallel Jobs	1 Parallel, 2 Sequential Jobs
Test & Set	0.79 ± 0.02 $\sigma = 0.06$	0.84 ± 0.02 $\sigma = 0.08$	1.34 ± 0.06 $\sigma = 0.27$	0.78 ± 0.02 $\sigma = 0.10$	1.22 ± 0.04 $\sigma = 0.16$
Test & Test & Set	0.836 ± 0.009 $\sigma = 0.043$	0.87 ± 0.02 $\sigma = 0.08$	1.34 ± 0.06 $\sigma = 0.26$	0.81 ± 0.02 $\sigma = 0.10$	1.23 ± 0.04 $\sigma = 0.17$
Ticket Lock	0.817 ± 0.008 $\sigma = 0.037$	0.95 ± 0.05 $\sigma = 0.23$	2.0 ± 0.2 $\sigma = 0.8$	1.00 ± 0.09 $\sigma = 0.48$	2.7 ± 0.3 $\sigma = 1.3$
Anderson Lock	0.905 ± 0.009 $\sigma = 0.040$	1.00 ± 0.05 $\sigma = 0.25$	2.0 ± 0.2 $\sigma = 0.7$	1.0 ± 0.1 $\sigma = 0.5$	3.1 ± 0.3 $\sigma = 1.4$
Lock-Free	0.682 ± 0.009 $\sigma = 0.043$	0.76 ± 0.02 $\sigma = 0.09$	1.18 ± 0.05 $\sigma = 0.24$	0.72 ± 0.02 $\sigma = 0.09$	1.10 ± 0.04 $\sigma = 0.16$
FIFO Lock-Free	0.68 ± 0.02 $\sigma = 0.05$	0.76 ± 0.02 $\sigma = 0.09$	1.20 ± 0.05 $\sigma = 0.24$	0.71 ± 0.02 $\sigma = 0.09$	1.10 ± 0.04 $\sigma = 0.18$
Posix Mutex	1.284 ± 0.009 $\sigma = 0.042$	0.92 ± 0.02 $\sigma = 0.08$	1.54 ± 0.05 $\sigma = 0.24$	0.85 ± 0.02 $\sigma = 0.09$	1.34 ± 0.03 $\sigma = 0.15$

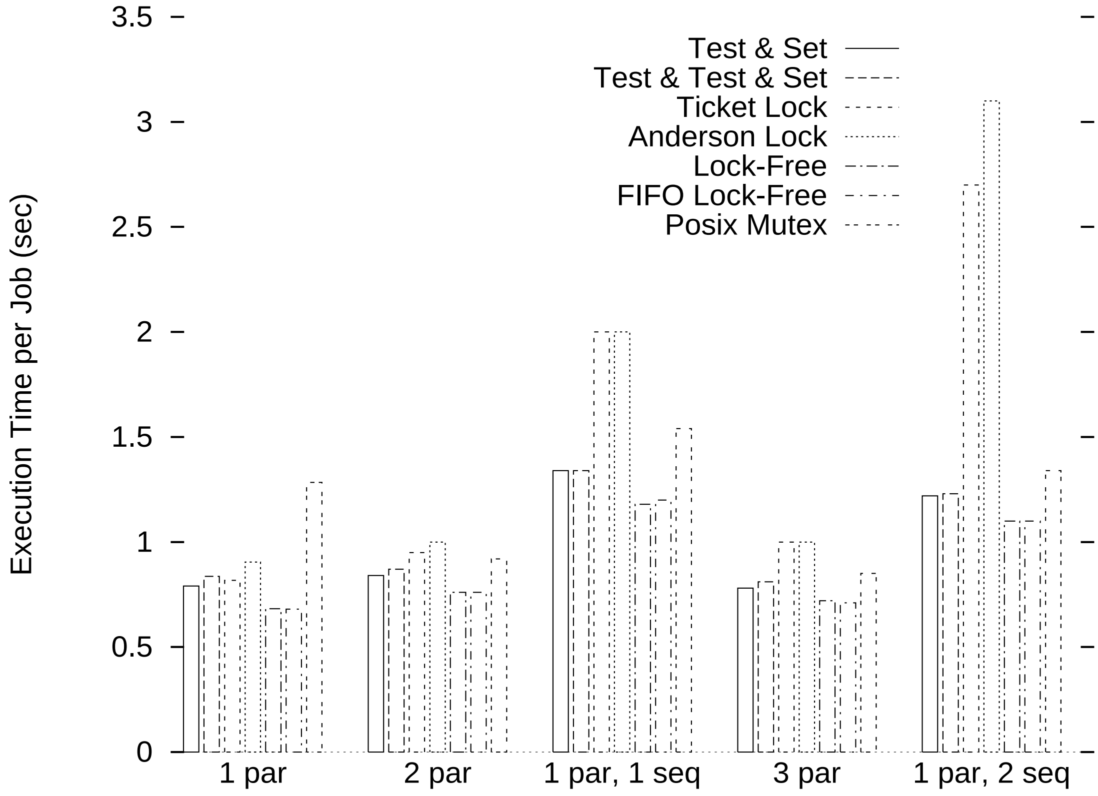

Figure 6.26: EM3D execution times in seconds per job.

gorithms. Per-job execution times with two competing parallel jobs range from 2.74 to 2.99 seconds with the spin lock algorithms, from 5% to 14% larger than the 2.62 seconds required with the lock-free algorithm. With three parallel jobs, spin lock execution times range from 5% to 23% larger than that of the lock-free algorithm. Using the Posix mutes, CON/comm executes in 3.01 seconds per job with two jobs and 3.62 seconds per job with three jobs. Both times are 15% larger than that achieved with the lock-free algorithm.

Competition with groups of sequential processes leads to wide variations in execution time for CON/comm, allowing the simpler spin locks to deliver performance comparable to that of the lock-free algorithm. Performance with both the ticket lock and the Anderson lock was very erratic, and the distributions for two sequential processes were truncated by timeouts in our global execution layer. Execution times are still well above those of the other algorithms, however, requiring roughly 39% more time with one group of sequential jobs and roughly 126% more time with two groups. Using the Posix mutex, execution time is competitive with one group and roughly 13% larger with two groups.

As discussed earlier, a process in a CON/comm job does not eliminate all cached data when it intervenes between two time slices of another process. The result is apparent from the data: three competing parallel jobs require more execution time per job than do two, as more cached data must be reread from memory. In particular, times rise from 39% to 66% larger than those on a dedicated machine. The sequential processes used in these experiments stress the cache by design, however. A CON/comm job competing with sequential processes loses nearly all cached data between time slices, incurring a large but relatively constant performance penalty. Execution time per job is roughly the same when competing with two groups as when competing with only one, about 87% larger than on a dedicated machine.

In contrast, EM3D performance (Figure 6.26) remains stable between two and three parallel jobs and between one and two sequential jobs. Compared with a dedicated machine, execution time per job is about 11% slower when competing with parallel jobs and about 73% slower when competing with sequential jobs. The mean values in fact improve slightly for some algorithms, but are within the confidence intervals for the lock-free algorithm. We attribute these slight improvements to a reduction in the likelihood of triggering the application-level blocking problem described earlier.

EM3D performance across algorithms demonstrates a clear separation between our lock-free algorithm and the rest. Execution time using the spin lock algorithms generally increases with the complexity of the lock. Performance with both the ticket lock and the Anderson lock was very erratic, but the other algorithms delivered reasonably stable (and faster) results. With two parallel jobs, execution times range from 0.84 to 1.00 seconds per job, 11% to 32% larger than the 0.76 seconds required with our algorithm. With three parallel jobs, the times range from 0.78 to 1.00 seconds, 8% to 39% larger than our algorithm's time of 0.72 seconds. In both cases, performance with Posix mutexes falls between that of the simpler and that of the more complex spin locks, at 21% slower with two jobs and 18% slower with three.

When competing with sequential jobs, the ticket lock and the Anderson lock were very slow. Executions with either of the simpler spin locks were 14% slower than with the lock-free algorithm when competing against one group of sequential processes and 11% slower when competing against two groups. In the same comparisons, executions with the Posix mutex were 31% and 22% slower, respectively.

The last application run, SAMPLE (Figure 6.22), demonstrates the most significant differences between the algorithms. We were unable to obtain reliable multiprogramming data for the ticket lock or for the Anderson lock with this application, but a few application runs returned extremely variable times more than an order of magnitude larger than those of any other algorithm. SAMPLE executes roughly 50% slower with either of the simpler spin locks than with our lock-free algorithm when competing with two or three parallel jobs. Similar penalties arise for competition with sequential jobs, with 46% slower execution times against one sequential job and 40% slower times against two.

The Posix mutex demonstrates the benefits of preemption-safe locking in moving from two to three competing parallel jobs. With two jobs, execution time is 66% slower than with our algorithm, but with three jobs, performance with the Posix mutex is competitive. A similar improvement occurs for competition with sequential jobs, from 128% slower against one job to 55% slower against two.

6.3.6 Summary

This section introduced a variety of alternative algorithms for managing the concurrent message queue in the shared memory protocol, then used a suite of benchmarks to compare the performance of spin locks and preemption-safe locks with that of our lock-free algorithm.

The algorithms result in very similar LogGP parameters. Round-trip times for the more complex spin locks, the ticket lock and the Anderson lock, and for our lock-free algorithm, were slightly higher than those for the simpler spin locks, Test & Set and Test & Test & Set. The Posix mutex was slowest due to the necessity of interacting with the operating system in the lock and unlock operations.

The lock-free algorithm demonstrated the best results on the communication stress test, which measures performance at the extreme of high contention. The algorithms' performance separated primarily by the number of cache lines moved between processors. Execution time with the simpler spin locks increased fairly rapidly with the number of writers, whereas the ticket lock and the Anderson lock were less sensitive to contention. The Posix mutex also tolerated contention, but at an absolute time of more than twice that of all other algorithms. A gap reduction effect dominated the effect of contention on the lock-free algorithm, allowing execution time to decrease slowly with an increasing number of writers.

We next measured application performance on a single, dedicated SMP. For applications with very structured communication, the method used for concurrent access is clearly irrelevant. The infrequency of added overheads or penalties makes them insignificant compared with other sources of execution time variation. However, for applications with more irregular communication, and particularly for unstructured, all-to-all communication, the choice of algorithm is quite important.

A concurrent access algorithm must avoid unnecessary overheads to compete on a dedicated machine. Performance on a multiprogrammed machine, however, depends more on the ability of the algorithm to deliver performance in the presence of competing processes. The Posix mutex algorithm is designed for such an environment, but sacrifices performance on a dedicated machine. The spin lock algorithms are designed for minimal execution time, but can suffer performance degradation under the control of an oblivious scheduler. Our lock-free algorithm balances the low overhead necessary to remain competitive with the robustness necessary to handle multiprogramming. It provides competitive or superior performance for applications with a range of communication patterns on both dedicated and multiprogrammed machines.

6.4 Future Performance

Previous sections in this chapter evaluate our communication layer and the lock-free algorithm relative to other process-based approaches. In contrast, this section investigates performance in a more absolute sense by illustrating the relationship between the memory hierarchy parameters and the cost of passing a message through that hierarchy. We begin the section by breaking the send overhead for our shared memory protocol into components based on functionality and necessary coherence transactions. The section next predicts the impact of current architectural trends on shared memory message-passing performance and discusses the importance of these trends for future systems. These observations lead to a few suggestions for improving performance, with which the section closes.

6.4.1 Overhead breakdown

As an aid to understanding the relationship between access times in the hardware memory hierarchy and the cost of sending messages between processors, we next present a detailed breakdown of short message send overhead in the shared memory protocol. To obtain these data, we repeatedly eliminate single pieces of functionality from the communication layer and calculate the difference in overhead with and without each component. As the time spent on a given component may depend to some degree on the presence or absence of other components, our approach requires that we first define a canonical order. After describing that order and commenting on our rationale for selecting it, we present the results of this experiment and discuss their significance.

Canonical order

We break send overhead into seven main components as outlined below; the annotations explain our rationale.

1. Base cost for writing a message. This component represents the bare minimum possible for message-passing: call overhead, retrieving a cache line from memory, depositing the data into the cache line, and incrementing the queue tail to note that a message has been sent.

2. Cache-to-cache transfer. As we expect cache lines used for message-passing to remain resident in the receiving process' cache until requested by another sender, we next measure the additional overhead associated with a cache-to-cache transfer.

3. Overflow check. Ensuring that a packet does not already contain a message can incur an extra coherence transaction, but is necessary for correctness.

4. Shared memory poll. A poll for each message sent helps the communication layer remain responsive to incoming messages.

5. Argument and tag checking. Although perhaps not absolutely necessary for research prototypes, checking for errors at the interface to a communication layer aids in the debugging and testing process for programs built on top of that layer.

6. Concurrency management. The cost of allowing multiple senders to access the message queue concurrently. Placing this component after argument and tag checking provides an estimate of the relative cost of one-to-one and many-to-one queues.

7. Multi-protocol overhead. The cost of supporting network messages on shared memory send overhead. We add this component last to allow a meaningful breakdown of performance within an SMP.

We further separate the first component—the base cost for writing a message—into four subcomponents to expose the relationship between the necessary coherence transaction and the message insertion. The subcomponents include the following:

1. Call overhead. The cost of argument marshaling, a call to a null function, and return.

2. Increment queue tail. The cost to manipulate a control variable in a queue to deposit each message into a distinct packet. In our system, this manipulation requires only incrementing the tail of a queue.

3. L2 cache miss. The cost to write the first word of data into a message. The associated coherence transaction should require very few cycles until the write buffer fills. When the buffer does fill, as with the tight loop used in our benchmark, coherence transaction latency hides the previous two components. Thus we can only measure those components in the absence of the cache miss.

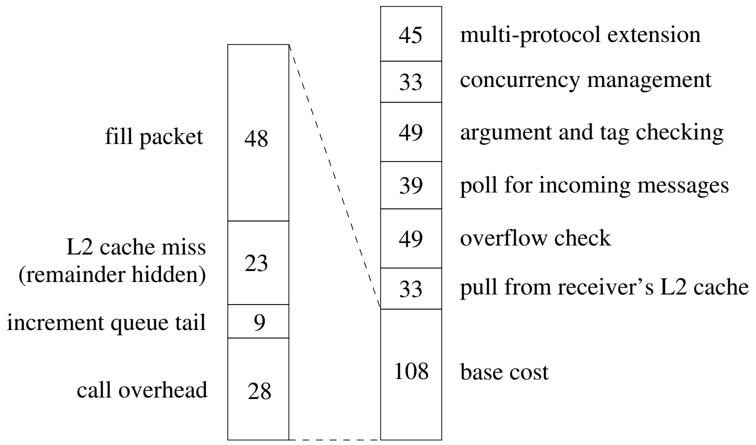

Figure 6.27: Breakdown of send overhead in cycles for the shared memory protocol. The left bar shows the costs of each component for the base case, which performs no error checking or concurrency management for the destination queue. The cost of the latter appears in the right bar. The send overhead totals 311 cycles (1.9 microseconds) for the shared memory protocol and 356 cycles (2.1 microseconds) for the multi-protocol layer.

4. Fill packet. The cost to write the remaining data into the message packet and to signal completion by changing the packet state.

Results

A breakdown of the send overhead for short messages appears in Figure 6.27. The left bar illustrates the base cost of a short message in the absence of error checking and concurrency management for the destination queue. The total of 108 cycles (0.65 microseconds) also assumes that the message packet is not resident in the receiver's cache. To reach the base cost, the sender prepares eight arguments and calls the appropriate function in a total of 28 cycles. Advancing the tail of the queue requires another 9 cycles.

The L2 cache miss incurred by the first write into the packet adds only 23 cycles. In fact, the remainder of this coherence transaction hides most of the previous operations, as is evident from a comparison with similar results from an earlier version of our communication layer. To support multiplexing functionality not necessary in this thesis, our communication layer stores function pointers in a table attached to each endpoint. A previous version of the layer allowed direct calls and required only 11 cycles for call overhead and the same 9 cycles for advancing the queue tail. Including the coherence transaction brought the time to within one cycle of the 60 required by the current layer, however, indicating that the additional overhead of the indirect function call is completely hidden by the cache miss. Filling the remainder of the packet, enforcing a memory barrier on store ordering, and changing the packet state requires the remaining 48 cycles.

The right bar in the figure extends the base cost with measurements of the remaining components. When queue packets are resident in a receiver's L2 cache, each message incurs an additional 33 cycle penalty, the difference between a memory read and a cache-to-cache transfer.

Retaining packets in the receiver's cache also increases the cost of the next component, the check for queue overflow. The check reads the packet state and immediately uses the result, incurring a 49 cycle coherence transaction to bring the data into the cache. The read operation does not invalidate the copy in the receiver's L2 cache, however, and leaves the packet in a shared state. Filling the packet thus still requires an invalidation, the second transaction. In contrast, the Gigaplane coherence protocol places lines read from memory into an exclusive state, voiding the need for invalidation.

The sender can also eliminate the extra transaction by writing an unused part of the packet before the overflow check and using a memory barrier to prevent reordering. This approach reduces send overhead by 40 cycles when measured in isolation, but has a negative impact on other LogP parameters and on application results, presumably due to cache line thrashing when a receiver polls its queue during the send operation.

The local poll operation performed before each send adds another 39 cycles. The poll operation is responsible for the failure of our statistical averaging techniques with send overhead. Measurements of prior components resulted in reasonably independent and normally distributed data, whereas measurements including the poll operation suffer from the non-independence and bimodality discussed in Chapter 5.

Function argument and endpoint tag checking by the communication layer introduce another 49 cycles of overhead. The layer checks that it has been initialized, then validates both the source and destination endpoints for the message. The layer also guarantees that both the sending and receiving processes have mapped each other's shared memory queue blocks. Finally, the layer verifies that the correct endpoint tag has been provided by the sender before allowing message insertion into the destination queue.

Concurrency management adds 33 cycles when both packet assignment and claiming succeed on the first try, bringing the total for the shared memory protocol used in isolation to 311 cycles (1.9 microseconds).

The final component represents the overhead required to support the network protocol when using only the shared memory protocol. For this measurement, we forced the local traffic estimate to remain high to simulate actual use of the protocol. The send overhead benchmark otherwise brings the estimate artificially low, as the benchmark prevents the receiver from issuing reply messages. On average, inclusion of the network protocol more than doubles the time spent in the poll operation, bringing the total for the multi-protocol implementation to 356 cycles (2.1 microseconds).

6.4.2 Architectural trends

The detailed breakdown of send overhead sheds light on the relationship between the message costs and the parameters of the memory hierarchy. Regrouping the overhead breakdown data, we find that that coherence transactions make up only a third of the whole. Another five-twelfths is spent on the basic mechanisms of the operation: call overhead, argument checks, queue advancement, and packet filling. The final quarter of the time is split between managing concurrent access between senders and polling for incoming local messages. Two current trends in computer architecture will significantly shift this balance and impact overall message performance: the diverging performance of processors and memory, and the trading of latency for scalability in servers. We now explain these trends and apply specific examples to our data in order to understand their impact.

The performance of memory systems relative to processors is constantly declining. Rapidly climbing clock speeds and advances in instruction-level parallelism have dwarfed improvements in memory latency. According to Patterson and Hennessy [PH98], clock rates increased at an average of 40% per year after 1985, whereas memory increased at an average of only 9% per year. Interleaved memories and pipelined misses have boosted memory bandwidth to adequate levels in spite of this difference, but latency-critical operations do not benefit from these techniques.

A comparison of the UltraSPARC Model 170 workstation with the more recent Ultra 10 workstation exemplifies this trend. The Model 170 features the first chipset based on the UltraSPARC V9 processor, with a 6 nanosecond (167 MHz) clock. The Ultra 10 uses the UltraSPARC-IIi, with a 3.33 nanosecond (300 MHz) clock (a second configuration features a 3 nanosecond clock). Conservatively assuming that improvements in instruction-level parallelism between the architectures are nullified by dependencies, mispredictions, or other problems, the Ultra 10 should execute code that requires no off-chip accesses in about 56% of the time required by the Model 170. In contrast, the memory system has improved very little between these architectures: memory read latency is 264 nanoseconds on the Model 170 and 240 nanoseconds on the Ultra 10. Data are delivered faster on the new machine—in 91% of the time required by the earlier machine—but the improvement falls far short of the processor speed improvement. In terms of cycles, memory read latency has risen from 44 to 72 cycles. Off-chip cache latency has tracked processor performance more closely, but L2 cache hit time still rises from 8 cycles on the Model 170 to 10 cycles on the Ultra 10.

We apply these numbers to the data by increasing the cycles required for memory and coherence transactions. The first three subcomponents of the base message cost together increase by 28 cycles, as the coherence transaction presumably still hides the costs of call overhead and queue tail manipulation. Assuming the same relative slowdown for cache-to-cache transfers as for memory accesses, the cache-to-cache transfer increases by 21 cycles ($33 \times 28/44$). With a similar assumption, the overflow check increases by 31 cycles. Finally, the concurrency management component increases by 8 cycles, corresponding to two CAS operations, each of which travels to and from the L2 cache. Summing the increases, we arrive at a total of 399 cycles (1.3 microseconds), or 68% of our system's time. Coherence transactions rise from 34% of the total in our system to 46% in the prediction.

Server-class architectures have focused partially on scalability, but have at the same time emphasized the performance of rarely- or non-communicating processes. Approximately a year after introducing the Gigaplane-based Enterprise systems, for example, Sun Microsystems brought out a more scalable platform, the Enterprise 10000. Through the Gigaplane-XB interconnect, the Enterprise 10000 scales to 64 processors, but sacrifices memory latency. Compared with the Gigaplane, the Gigaplane-XB adds roughly 240 nanoseconds to each memory or coherence transaction [CPWG97]. Just as the 16% increase in memory latency between the Model 170 and the Enterprise 5000 has little impact on the performance of sequential applications, the extra cost due to the Gigaplane-XB is unlikely to produce startling changes in that regime. For our communication layer, however, the increase translates to an additional 80 cycles of send overhead, changing the total to 391 cycles (2.3 microseconds). Coherence transactions in this case account for 47% of the overhead.

Coherence transactions have become a significantly more important component of message-passing in SMP's in the roughly one to two years separating our example architectures. As we move forward into the future, we expect such transactions to become the dominant component and to limit the performance of communication between processors.

6.4.3 Performance improvements

The trends just discussed increase the cost of passing messages through cache-coherent shared memory on future architectures. Based on our detailed understanding of the costs, we now suggest a few architectural changes to counteract the trends and to improve performance. Our goal in describing these changes is to provide a small set of straightforward modifications to typical commercial architectures that will improve message-passing performance without affecting the behavior of computations that make no use of these capabilities. We envision three complementary changes: one-sided caching, burst support, and transfer optimization. One-sided caching allows a message-centric approach to memory consistency. Burst support provides the ability to push a cache line of data into memory or another processor's cache in a single transaction. Transfer optimization focuses on improving the latency of cache-to-cache transfers.

The extensions suggested here represent an explicit approach to improving the performance of message-passing over cache-coherent interconnects; an application or library must explicitly identify data as the target of these extensions. Implicit approaches to automatically identifying a similar class of data within the coherence protocol itself have been suggested by several studies [CF93, SBS93]. We defer a discussion of the relationship between our approach and the implicit techniques until Chapter 9. COMA architectures such as the KSR-1 also use similar techniques in both implicit and explicit forms.

The overhead breakdown suggests that a significant fraction of the cost of a message involves scenarios in which a processor stalls while waiting for a coherence transaction to complete. One-sided caching caches message data only at a receiver and allows a sender to write the data without first reading it. Using one-sided caching, the data can remain in a receiver's cache, often eliminating the need to stall when accepting a message. A sender that reads one word of a line using one-sided caching reads only that word, receives a possibly inconsistent version, and does not cache the incoming data. Borrowing terminology from the message-passing literature [KC95], we say that a sender pushes a message across the interconnect rather than waiting for the receiver to pull it across. In coherence terminology, this approach is similar to a write-update protocol that updates only the cache belonging to the message recipient. Unlike write-update, however, one-sided caching ensures that the data are never cached outside of the receiver's local cache.

An implementation of one-sided caching requires three forms of support: minor coherence protocol modifications, minor instruction set extensions, and correct use of the new instructions by an application. None of these aspects are necessary for correct system behavior; in their absence, the system operates as a typical commercial SMP. An application that does not use the new instructions correctly may lose its own data, but cannot corrupt other processes in the system. Before providing the details of these extensions, however, we use our system to illustrate how one-sided caching coupled with burst support reduces message-passing overhead. For simplicity, assume that all processes remain resident on unique processors; we address the possibility of migration in describing the implementation strategy.

Consider a process sending a short message. After packet assignment, a sender reads the packet's state to test for queue overflow, at which point the coherence protocol brings the entire packet into the sender's cache. Regardless of the result of the test, however, the additional data serve no useful purpose: the sender writes over the data in a free packet and ignores the data in a claimed or ready packet. With one-sided caching, the sender bypasses the cache in the queue overflow check, recovering wasted data bus bandwidth and possibly eliminating coherence transactions for contended access to the packet. Avoiding the cache does have a drawback, however. When a sender detects an empty packet, the CAS primitive in the claim operation must occur with exclusive access to the data. Normally, the sender's cache places the cache line into an exclusive state, requiring that the cache line be present. The alternative—supporting remote CAS primitives in the processors, caches, and memory banks—seems overly complex. However, we can circumvent the need for an atomic claim operation by using the FIFO form of our algorithm.

A sender need not incur a second coherence transaction with one-sided caching. The state read confirms the sender's claim and implicitly confers exclusive ownership of the line despite the view of the machine's cache controllers. The sender can then fill the packet, using burst support to write the full cache line of data rather than pushing data out of the processor one word at a time. Integrating burst support with the processor can reduce packet filling overhead and contention for write buffer resources on the processor, and contention for the cache line within the memory system. As we discuss below, burst support also simplifies the implementation of one-sided caching. The data burst bypasses the sender's cache and is deposited either into memory or directly into the receiver's cache when it already contains the cache line. The hardware must guarantee either atomicity or an ordering for the burst. A guaranteed ordering is sufficient for a communication layer to ensure that a message packet only appears ready for delivery after all data have been written.

Next consider a process receiving a message. Our system incurs two coherence transactions after the sender fills the packet. The first brings a packet into the receiver's cache in a shared state to verify that the message is ready for delivery. The second invalidates the sender's copy in order to free the packet. One-sided caching reduces the number of transactions to zero or one. In the worst case, a receiver must load the cache line from memory. If the receiver does so before the message is ready, the cache line remains in the cache until kicked out by accesses to computation data. More frequently, a message packet is already cache-resident, and the message has been pushed into the receiver's cache by the sender. The receiver then incurs no coherence transactions.

A more abstract view of message-passing as a means of synchronizing concurrent processes through memory also validates the advantages provided by one-sided caching and burst support. In the general case, consider a process, the receiver, waiting for a signal flag from a second process, the sender. The sender sets the flag only once, but the receiver may check the flag many times, performing other useful work between checks. Allowing the flag to migrate out of the receiver's cache potentially penalizes performance, but the sender must be able to write the flag. When the sender does write the flag, however, the flag's previous value, although known to be clear in this scenario, becomes irrelevant. With one-sided caching and burst support, the data remain cached close to the receiver, yet the sender can deliver not only the flag but a small amount of data atomically into the cache.

One-sided caching and burst support require three new instructions: specialized loads for senders and receivers and a burst store instruction. We use the UltraSPARC-I architecture [SME97], the processor used in our experimental platform, as a reference in discussing the implementation details. The interconnect specification for the UltraSPARC-I processor is the Ultra Port Architecture, or UPA.

The first instruction, `load-receiver`, performs a standard load operation when the target cache line resides in either the local cache or in memory. As with the protocol used by the UPA, data cache lines read from memory enter the cache in an exclusive state.

When a line resides in another cache, however, `load-receiver` causes the remote cache to invalidate the line after providing the data to the local cache and places the line in an exclusive state in the local cache. Many architectures already support this kind of transaction on the interconnect; in the UPA, for example, the Sparc issues a read-to-own transaction (P_RDO_REQ). A processor typically issues a read-to-own transaction in response to a store or synchronization primitive cache miss. The `load-receiver` instruction requires that a processor issue a read-to-own transaction for a load miss as well. In the absence of cache or interconnect support, the memory system can safely treat `load-receiver` misses as normal load misses.

A receiver process is not in practice uniquely linked to a processor. Instead, the operating system can migrate a receiver process from processor to processor in the system. When the scheduler removes a receiver process, message data remain in the processor's cache. If the scheduler later reschedules the receiver on the same processor, the data remain cached only by the receiver. However, if the scheduler places the receiver process on a different processor, it temporarily violates the semantics of one-sided caching. The next `load-receiver` instruction issued by the receiver process restores the proper state of affairs, however, moving the data to the new processor and invalidating the old cache line. Data updated in the wrong processor's cache eventually migrate to the new processor or return to memory.

The second instruction, `load-sender`, reads a single word (32- or 64-bit) without changing any coherence state. If the target cache line is in the local cache, `load-sender` operates as a normal load. If the target cache line is in memory, `load-sender` obtains the data but does not allocate the cache line. If the target cache line is in another cache, `load-sender` obtains the data without affecting the state of the line in the remote cache. The data returned from a `load-sender` may be inconsistent; the application—or in our case the communication layer—is responsible for ensuring that the inconsistency does not lead to incorrect behavior. The UPA provides two signals with behavior close to that desired. A read-to-discard transaction (P_RDD_REQ) reads an entire cache line from a remote cache without affecting the state. A processor can issue this signal in response to a `load-sender` miss and discard the remainder of the cache line, but only at the expense of wasted data interconnect bandwidth. A noncached read transaction (P_NCRD_REQ) bypasses the cache to obtain up to 16 B of data, but cannot legally be delivered to another processor in the current specification. `load-sender` must also check the local cache before issuing the transaction, as a sender might be scheduled immediately after the receiver on the same processor. In the absence of cache or interconnect support, the memory system can safely treat `load-sender` misses as normal load misses.

The last instruction, `store-burst`, bypasses the cache and emits a full cache line of data across the memory interconnect. The Sparc V9 Visual Instruction Set extension already supports generation of store bursts to memory. However, the write invalidate UPA transaction (P_WRI_REQ) issued in response to a store burst on the Sparc invalidates any remote caches and delivers the data to memory. In contrast, `store-burst` must update the receiver's cache, requiring that the cache snoop to identify `store-burst` transactions to exclusively owned cache lines and replace the cached data with the burst coming across the interconnect. After receiving a burst, a receiver changes a previously unmodified cache line to a modified state. As cache controllers must already handle full lines of incoming data in regular operation, accepting `store-burst` transactions should not be difficult.

As an alternative to burst support in the processor, the cache controller can be modified to issue a burst when the processor writes to a specific location within a marked cache line, a notion similar to the automatic signaling on write described in [CK96]. With this approach, however, the sender's cache must hold the line in an exclusive state while writing the message. Allowing a receiver to simultaneously maintain a stale copy of the line to speed polling requires more significant changes to the coherence protocol. The lines must also be identified as message data in some fashion, whereas burst support allows implicit identification by an application.

The ability of system architects to optionally support one-sided caching is an important aspect of our design, allowing not only a smooth integration into future systems but also, if necessary, a smooth removal. Although current trends indicate that the explicit control of data transfer embodied by one-sided caching will become more important over time, changes in computer architecture have never been easy to predict, and future architects may decide to eliminate one-sided caching. In such an event, third-party software built to use one-sided caching will continue to operate on machines that no longer support it, degrading gracefully to the performance of an invalidation-based protocol. The same cannot be said of burst support, as software simulation cannot inexpensively achieve multi-word atomic transactions. However, a number of system architects have already chosen to include such support in their architectures.

One-sided caching and burst support aim primarily at hiding coherence transaction latency rather than reducing it. Transfer optimization, the last improvement, addresses reduction directly, but is the least constructive and most open-ended of the three. Communication data tend to move between processors and their caches rather than down into memory. Optimizing the transfer of data between processor caches thus reduces overhead and application-to-application latency for most messages. The fact that cache-to-cache transfers are slower than memory accesses on the Enterprise 5000 came as something of a surprise to us, as a transfer involves no DRAM access latency. The difference most likely reflects an emphasis on memory system performance at the expense of the path between caches rather than an intrinsic limitation of the hardware. As logic performance continues to grow relative to memory performance, transfer latency improvement relative to memory latency should come naturally on future architectures. We urge the architects of these systems to take a step further, however, and to actively reduce the time required to move data across the memory hierarchy above the main memory.

6.5 Comparison with NOW's

A significant body of literature documents the qualitative advantages of cache-coherent shared memory systems over their less illustrious cousins, networks of workstations or NOW's. We assumed in this thesis a model of multiple communicating processes rather than a thread-based model, and in doing so have perhaps sacrificed some of the performance potential of the SMP. In return, we have gained simplicity and generality, allowing a direct comparison using the same application code on very different platforms. Within the process model, the results presented in this chapter establish the effectiveness of our solution compared with other possibilities and illustrate the relationship between the overhead of the communication layer and the latencies in the memory system. From this perspective, the difference between performance on a single SMP and performance on a NOW places an upper bound on the advantages of using a hybrid system such as a Clump. In this section, we report application performance data on a network of UltraSPARC Model 170 workstations, a system that differs from our Clump only in the memory hierarchy and in the network topology; the main processors, the network fabric, and the application code are all exactly the same. The next chapter reports performance on the Clump and explains how the structure of an application affects that performance.

The execution times for five application runs on an 8-processor NOW appear in

	SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
SMP	0.762 ± 0.004 $\sigma = 0.017$	1.11 ± 0.02 $\sigma = 0.05$	1.89 ± 0.04 $\sigma = 0.19$	0.677 ± 0.008 $\sigma = 0.036$	0.682 ± 0.009 $\sigma = 0.043$
8-way NOW	1.50 ± 0.02 $\sigma = 0.06$	1.095 ± 0.003 $\sigma = 0.014$	3.755 ± 0.006 $\sigma = 0.029$	0.770 ± 0.003 $\sigma = 0.013$	4.01 ± 0.02 $\sigma = 0.10$

Table 6.6: Comparison between SMP and NOW performance. Each platform has eight processors.

Table 6.6 alongside the times on one 8-processor SMP. The polling strategy brings NOW execution times with the multi-protocol layer lower than those with only the network protocol, and we chose to present the faster runs in the table. The converse is true for the SMP, hence the SMP executions used only the shared memory protocol. For the computation-bound CON/comp run, the SMP has no advantage over the NOW; the increased memory latency cancels the benefit of faster communication. The computation and structured communication in 3-D FFT attenuate the advantage of higher communication bandwidth, and the execution time on the SMP is 0.677 seconds, or 88% of the 0.770 seconds required on the NOW. The remaining applications perform substantially better on the SMP. SAMPLE executes in 0.762 seconds, or 51% of the 1.50 second NOW execution time. CON/comm executes in 1.89 seconds, or 50% of the 3.755 second NOW execution time. EM3D executes in 0.682 seconds, or 17% of the 4.01 second NOW execution time.

The potential benefit of the faster shared memory protocol depends to a large extent on the amount and style of communication between the processors. Applications that send many messages in a pipelined fashion reap a large benefit. Applications that employ more structured communication add dependencies between processes. Although the benchmarked round-trip time of the shared memory protocol is lower than that of the network protocol (as reported in the next chapter), the dominant portion of message response latency in an application executing on a dedicated machine arises from the response time of the application itself. Latency-dependent applications thus gain less from the faster communication protocol.

6.6 Summary

This chapter reported the performance of our shared memory protocol: tuning the protocol parameters, comparing the lock-free algorithm with alternatives, exploring the reasons behind the demonstrated performance, and comparing that performance with that achieved by a NOW.

We showed that a backoff yield time of 255 microseconds provides good performance on a range of applications, whether the machine be dedicated to a single job or time-shared between multiple jobs. Using the same criteria, we selected a packet queue length of 4,096 entries and a bulk data queue length of 16 entries.

We argued that our lock-free algorithm enjoys a performance advantage over alternative algorithms, and backed these arguments up with empirical data on both spin locks and the preemption-safe Posix mutex. These data established our algorithm's superior performance across a range of contention levels and for applications on both dedicated and multiprogrammed machines

We presented a detailed breakdown of send overhead with our shared memory protocol and noted that coherence transactions account for only a third of the time. Given the current trends in architecture, we expect this fraction to increase in the future, requiring more cycles to communicate a single message between processors. We suggested a variety of means by which this additional time can be reduced or hidden, leaving the implementation of these approaches for future work.

Finally, we compared the performance of our system running on a single SMP with its performance running on a similar network of workstations. For some applications, communication was not an important factor in execution time, and the two results were nearly equal. For applications more dependent on communication latency and bandwidth, the SMP executed the codes in between 17% and 88% of the time required by the NOW. These improvements bound the performance potential of the Clump, as applications running on such a system must use a combination of the slower and faster protocols. In the next chapter, we continue to report the performance of our layer with details of performance on the Clump.

Chapter 7

Multi-Protocol Performance

The previous chapter focused on the performance of the shared memory protocol used as a single-protocol communication layer. This chapter reports the performance of the full multi-protocol layer, exploring the impact of supporting multiple protocols and investigating the potential performance advantage of a Clump architecture over a network of workstations. The chapter begins by determining appropriate parameter values for the adaptive polling strategy. Performance results follow, beginning with a study of the impact of multi-protocol support on both microbenchmark and application performance, and followed by a comparison between the Clump and a NOW. The chapter closes with a discussion of the relationship between the architecture, the programming model, and performance.

7.1 Polling Strategy Selection

A multi-protocol communication layer that naively polls each protocol for incoming messages on every message insertion can significantly reduce the performance of the individual protocols. Disparate protocol costs are the source of the problem; in our layer, the high cost of polling the network significantly increases the overhead of inserting and accepting messages with the shared memory protocol. Network performance is also be affected by inclusion of the shared memory protocol, but the effects are relatively minor. We addressed this problem in Chapter 4 by developing a framework for an adaptive polling strategy. This approach uses estimates of traffic from previous poll operations to make protocol polling decisions, minimizing the impact of supporting multiple protocols and allowing a single binary to obtain good performance on a range of platforms.

Shared Memory	Network
0.306 ± 0.002 μsec	2.604 ± 0.002 μsec
51 cycles	434 cycles

Table 7.1: Cost of polling an empty queue on an Enterprise 5000. The uncacheable NIC memory makes remote messages queues roughly an order of magnitude slower than shared memory queues.

This section reports our selection of appropriate parameter values for the adaptive polling strategy. Four parameters arise in the model: the damping parameter d determines the rate at which the traffic estimates adjust to changes in message arrival rates; the equality parameter k gives the number of network polls to skip when message traffic on the two protocols is roughly equal; and the minimum and maximum skip parameters, s_{min} and s_{max} respectively, bound the number of poll operations between network polls. Our system uses a damping parameter of 256, an equality parameter of 4, a minimum skip of 4, and a maximum skip of 64. We study the impact of varying each parameter individually on the performance of applications, holding the other parameters at their nominal values. After highlighting the differences in the cost of the application-level poll operation with our two protocols, we present data from an SMP and a NOW. As these platforms require only a single protocol, they allow us to determine acceptable parameter values for environments in which only a single protocol is used. The data for the full Clump follow.

7.1.1 Polling differences

The cost of an application-level poll operation for each single protocol appears in Table 7.1. We obtained these values by timing a loop that polls an empty endpoint one million times, then repeating the experiment 100 times and reporting the mean and 95% confidence interval for each measurement. The 51 cycle cost of the shared memory poll consists primarily of call overhead and data structure manipulation; checking the queue generally requires only a read from the L1 cache, as we expect this data to remain hot when a process communicates. A message arrival, of course, requires that the receiver pull the message packet across the interconnect. The network poll performs essentially the same operations as does the shared memory poll, but reads packet states across the SBUS I/O bus. The longer access times increase the poll time to 434 cycles, even when the message queues are empty.

The cost reported here for the shared memory poll differs from that reported in the overhead breakdown of the previous chapter for two reasons. First, the application-level poll uses an indirect function call, loading a function pointer from a table attached to the endpoint. In contrast, the message send operation invokes the poll operation directly. Second, the numbers reported here include a couple of cycles of loop overhead.

7.1.2 SMP data

An application using a multi-protocol communication layer on an SMP makes use of only the shared memory protocol. With an adaptive polling strategy, the traffic estimate for remote traffic remains at its minimum value, while the estimate for local traffic varies with communication frequency. The parameters of primary importance in this regime are the maximum skip s_{max} and the equality parameter k. The maximum skip bounds the ability of the communication layer to ignore the existence of the network protocol by requiring that an application check for remote messages. The equality parameter is less important, but potentially governs behavior after a process idles at a barrier, reducing its local traffic estimate.

The data on one SMP appear in Figures 7.1 and 7.2. The upper section of each figure reports application execution times in seconds for five application runs on one Enterprise 5000. The lower section illustrates the same data graphically, with the order in the legend selected to match the order of application runs in the graph. The vertical scales are the same for the two SMP-based graphs, but are a factor of four smaller than those used for the NOW- and Clump-based graphs presented later in this section.

The maximum skip parameter s_{max} typically defines the network polling rate for an application running on an SMP. As the value of s_{max} increases, the amortized cost of such polling decreases, allowing performance to asymptotically approach the performance achieved with a single-protocol communication layer. Figure 7.1 presents data on the effect of the maximum skip parameter on application performance. We varied s_{max} from 1 to 256 to capture a wide range around our nominal value of 64. The character of the data obeys our predictions: for each application run, execution time decreases towards an asymptotic limit as s_{max} increases. CON/comp and 3-D FFT are relatively insensitive to this parameter; execution times at a maximum skip of 1 are 5% and 2% longer than their values for large s_{max}. For the other applications, the parameter is more important:

s_{max}	SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
1	1.702 ± 0.006 $\sigma = 0.028$	1.13 ± 0.02 $\sigma = 0.05$	2.27 ± 0.02 $\sigma = 0.06$	0.678 ± 0.004 $\sigma = 0.020$	1.028 ± 0.008 $\sigma = 0.035$
2	1.243 ± 0.005 $\sigma = 0.025$	1.10 ± 0.02 $\sigma = 0.07$	2.01 ± 0.02 $\sigma = 0.05$	0.682 ± 0.008 $\sigma = 0.036$	0.842 ± 0.007 $\sigma = 0.032$
4	1.002 ± 0.006 $\sigma = 0.028$	1.076 ± 0.006 $\sigma = 0.026$	1.89 ± 0.06 $\sigma = 0.28$	0.671 ± 0.006 $\sigma = 0.028$	0.756 ± 0.007 $\sigma = 0.031$
8	0.876 ± 0.005 $\sigma = 0.024$	1.083 ± 0.007 $\sigma = 0.031$	1.82 ± 0.01 $\sigma = 0.05$	0.667 ± 0.006 $\sigma = 0.026$	0.701 ± 0.005 $\sigma = 0.025$
16	0.820 ± 0.006 $\sigma = 0.028$	1.065 ± 0.006 $\sigma = 0.026$	1.767 ± 0.009 $\sigma = 0.042$	0.670 ± 0.007 $\sigma = 0.032$	0.680 ± 0.006 $\sigma = 0.028$
32	0.787 ± 0.005 $\sigma = 0.021$	1.088 ± 0.006 $\sigma = 0.026$	1.751 ± 0.009 $\sigma = 0.045$	0.662 ± 0.004 $\sigma = 0.020$	0.669 ± 0.006 $\sigma = 0.025$
64	0.773 ± 0.005 $\sigma = 0.023$	1.073 ± 0.007 $\sigma = 0.030$	1.74 ± 0.01 $\sigma = 0.05$	0.666 ± 0.006 $\sigma = 0.027$	0.667 ± 0.007 $\sigma = 0.030$
128	0.765 ± 0.005 $\sigma = 0.022$	1.062 ± 0.006 $\sigma = 0.029$	1.74 ± 0.01 $\sigma = 0.05$	0.666 ± 0.006 $\sigma = 0.027$	0.664 ± 0.006 $\sigma = 0.029$
256	0.759 ± 0.005 $\sigma = 0.021$	1.079 ± 0.007 $\sigma = 0.035$	1.726 ± 0.008 $\sigma = 0.039$	0.666 ± 0.007 $\sigma = 0.031$	0.663 ± 0.007 $\sigma = 0.031$

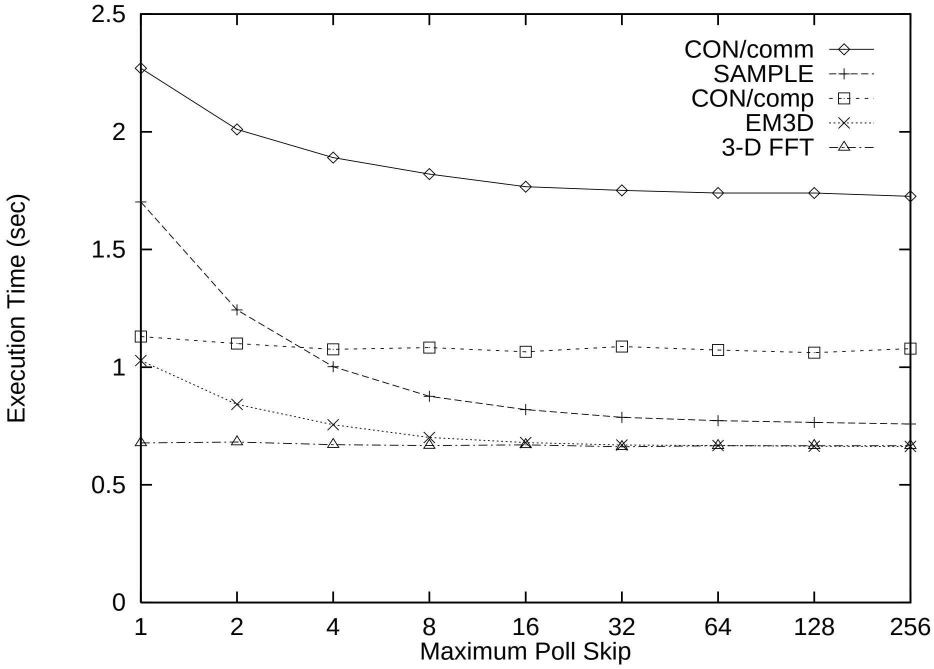

Figure 7.1: Maximum skip selection on an SMP.

k	SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
1	0.773 ± 0.006 $\sigma = 0.026$	1.081 ± 0.008 $\sigma = 0.040$	1.736 ± 0.008 $\sigma = 0.039$	0.670 ± 0.007 $\sigma = 0.032$	0.665 ± 0.007 $\sigma = 0.031$
2	0.774 ± 0.006 $\sigma = 0.026$	1.061 ± 0.005 $\sigma = 0.021$	1.731 ± 0.008 $\sigma = 0.037$	0.663 ± 0.005 $\sigma = 0.020$	0.667 ± 0.007 $\sigma = 0.030$
4	0.773 ± 0.005 $\sigma = 0.023$	1.073 ± 0.007 $\sigma = 0.030$	1.74 ± 0.01 $\sigma = 0.05$	0.666 ± 0.006 $\sigma = 0.027$	0.667 ± 0.007 $\sigma = 0.030$
8	0.772 ± 0.006 $\sigma = 0.026$	1.090 ± 0.008 $\sigma = 0.037$	1.731 ± 0.009 $\sigma = 0.042$	0.668 ± 0.006 $\sigma = 0.029$	0.668 ± 0.005 $\sigma = 0.022$
16	0.775 ± 0.006 $\sigma = 0.029$	1.066 ± 0.007 $\sigma = 0.032$	1.733 ± 0.009 $\sigma = 0.041$	0.666 ± 0.006 $\sigma = 0.026$	0.669 ± 0.007 $\sigma = 0.033$
32	0.778 ± 0.007 $\sigma = 0.034$	1.085 ± 0.007 $\sigma = 0.031$	1.734 ± 0.009 $\sigma = 0.044$	0.669 ± 0.007 $\sigma = 0.035$	0.665 ± 0.006 $\sigma = 0.028$
64	0.774 ± 0.006 $\sigma = 0.025$	1.058 ± 0.005 $\sigma = 0.021$	1.733 ± 0.008 $\sigma = 0.040$	0.667 ± 0.006 $\sigma = 0.029$	0.671 ± 0.007 $\sigma = 0.033$

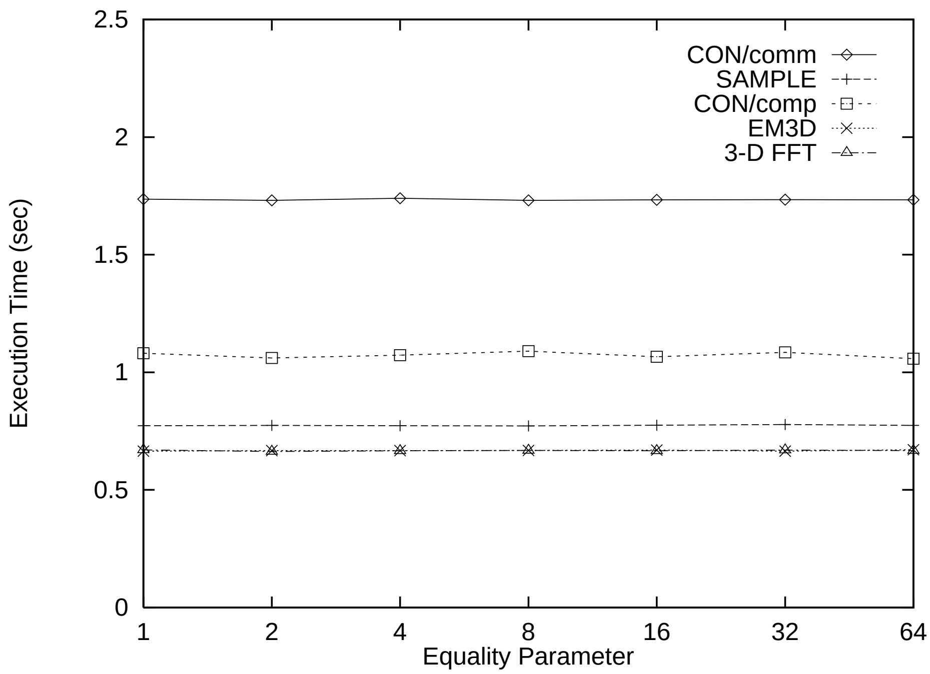

Figure 7.2: Equality parameter selection on an SMP.

CON/comp executes 32% more slowly at a maximum skip of 1 than at large s_{max}, EM3D executes 55% more slowly, and SAMPLE executes 124% more slowly. A value of 16 obtains most of the potential benefit, achieving execution times within 10% of the limits for all of the application runs. Smaller values result in more significant penalties.

The equality parameter k plays a more minor role on an SMP. Consider an imperfectly load-balanced phase. Idle processors wait at a barrier, repeatedly lowering their local traffic estimates and increasing the frequency with which they poll for remote messages. After such a synchronization operation terminates, normal communication begins to push the estimate back into a regime in which s_{max} determines the polling frequency. The equality parameter k affects the network polling rate during the synchronization operation and the subsequent recovery period. Figure 7.2 reports the results of varying k from 1 to 64 on one SMP. The data are essentially flat, although CON/comp performance does change by a few percent across the entire range studied. The phenomenon described does not play a major role in these applications, and does not place any restrictions on our selection of the parameter value.

7.1.3 NOW data

An application using a multi-protocol communication layer on a NOW makes use of only the network protocol. The local traffic estimate in the adaptive polling strategy remains at its minimum value, while the remote estimate tracks the arrival rate of messages from the network. The minimum skip rate s_{min} plays almost the same role on the NOW as the maximum skip rate plays on the SMP, bounding the impact of the shared memory protocol on the remote message performance. The other parameters have less significance for a NOW.

The data on an 8-processor NOW appear in Figure 7.3. The figure has the same form as those used to present the SMP-based data, with tabular numerical data in the upper section and a graphical equivalent in the lower section. The vertical scale in Figure 7.3 is a factor of four larger than in the previous figures, however, as execution times on a NOW are typically larger than those on an SMP.

The minimum skip parameter s_{min} typically defines the network polling rate for an application running on a NOW. As the value of s_{min} decreases, the average cost of intervening shared memory polls for a network poll decreases. However, for small values of s_{min},

s_{min}	SAMPLE	CON/comp	CON/comm	3-D FFT	EM3D
1	2.05 ± 0.02 $\sigma = 0.08$	1.090 ± 0.004 $\sigma = 0.019$	3.686 ± 0.007 $\sigma = 0.031$	0.775 ± 0.003 $\sigma = 0.013$	3.98 ± 0.02 $\sigma = 0.08$
2	1.69 ± 0.02 $\sigma = 0.06$	1.100 ± 0.004 $\sigma = 0.020$	3.698 ± 0.007 $\sigma = 0.033$	0.773 ± 0.003 $\sigma = 0.013$	3.99 ± 0.02 $\sigma = 0.10$
4	1.50 ± 0.02 $\sigma = 0.06$	1.095 ± 0.003 $\sigma = 0.014$	3.755 ± 0.006 $\sigma = 0.029$	0.770 ± 0.003 $\sigma = 0.013$	4.01 ± 0.02 $\sigma = 0.10$
8	1.410 ± 0.009 $\sigma = 0.041$	1.112 ± 0.004 $\sigma = 0.019$	4.0 ± 0.1 $\sigma = 0.5$	0.769 ± 0.003 $\sigma = 0.014$	4.03 ± 0.05 $\sigma = 0.22$
16	1.400 ± 0.006 $\sigma = 0.027$	1.141 ± 0.004 $\sigma = 0.016$	4.149 ± 0.006 $\sigma = 0.027$	0.765 ± 0.003 $\sigma = 0.012$	4.00 ± 0.02 $\sigma = 0.10$
32	1.381 ± 0.005 $\sigma = 0.021$	1.204 ± 0.005 $\sigma = 0.022$	4.738 ± 0.007 $\sigma = 0.035$	0.766 ± 0.003 $\sigma = 0.013$	3.99 ± 0.02 $\sigma = 0.08$
64	1.373 ± 0.007 $\sigma = 0.033$	1.294 ± 0.004 $\sigma = 0.019$	5.65 ± 0.01 $\sigma = 0.05$	0.767 ± 0.003 $\sigma = 0.013$	4.02 ± 0.03 $\sigma = 0.11$
128	1.369 ± 0.008 $\sigma = 0.037$	1.545 ± 0.004 $\sigma = 0.018$	8.1 ± 0.2 $\sigma = 0.6$	0.771 ± 0.003 $\sigma = 0.014$	4.10 ± 0.03 $\sigma = 0.11$

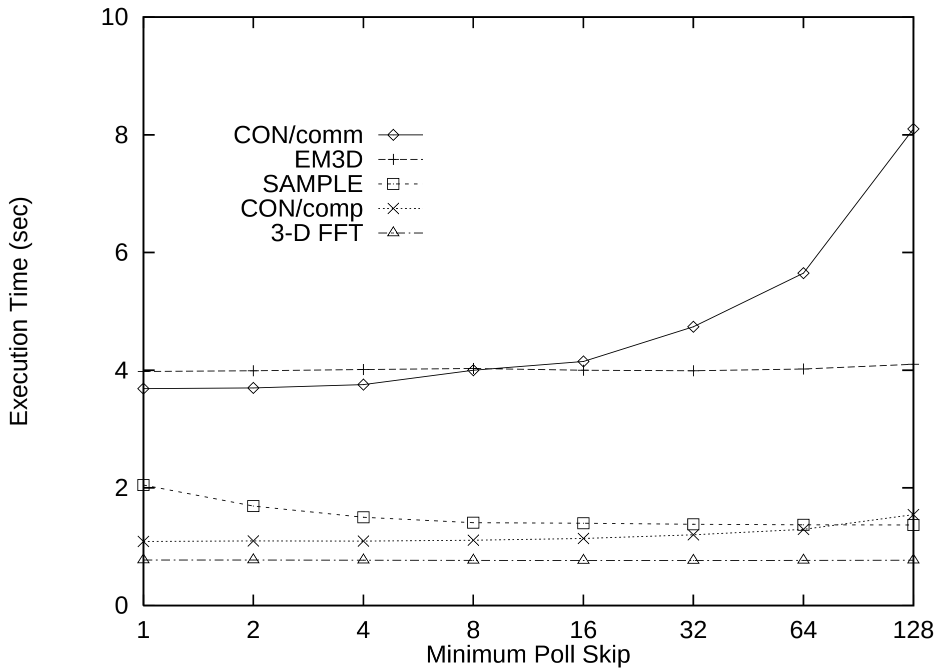

Figure 7.3: Minimum skip selection on an 8-way NOW.

overly frequent polling of empty network queues can increase execution time. Figure 7.3 presents data on the effect of the minimum skip parameter on application performance. We varied s_{min} from 1 to 128 to capture a wide range around our nominal value of 4. The performance character of the application runs varies in this measurement. 3-D FFT and EM3D are essentially flat, demonstrating performance independent of the minimum skip. Due to some latency-critical sections in the implementation, both runs of the connected components algorithm obtain the best performance at a minimum skip of 1. CON/comp execution time increases by 42% across the range shown in the graph, and CON/comm execution time increases by 120% across the same range. SAMPLE sort hides latency more effectively, and performance improves as s_{min} increases due to amortization of the network polling cost in the message send overhead. For SAMPLE, execution time at a minimum skip of 1 is 50% longer than at a minimum skip of 128. A value of 4 or 8 balances between the two types of applications, achieving execution times within 10% of the limits for all of the runs shown. Smaller values penalize applications that issue many messages before requiring a reply, and larger values penalize applications that rely on immediate replies.

7.1.4 Clump data

Applications executing on a Clump depend on all polling strategy parameters, as each plays a role in defining how the strategy adapts to the patterns of communication within the application. The impact of the parameters on performance tends to be less dramatic than with the platforms that require only a single protocol, however. As the application runs do not explicitly recognize the hierarchy within the Clump, they switch protocols frequently, and the network polling frequency generally remains in the middle of the range rather than at either extreme.

The data on the Enterprise 5000 Clump appear in Tables 7.2 through 7.5 and Figures 7.4 through 7.7. The tables present execution times for six application runs using our standard reporting style, and the figures present the same data graphically. The vertical scale in the figures is the same as that used for the NOW-based data. The horizontal ranges for the individual parameters are subsets of those used for the SMP- and NOW-based data; we chose to study narrower ranges in light of the previous data.

We varied the minimum skip parameter s_{min} from 1 to 128. The numerical results appear in Table 7.2, and a graphical form appears in Figure 7.4. The application behavior is

Minimum Skip s_{min}	SAMPLE	CON/comp	CON/comm
1	3.30 ± 0.07 $\sigma = 0.32$	1.408 ± 0.006 $\sigma = 0.026$	7.26 ± 0.08 $\sigma = 0.37$
2	3.25 ± 0.03 $\sigma = 0.12$	1.415 ± 0.007 $\sigma = 0.033$	7.20 ± 0.03 $\sigma = 0.10$
4	3.33 ± 0.07 $\sigma = 0.31$	1.416 ± 0.007 $\sigma = 0.033$	7.19 ± 0.03 $\sigma = 0.14$
8	3.29 ± 0.04 $\sigma = 0.17$	1.413 ± 0.006 $\sigma = 0.028$	7.21 ± 0.03 $\sigma = 0.12$
16	3.29 ± 0.04 $\sigma = 0.19$	1.44 ± 0.02 $\sigma = 0.06$	7.49 ± 0.03 $\sigma = 0.14$
32	3.23 ± 0.03 $\sigma = 0.13$	1.50 ± 0.04 $\sigma = 0.17$	7.91 ± 0.02 $\sigma = 0.09$
64	3.13 ± 0.03 $\sigma = 0.13$	1.51 ± 0.02 $\sigma = 0.07$	8.48 ± 0.02 $\sigma = 0.09$
128	3.08 ± 0.04 $\sigma = 0.16$	1.64 ± 0.03 $\sigma = 0.13$	9.92 ± 0.03 $\sigma = 0.10$

Minimum Skip s_{min}	3-D FFT	EM3D/naive	EM3D/good
1	0.497 ± 0.008 $\sigma = 0.035$	3.71 ± 0.01 $\sigma = 0.05$	3.514 ± 0.009 $\sigma = 0.044$
2	0.486 ± 0.006 $\sigma = 0.026$	3.677 ± 0.008 $\sigma = 0.035$	3.429 ± 0.008 $\sigma = 0.038$
4	0.481 ± 0.006 $\sigma = 0.025$	3.654 ± 0.008 $\sigma = 0.039$	3.402 ± 0.007 $\sigma = 0.033$
8	0.48 ± 0.02 $\sigma = 0.06$	3.66 ± 0.02 $\sigma = 0.09$	3.390 ± 0.008 $\sigma = 0.035$
16	0.473 ± 0.006 $\sigma = 0.027$	3.632 ± 0.008 $\sigma = 0.040$	3.370 ± 0.007 $\sigma = 0.035$
32	0.471 ± 0.004 $\sigma = 0.019$	3.602 ± 0.008 $\sigma = 0.038$	3.330 ± 0.008 $\sigma = 0.039$
64	0.475 ± 0.006 $\sigma = 0.027$	3.57 ± 0.01 $\sigma = 0.05$	3.335 ± 0.007 $\sigma = 0.033$
128	0.478 ± 0.008 $\sigma = 0.036$	3.60 ± 0.01 $\sigma = 0.05$	3.47 ± 0.02 $\sigma = 0.05$

Table 7.2: Minimum skip selection on a Clump.

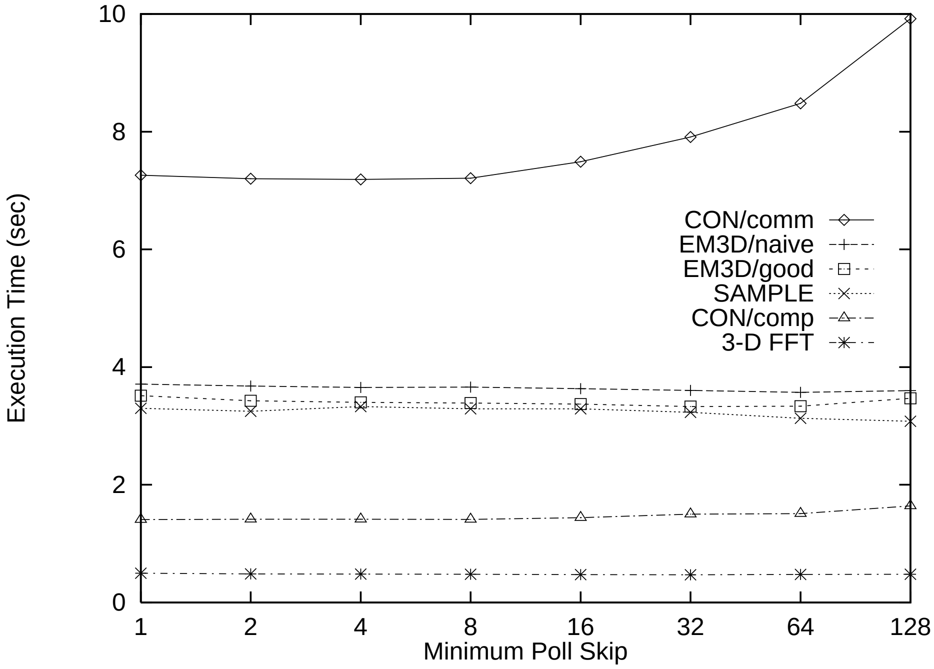

Figure 7.4: Minimum skip selection on a Clump.

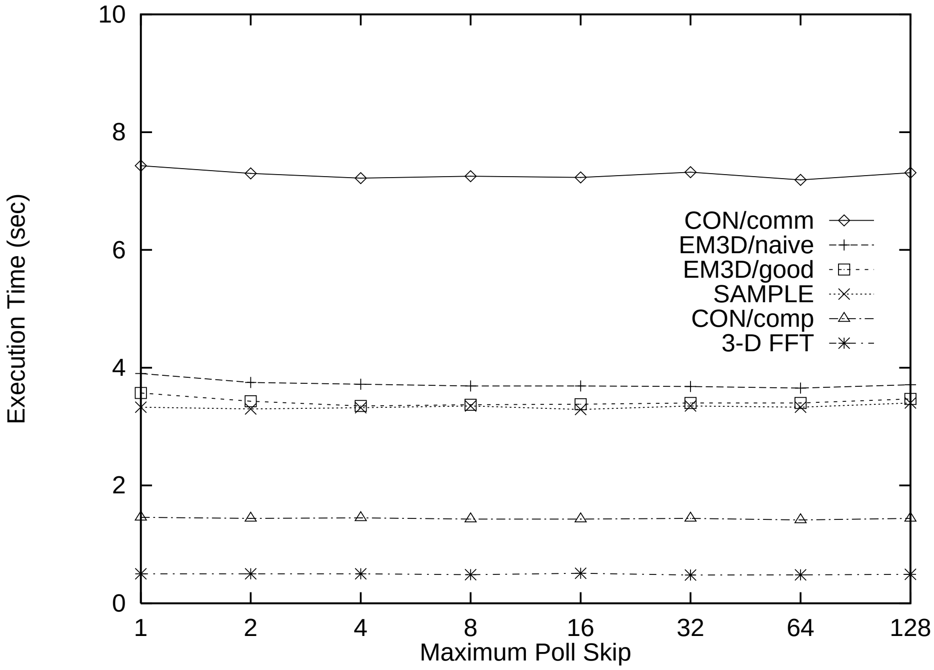

Figure 7.5: Maximum skip selection on a Clump.

Maximum Skip s_{max}	SAMPLE	CON/comp	CON/comm
1	3.33 ± 0.04 $\sigma = 0.17$	1.46 ± 0.02 $\sigma = 0.08$	7.43 ± 0.04 $\sigma = 0.16$
2	3.30 ± 0.03 $\sigma = 0.13$	1.44 ± 0.02 $\sigma = 0.06$	7.30 ± 0.04 $\sigma = 0.16$
4	3.32 ± 0.05 $\sigma = 0.22$	1.45 ± 0.02 $\sigma = 0.09$	7.22 ± 0.04 $\sigma = 0.17$
8	3.35 ± 0.04 $\sigma = 0.15$	1.43 ± 0.01 $\sigma = 0.05$	7.25 ± 0.04 $\sigma = 0.15$
16	3.29 ± 0.03 $\sigma = 0.11$	1.43 ± 0.02 $\sigma = 0.08$	7.23 ± 0.04 $\sigma = 0.17$
32	3.35 ± 0.05 $\sigma = 0.21$	1.44 ± 0.02 $\sigma = 0.07$	7.32 ± 0.04 $\sigma = 0.18$
64	3.33 ± 0.07 $\sigma = 0.31$	1.416 ± 0.007 $\sigma = 0.033$	7.19 ± 0.03 $\sigma = 0.14$
128	3.4 ± 0.1 $\sigma = 0.5$	1.44 ± 0.02 $\sigma = 0.07$	7.31 ± 0.04 $\sigma = 0.20$

Maximum Skip s_{max}	3-D FFT	EM3D/naive	EM3D/good
1	0.50 ± 0.02 $\sigma = 0.05$	3.90 ± 0.02 $\sigma = 0.08$	3.57 ± 0.02 $\sigma = 0.08$
2	0.50 ± 0.02 $\sigma = 0.07$	3.75 ± 0.02 $\sigma = 0.08$	3.43 ± 0.02 $\sigma = 0.09$
4	0.50 ± 0.02 $\sigma = 0.06$	3.72 ± 0.02 $\sigma = 0.08$	3.35 ± 0.02 $\sigma = 0.06$
8	0.485 ± 0.009 $\sigma = 0.042$	3.69 ± 0.02 $\sigma = 0.07$	3.37 ± 0.03 $\sigma = 0.13$
16	0.51 ± 0.02 $\sigma = 0.09$	3.69 ± 0.02 $\sigma = 0.08$	3.38 ± 0.02 $\sigma = 0.07$
32	0.48 ± 0.02 $\sigma = 0.05$	3.68 ± 0.02 $\sigma = 0.06$	3.40 ± 0.02 $\sigma = 0.07$
64	0.481 ± 0.006 $\sigma = 0.025$	3.654 ± 0.008 $\sigma = 0.039$	3.402 ± 0.007 $\sigma = 0.033$
128	0.49 ± 0.02 $\sigma = 0.05$	3.71 ± 0.02 $\sigma = 0.10$	3.47 ± 0.03 $\sigma = 0.11$

Table 7.3: Maximum skip selection on a Clump.

Equality Parameter k	SAMPLE	CON/comp	CON/comm
1	3.38 ± 0.04 $\sigma = 0.20$	1.44 ± 0.02 $\sigma = 0.09$	7.35 ± 0.09 $\sigma = 0.44$
2	3.34 ± 0.03 $\sigma = 0.14$	1.44 ± 0.02 $\sigma = 0.07$	7.25 ± 0.04 $\sigma = 0.16$
4	3.33 ± 0.07 $\sigma = 0.31$	1.416 ± 0.007 $\sigma = 0.033$	7.19 ± 0.03 $\sigma = 0.14$
8	3.30 ± 0.05 $\sigma = 0.20$	1.43 ± 0.02 $\sigma = 0.06$	7.33 ± 0.04 $\sigma = 0.15$
16	3.32 ± 0.04 $\sigma = 0.17$	1.43 ± 0.02 $\sigma = 0.06$	7.45 ± 0.06 $\sigma = 0.25$
32	3.32 ± 0.04 $\sigma = 0.17$	1.44 ± 0.02 $\sigma = 0.08$	7.50 ± 0.04 $\sigma = 0.18$

Equality Parameter k	3-D FFT	EM3D/naive	EM3D/good
1	0.49 ± 0.02 $\sigma = 0.06$	3.71 ± 0.03 $\sigma = 0.13$	3.39 ± 0.02 $\sigma = 0.08$
2	0.50 ± 0.04 $\sigma = 0.20$	3.68 ± 0.02 $\sigma = 0.07$	3.37 ± 0.02 $\sigma = 0.07$
4	0.481 ± 0.006 $\sigma = 0.025$	3.654 ± 0.008 $\sigma = 0.039$	3.402 ± 0.007 $\sigma = 0.033$
8	0.485 ± 0.008 $\sigma = 0.035$	3.70 ± 0.02 $\sigma = 0.07$	3.49 ± 0.02 $\sigma = 0.08$
16	0.49 ± 0.02 $\sigma = 0.05$	3.70 ± 0.02 $\sigma = 0.07$	3.52 ± 0.03 $\sigma = 0.10$
32	0.49 ± 0.02 $\sigma = 0.06$	3.71 ± 0.02 $\sigma = 0.10$	3.53 ± 0.03 $\sigma = 0.11$

Table 7.4: Equality parameter selection on a Clump.

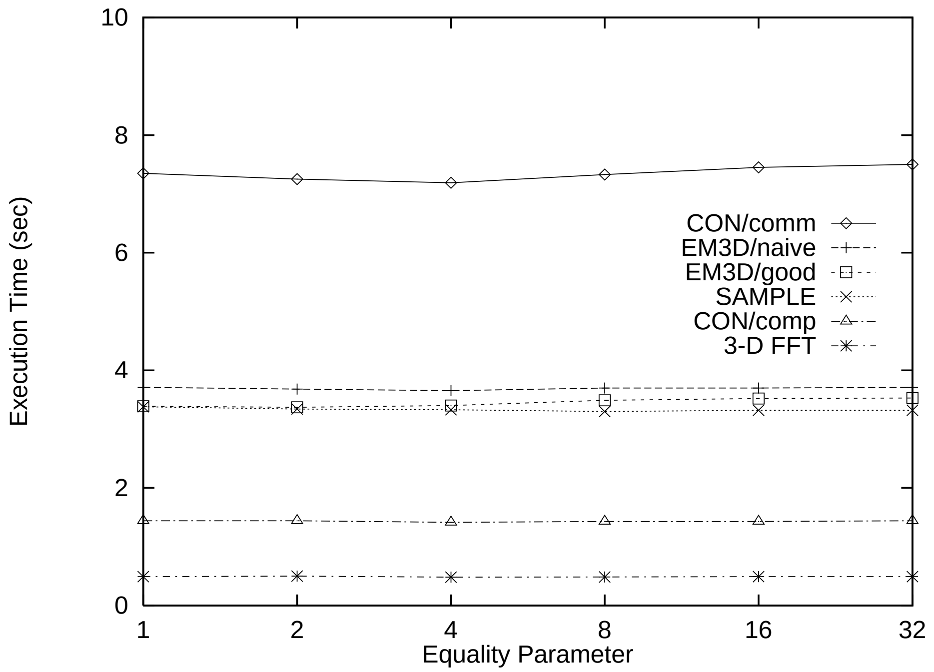

Figure 7.6: Equality parameter selection on a Clump.

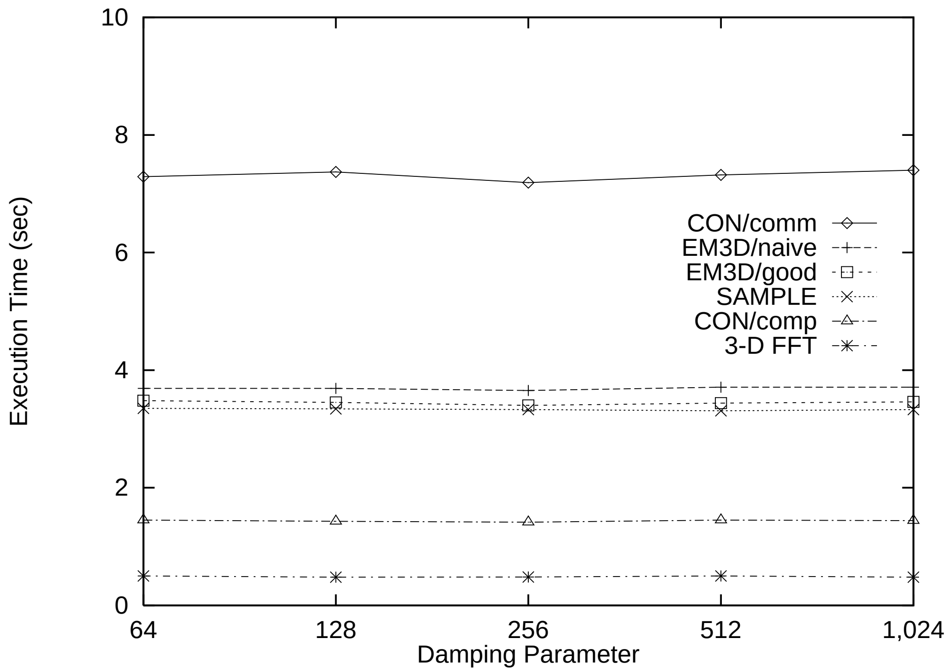

Figure 7.7: Damping parameter selection on a Clump.

Damping Parameter d	SAMPLE	CON/comp	CON/comm
64	3.35 ± 0.04 $\sigma = 0.17$	1.45 ± 0.03 $\sigma = 0.10$	7.29 ± 0.04 $\sigma = 0.16$
128	3.34 ± 0.04 $\sigma = 0.16$	1.43 ± 0.02 $\sigma = 0.06$	7.37 ± 0.09 $\sigma = 0.45$
256	3.33 ± 0.07 $\sigma = 0.31$	1.416 ± 0.007 $\sigma = 0.033$	7.19 ± 0.03 $\sigma = 0.14$
512	3.31 ± 0.03 $\sigma = 0.11$	1.45 ± 0.02 $\sigma = 0.10$	7.32 ± 0.03 $\sigma = 0.14$
1024	3.33 ± 0.04 $\sigma = 0.16$	1.44 ± 0.02 $\sigma = 0.07$	7.40 ± 0.08 $\sigma = 0.39$

Damping Parameter d	3-D FFT	EM3D/naive	EM3D/good
64	0.50 ± 0.02 $\sigma = 0.06$	3.69 ± 0.02 $\sigma = 0.07$	3.48 ± 0.03 $\sigma = 0.12$
128	0.480 ± 0.007 $\sigma = 0.032$	3.69 ± 0.02 $\sigma = 0.07$	3.45 ± 0.02 $\sigma = 0.09$
256	0.481 ± 0.006 $\sigma = 0.025$	3.654 ± 0.008 $\sigma = 0.039$	3.402 ± 0.007 $\sigma = 0.033$
512	0.50 ± 0.02 $\sigma = 0.06$	3.71 ± 0.02 $\sigma = 0.08$	3.44 ± 0.02 $\sigma = 0.08$
1024	0.48 ± 0.02 $\sigma = 0.05$	3.71 ± 0.02 $\sigma = 0.08$	3.46 ± 0.02 $\sigma = 0.08$

Table 7.5: Damping parameter selection on a Clump.

dominated by the effect on remote messages, and the character of the individual lines is the same as that observed on the NOW. The significance of the underlying effects is muted by the use of local communication, however. 3-D FFT and the EM3D runs, EM3D/naive and EM3D/good, are again essentially flat, executing only a few percent slower at a minimum skip of 1 than at other values. CON/comp execution time increases by 16% across the range measured, and CON/comm increases by 37%. SAMPLE execution time again decreases with increasing s_{min}, requiring 7% more time at a minimum skip of 1 than at a minimum skip of 128. The satisfactory values for s_{min} on the NOW—4 and 8—obtain equivalent performance on the Clump, and we select a minimum skip of 4 as it allows a more flexible polling strategy.

The importance of the maximum skip parameter s_{max} on the Clump extends beyond its impact on local message performance. It also determines the response time to remote messages after periods of network inactivity. If the maximum skip parameter is too large, the polling strategy cannot respond quickly to a change in the balance of traffic. We varied s_{max} from 1 to 128, a slightly narrower range than that measured on an SMP. Table 7.3 presents the numerical results, and Figure 7.5 presents the same results graphically. Although 3-D FFT and SAMPLE execution times were flat across this range, execution times for the other applications are a few percent longer at either end of the range than in the middle. We select a maximum skip of 64 to minimize SMP execution times without penalizing Clump execution times. This value results in minimal execution times for all applications except EM3D/good on the Clump; EM3D/good executes in 2% more time than is possible with a maximum skip of 4, but such a choice is clearly unacceptable for applications executing on SMP's.

The equality parameter k determines the frequency of network polling when observed traffic levels are roughly equal. We varied k from 1 to 32, a slightly narrower range than that explored on an SMP. Table 7.4 reports the results numerically, Figure 7.6 illustrates them graphically. SAMPLE, CON/comp, and 3-D FFT are not significantly affected by the value of k, and their execution times are flat across the range measured. For the remaining application runs, values of 2 and 4 result in optimal performance. Smaller values result in overly frequent network polling, and larger values result in overly infrequent polling, in either case increasing execution time by a few percent. We choose an equality parameter of 4.

The damping parameter d, which determines the degree of smoothing in the traffic estimates, has roughly the same impact as the equality parameter on the Clump. We varied d from 64 to 1,024 to cover a wide range around our nominal value of 256. Table 7.5 presents numerical results, and Figure 7.7 presents the same results graphically. For SAMPLE and 3-D FFT, d plays no significant role, and execution times remain flat across the range measured. For the other applications, execution times at either end of the range are a few percent longer than those at a damping parameter of 256. We thus select 256 as our damping parameter, completing the tuning of the adaptive polling strategy.

7.2 Multi-Protocol Overhead

The multi-protocol layer allows an application to take advantage of fast communication within each SMP without explicitly recognizing the hierarchy in the architecture, but incurs unnecessary on platforms that require only a single protocol for communication. In this section, we quantify the effects of this additional overhead at two levels, first in terms of their LogGP parameters, then in terms of application performance. The results highlight the success of the adaptive polling strategy in limiting the impact of multi-protocol support on performance.

7.2.1 Communication parameters

LogGP parameters for the two protocols used in isolation and in a multi-protocol layer appear in Table 7.6. The column at the left replicates the results reported in Chapter 6 for the shared memory protocol in isolation. The right column reports the parameters of the network protocol in isolation, between two Enterprise 5000's. The two middle columns present the parameters for each protocol in the multi-protocol layer. As with the measurement of the average network polling cost, the local traffic estimate is held high when measuring the parameters of the shared memory protocol in the multi-protocol layer. Forcing the estimate high prevents the benchmark from artificially reducing it by delaying the receiver.

Compared with the network protocol, round-trip times for the shared memory protocol are nearly nine times faster, requiring 5.588 microseconds for a local message and 48.2 microseconds for a remote message. The gap across the cache-coherent bus, 3.005 microseconds, is roughly five times faster than the 14.802 microseconds for the network. The

Parameter	Shared Memory	Multi-Protocol Shared Memory	Multi-Protocol Myrinet	Myrinet
Latency (L)	-0.50 ± 0.01	-0.52 ± 0.02	12.56 ± 0.02	8.16 ± 0.08
Send Overhead (o_s)	1.819 ± 0.005	2.135 ± 0.005	3.463 ± 0.003	5.424 ± 0.003
Receive Overhead (o_r)	1.47 ± 0.02	1.73 ± 0.02	8.93 ± 0.01	10.51 ± 0.01
Gap (g)	3.005 ± 0.002	3.251 ± 0.002	14.51 ± 0.01	14.802 ± 0.007
Gap per Byte (G)	6.035 ± 0.006 nanoseconds/B	6.30 ± 0.02 nanoseconds/B	31.208 ± 0.004 nanoseconds/B	30.962 ± 0.002 nanoseconds/B
Bandwidth ($1/G$)	158.0 ± 0.2 MB/sec	151.3 ± 0.3 MB/sec	30.558 ± 0.003 MB/sec	30.801 ± 0.002 MB/sec
Round Trip Time (RTT)	5.588 ± 0.004	6.675 ± 0.003	49.912 ± 0.007	48.2 ± 0.2

Table 7.6: Effect of multi-protocol overhead on communication parameters. The table presents LogGP parameters and round trip times in microseconds for single- and multi-protocol layers running either within one SMP or between two SMP's.

158.0 MB/sec of peak realized bandwidth achieved by the shared memory protocol is more than five times the 30.801 MB/sec achieved over the network.

The adaptive polling strategy prevents the multi-protocol support from significantly affecting either protocol. Local message round-trip time increases by 19%, from 5.588 to 6.675 microseconds. Send overhead increases by 17%, from 1.819 to 2.135 microseconds. Receive overhead also goes up, increasing by 18% from 1.47 to 1.73 microseconds. Gap rises by 8%, from 3.005 to 3.251 microseconds. The overhead of network polling is further amortized for bulk transfers, allowing the gap per byte to increase by only 4%, from 6.035 to 6.30 nanoseconds/B. For the network protocol, the adaptive polling strategy can actually improve performance. Although round-trip time and gap per byte rise slightly, by 4% and 1% respectively, reducing network polling frequency also reduces overhead and gap. Send overhead in the multi-protocol layer is 64% of overhead in the single-protocol layer, or 3.463 microseconds compared to 5.424. Receive overhead is 85% of that required in the single-protocol layer, or 8.93 microseconds compared to 10.51. Finally, gap is 98% of the single-protocol value, 14.51 microseconds compared to 14.802.

The numbers presented in this section help to illuminate the performance of message-passing across cache-coherent buses and the interactions between the two protocols. Until the most recent generation of machines and interconnection technology, network

	8-way SMP			8-way NOW
	Shared Memory		Multi-Protocol	Multi-Protocol
	No Yield	Sleep Yield		
SAMPLE	0.697 ± 0.007 $\sigma = 0.032$	0.762 ± 0.004 $\sigma = 0.017$	0.773 ± 0.005 $\sigma = 0.023$	1.50 ± 0.02 $\sigma = 0.06$
CON/comp	1.086 ± 0.006 $\sigma = 0.026$	1.11 ± 0.02 $\sigma = 0.05$	1.073 ± 0.007 $\sigma = 0.030$	1.095 ± 0.003 $\sigma = 0.014$
CON/comm	1.660 ± 0.006 $\sigma = 0.025$	1.89 ± 0.04 $\sigma = 0.19$	1.74 ± 0.01 $\sigma = 0.05$	3.755 ± 0.006 $\sigma = 0.029$
3-D FFT	0.676 ± 0.005 $\sigma = 0.023$	0.677 ± 0.008 $\sigma = 0.037$	0.666 ± 0.006 $\sigma = 0.027$	0.770 ± 0.003 $\sigma = 0.013$
EM3D	0.643 ± 0.005 $\sigma = 0.021$	0.682 ± 0.009 $\sigma = 0.043$	0.667 ± 0.007 $\sigma = 0.030$	4.01 ± 0.02 $\sigma = 0.10$

Table 7.7: Effect of multi-protocol overhead on application performance. The table reports execution times in seconds for five application runs on one 8-processor SMP. Separate column present single-protocol timings with and without processor yielding, and multi-protocol timings. Times for an 8-way NOW are included for comparison.

communication often provided greater bandwidth than that available from the memory system. With our system, the converse is true: the end-to-end latency of local messages is nearly an order of magnitude smaller than that of remote messages. The adaptive polling strategy limits the impact of network polling on local message performance and reduces the overheads associated with remote messages as well.

7.2.2 Application suite

The adaptive polling strategy is clearly successful in limiting multi-protocol overhead in point-to-point communication, but a full evaluation requires that we also examine its affect at the application level. Table 7.7 reports application execution times on one 8-processor SMP and on an 8-way NOW. The SMP-based data break into three columns: the leftmost column reports execution times using a layer that supports only a shared memory protocol and does not yield processors; the second column reports execution times for the same arrangement with processor-yielding and event-based wakeup; the third column reports execution time on a multi-protocol communication layer, which does not yield processors in our implementation. The right column in the table provides execution time data for a NOW using a multi-protocol communication layer. Software bugs prevented the collection of accurate data for a NOW with a layer that supports only the network protocol, but previous data [LMC97] indicate that the difference is inconsequential.

The use of a communication layer with multi-protocol support increases the application execution times by at most 11%. CON/comp and 3-D FFT times are not affected. EM3D increases by 4%, from 0.643 seconds with the single-protocol layer to 0.667 seconds with the multi-protocol layer. CON/comm increase by 5%, from 1.660 to 1.74 seconds. SAMPLE demonstrates the largest increase, rising 11% from 0.697 to 0.773 seconds. The table also shows the application-level overhead of processor yielding, which improves multiprogrammed performance on an SMP at the cost of dedicated performance. Interestingly, supporting the network protocol incurs overhead comparable and in some cases less than that incurred by processor yielding.

7.3 Comparison with NOW's

The high-performance memory interconnect within each SMP promises Clumps a significant performance advantage with respect to a NOW. As demonstrated by the results of the last chapter, most applications can easily reap the benefits offered by fast communication. The full potential of Clumps may be difficult to achieve, however, as a variety of hardware and software interactions tend to degrade overall performance. We begin this section with a brief discussion of these interactions, then illustrate them with application execution times on our cluster of four 8-processor Enterprise 5000's. In the next section, we amplify our discussion of the interactions.

The phenomena that complicate the process of obtaining optimal performance on a Clump take three forms: contention, load imbalance, and interconnect latency. Processes within an SMP compete for resources at many levels—the memory interconnect, the memory banks, and the network interface cards, for example—and the resulting contention creates additional overhead that detracts from performance. Contention is particularly important when capacity is inadequate to meet the demands of the processors. Load imbalance refers both to the distribution of work between processors and to the distribution of communication data between network interface cards. An imbalance in either case results in some resources remaining idle while others complete their larger workloads. Interconnect latency refers to the relatively long latency across an SMP interconnect compared with a workstation interconnect; SMP interconnects generally trade performance for scalability to

	Enterprise 5000 Clump	32-way UltraSPARC 170 NOW
SAMPLE	3.33 ± 0.07 $\sigma = 0.31$	2.46 ± 0.02 $\sigma = 0.07$
CON/comp	1.416 ± 0.007 $\sigma = 0.033$	1.22 ± 0.06 $\sigma = 0.25$
CON/comm	7.19 ± 0.03 $\sigma = 0.14$	6.72 ± 0.08 $\sigma = 0.38$
3-D FFT	0.481 ± 0.006 $\sigma = 0.025$	0.31 ± 0.02 $\sigma = 0.07$
EM3D/naive	3.654 ± 0.008 $\sigma = 0.039$	4.93 ± 0.03 $\sigma = 0.10$
EM3D/good	3.402 ± 0.007 $\sigma = 0.033$	5.39 ± 0.03 $\sigma = 0.14$

Table 7.8: Application execution times in seconds on a Clump and a NOW. Contention for NIC access limits performance on the Clump, particularly for applications dominated by remote communication.

avoid unacceptable increases in price. As many applications exhibit fairly high levels of locality in their data access patterns, interconnect latency is the least important of the three phenomena.

Execution times for six application runs on our Clump and on a 32-processor NOW appear in Table 7.8; both platforms use the multi-protocol communication layer in these measurements. With the exception of EM3D, the NOW outperforms the Clump. The difference is largest for the applications that use all-to-all communication: the Clump's 0.481 second 3-D FFT execution time is 55% longer than the 0.31 seconds required on the NOW, and the Clump's 3.33 second SAMPLE execution time is 35% longer than the 2.46 seconds required on the NOW. The connected component runs are also faster on the NOW. The Clump requires 7% longer to execute CON/comm, a total of 7.19 seconds compared with 6.72 seconds on the NOW. For CON/comp, the difference is greater, the Clump's time of 1.416 seconds is 16% greater than the NOW's time of 1.22 seconds. The Clump does execute the EM3D runs in less time than does the NOW: EM3D/naive runs in 3.654 seconds on the Clump, or 74% of the 4.93 second NOW execution time; EM3D/good runs in 3.402 seconds on the Clump, or 63% of the 5.39 second NOW execution time.

The data in Table 7.8 contain several surprises, of which the relative performance between the Clump and the NOW is the most notable. The primary reason for the longer execution times on the Clump is contention for the NIC's. The Split-C implementation statically assigns one NIC to each process in a parallel job. For the Clump, this approach implies that each of the NIC's is shared between two processes, whereas each NIC in a NOW supports only a single process. Although contention at the interconnect and memory level may also play minor roles, the Gigaplane interconnect and the memory system on the Enterprise 5000 are more than adequate to support concurrent execution on 8 processors.

The second surprise is that CON/comm has a smaller relative increase than does CON/comp between the NOW and the Clump. Although communication contention has a greater impact on CON/comm than on CON/comp, its importance is outweighed by another factor. The connected components algorithm walks the full graph in multiple phases. The final phase, for example, marks each graph node with the unique component identifier of its representative in the reduced graph. Examining each node in turn incurs a large number of cache misses, and a combination of increased memory latency, interconnect contention, and memory contention make this phase of the algorithm take substantially longer on the Clump than on the NOW. The absolute increase is minor compared with the effect of communication contention in CON/comm, but is significant in the less communication-intensive CON/comp.

The final surprise is the relationship between EM3D/naive and EM3D/good execution times on the NOW. Although the virtual processor layout should not be significant on a machine with a flat communication structure, the results exhibit a clear difference. We attribute the change to the specific fat-tree-like topology used to connect the NOW. Most communication data sent in the EM3D/naive pattern traverse subclusters in the network. In contrast, the EM3D/good communication pattern sends more data into the upper levels of the network, resulting in increased contention and degrading performance.

A comparison of execution times using only 4 processors in each SMP with a 16-processor NOW illustrates relative performance in the absence of communication contention. Each NIC in the Clump handles remote communication traffic for only a single process. The results for four of the six runs appear in Table 7.9. With equivalent network resources, the availability of fast communication typically brings execution times below those on the NOW. On the Clump, SAMPLE and CON/comm execute in 88% of their respective execution times on the NOW, and 3-D FFT executes in 97% of its execution time on the NOW. Each of the communication phases in 3-D FFT contains some remote communication, preventing the shared memory protocol from significantly benefiting performance. Slower

	Enterprise 5000 Clump	16-way UltraSPARC 170 NOW
SAMPLE	1.76 ± 0.02 $\sigma = 0.08$	2.01 ± 0.02 $\sigma = 0.06$
CON/comp	1.128 ± 0.002 $\sigma = 0.009$	1.110 ± 0.004 $\sigma = 0.015$
CON/comm	4.18 ± 0.02 $\sigma = 0.07$	4.76 ± 0.02 $\sigma = 0.09$
3-D FFT	0.504 ± 0.002 $\sigma = 0.006$	0.518 ± 0.003 $\sigma = 0.013$

Table 7.9: Application execution times in seconds on 4x4 Clump and 16-way NOW. Each processor uses a private NIC, eliminating communication contention.

memory accesses also have a significant effect on 3-D FFT and CON/comp performance. For the CON/comp run, this effect slightly outweighs the benefit of fast communication, thus CON/comp requires 2% more time on the Clump than on the NOW.

Returning to Table 7.8, consider the effect of optimizing the layout of virtual processors on the execution time of EM3D. The EM3D/naive run requires 3.654 seconds on the Clump, while the EM3D/good run requires only 3.402 seconds, or 93% of the previous time. The improvement is less dramatic than one might expect given that the layout in EM3D/good transforms an average of one-third of the remote communication traffic into local traffic. Two factors mitigate the effects of the improved layout: the static process to NIC relationship and the phase structure of the application.

The static process to NIC relationship leads to communication contention, as discussed earlier. Although EM3D/good reduces the average amount of remote traffic, the reduction perceived by individual NIC's is not uniform. In fact, due to the specific process-to-NIC mapping employed, one NIC handles the same amount of remote traffic in EM3D/good as it does in EM3D/naive.

The use of a phase-structured style amplifies the importance of the first factor by negating uneven improvements across processes. After completing a phase, a process waits for all other processes to finish the phase before beginning the next phase. This global synchronization implies that a phase executes at the speed of the slowest process. Processes that perform no remote communication in EM3D/good wait idly for those that do. This effect is apparent in the load balance between processes, the ratio of the slowest process'

	Enterprise 5000 Clump	32-way UltraSPARC 170 NOW
EM3D/naive	2.36 ± 0.01 $\sigma = 0.05$	1.73 ± 0.02 $\sigma = 0.07$
EM3D/good	7.74 ± 0.03 $\sigma = 0.10$	1.64 ± 0.02 $\sigma = 0.09$

Table 7.10: Load imbalance in EM3D. The values in the table represent the ratio of execution time between the slowest and the fastest process in the communication phase.

execution time to that of the fastest. Table 7.10 reports the load balance in the two EM3D runs on both the Clump and on the NOW. On the latter platform, the processes perform roughly the same amount of work, and the load balance is within a factor of two; in fact, the actual workloads are more closely matched, but the application-level blocking behavior spreads them apart. EM3D/naive maintains a reasonable balance on the Clump as well, with a factor of 2.36 separating the slowest from the fastest process. With the optimized virtual processor layout, however, some processes perform no remote communication, and the workloads are very skewed: the fastest process in EM3D/good finishes 7.74 times as fast as the slowest.

7.4 Discussion

The results just reported illustrate the effects of contention, interconnect latency, and load imbalance on Clump application performance. In this section, we describe these problems and their significance in more detail, then touch on their interactions with a phase-structured style of programming.

7.4.1 Contention

Contention occurs at all levels of the memory hierarchy below the L2 cache. Contention for NIC resources dominates our application results, but contention for the memory interconnect and for memory itself can also significantly impact performance. Contention within the network is addressed in [CC97].

NIC contention occurs when more than one process initiates or receives remote communication simultaneously. The notion of simultaneity here refers to the presence of messages in multiple remote queues—each NIC processes only a single message at any time, and must multiplex traffic from multiple processes to and from the physical network. When more than one process communicates through a NIC, the processes perceive lower throughput and longer and more broadly distributed latencies. Total NIC throughput remains roughly the same [CMC98]. The impact of NIC contention on performance depends both on the balance between processors and NIC's in a system and on the communication demands of the processes that make up an application. Numerous studies in the literature assume either explicitly or implicitly that an SMP need support only one NIC. The recent offerings in low-end servers, in fact, support only one NIC for as many as four processors. Clumps built with such SMP's are unlikely to compete effectively with NOW's using the same NIC's on applications with non-negligible communication demands. Similarly, only very communication-intensive applications require that each SMP provide a separate NIC for each processor. The static mapping between processes and NIC's in our system slightly exacerbates contention by dividing an Enterprise 5000 into four two-processor virtual machines with one NIC each. However, the dominant role of NIC contention in our system arises from interactions with the phase structure of the programs, as we discuss later in this section.

Neither memory nor interconnect connection contributes significantly to our results. With respect to memory contention, the Enterprise 5000 design implicitly matches memory performance to peak demands by adding memory bandwidth with processors. Each processor board in the system bears two CPU's and two memory banks, scaling memory bandwidth with the number of processors and minimizing the effects of memory contention. As to contention at the interconnect, the Gigaplane is fully pipelined and allows multiple outstanding transactions to individual cache lines. Peak bandwidth is 2.7 GB/sec, and sustainable bandwidth is 2.5 GB/sec, an amount that easily satisfies the eight processors in our system. In contrast, neither the memory bandwidth nor the interconnect bandwidth are adequate for the processors in low-end SMP's. To illustrate the potential impact of inadequate memory bandwidth, we replicate results from [LMC97] in Table 7.11. The times in the table were obtained on one of our SMP's with six of eight memory banks removed. The aggregate memory bandwidth in the resulting system could support the peak memory demands of only two processors, increasing application execution times.

	8-way SMP		NOW (8 proc.)	
	Shared Memory	Multi-Protocol		Network
3-D FFT	8.7	9.0	7.0	6.76
CON/comm	1.96	2.1	4.01	4.01
CON/comp	1.40	1.44	1.18	1.19
EM3D	7.7	8.6	33.9	33.8

Table 7.11: Application execution times in seconds from [LMC97]. These results demonstrate the effect of inadequate memory bandwidth on performance. The application input parameters are not the same as those used in this thesis, however, thus the times are not directly comparable with those presented in Table 7.7.

7.4.2 Interconnect latency

Interconnect latency also affects performance on Clumps. The complexity of multiprocessor support and higher peak demands in an SMP create a tradeoff between price and performance. As price cannot be driven arbitrarily high in a commercial architecture, performance must suffer to some degree. Compared with their workstation counterparts, low-end SMP's penalize both throughput and latency. High-end SMP's attempt to retain or even improve throughput at the expense of latency. Recalling the base parameters for the Enterprise 5000 and Model 170 workstations as presented in Chapter 5, the SMP provides 184 MB/sec, or 10% higher memory copy bandwidth than the 168 MB/sec available on the workstation. Realized bandwidth degrades when multiple processes perform memory copies simultaneously, but requires four or more such processes before falling to the workstation value. Memory access, in contrast, requires requires 51 cycles on the SMP, or 16% more time than the 44 cycles required on the workstation. Latency to the NIC memory is also longer on the SMP, requiring 153 cycles to read a 32-bit word, or 34% longer than the 114 cycles required by the workstation. Contention for the SMP interconnect further inflates latency by stalling coherence transactions until the bus becomes available. The relative increases are thus fairly large, but their impact on performance is typically mitigated by data access locality in application programs. The effect of higher interconnect latency is clear only in the CON/comp results, exposed by a combination of a limited amount of communication and poor locality in performing simple operations on a graph.

7.4.3 Load imbalance

The last performance problem, load imbalance, takes two forms in our system: uneven improvements and uneven communication demands. Relative to a NOW, the Clump's ability to route some communication traffic across the SMP memory interconnects improves performance. However, the benefit to each process in a parallel job may be different, as some processes may send and receive only local messages while others use the network for the bulk of their communication. Such uneven improvements result in load imbalance. The results of the previous section examined the effect of optimizing the layout of virtual processors for the EM3D application, demonstrating only a small improvement. In that case, the improvement in layout exacerbated the uneven communication improvements, resulting in further load imbalance. The second form of load imbalance occurs within the NIC's themselves: the communication traffic sent through a NIC depends on the processes that make use of it. In our system, each process is statically mapped to a single NIC for the lifetime of the parallel job. As the distribution of network communication demands among processes is uneven, the resulting distribution of demands on each NIC is also uneven. More heavily loaded NIC's deliver lower throughputs and longer messages latencies.

Neither form of load imbalance is necessarily detrimental to performance. An application that allows sufficient overlap between communication and computation can hide the higher latencies due to uneven communication demands. Alternatively, an intelligent runtime system may schedule communication to minimize the impact of contention on performance. Hiding latency and scheduling communication are not easy, however. The applications studied in this thesis are designed to hide latency when possible, but are also fairly communication intensive.

7.4.4 Programming model

The phase-structured style of programming used in our applications has achieved some degree of popularity for parallel programming, particularly for message-passing codes such as those written to the MPI specification [MPI97]. A phase-structured approach can interact with the effects just described to further reduce performance, however. One approach to addressing computational load imbalance, for example, is to shift work to idle processes. A programming model that allows such dynamic scheduling of computation effectively balances the load across the processes. Dynamic scheduling and the phase-structured approach are not often mixed in explicitly parallel languages, however. Although phase structure does not prohibit dynamic scheduling within each phase, few programming environments provide support for such a style. Programmers can build their own task schedulers, and a few do, but many parallel applications apply a static schedule for each process in a job. The difficulty of debugging dynamically scheduled applications, particularly without any support in the programming environment, encourages programmers to pursue the static approach.

Static scheduling degrades performance on a Clump architecture, however, as improvements in the execution time of a phase reflect only the minimum of the per-processor improvements. When using a Clump, a programmer must put in a greater effort to balance the load between processes. Balancing the load for an unknown architecture is difficult—a well-balanced code on a Clump with eight processors per SMP may be very poorly balanced when run on a Clump with four processors per SMP. A programming model that incorporates dynamic scheduling provides such balance implicitly and is an attractive approach to abstracting the complexity of scheduling.

The use of phase structure also affects NIC contention by constraining the processes to communicate at the same time. Remote communication between two global synchronizations is always fully correlated between the processors, resulting in increased levels of contention for the NIC resources during communication phases of an application. In the next chapter, we develop an analytic model that captures the effect of this correlation and the other performance issues illustrated by our data.

Chapter 8

Performance Model

The last three chapters reported the performance of our multi-protocol communication layer in detail, comparing our software architecture with alternatives and exploring the interactions between software performance and characteristics of the hardware. This chapter approaches performance from the opposite extreme, applying a high-level model of communication to illustrate the relationship between performance and an abstraction of the Clump architecture. We begin the chapter with a brief overview of the model, then introduce the queueing theory that forms the basis for our computations. After extending the model to include more realistic phenomena, the chapter illustrates the resulting formulae graphically, bridging the gap between the previous chapters on performance and the technical issues discussed in the introduction. The chapter concludes with a discussion of the limitations of the model.

8.1 Overview of the Model

The model developed in this chapter assumes a communication layer that abstracts multiple physical network interface cards as a single virtual device. Although such an abstraction presents several problems not solved by this thesis, a more general model results from the assumption. Virtualizing multiple NIC's as a single device implies the existence of a mechanism and protocol for dynamically balancing communication demands evenly amongst the NIC's, allowing us to disregard the issue of uneven communication demands. As we alluded earlier, our experimental system is a special case of the more general model:

the static mapping between processes and NIC's makes each 8-processor SMP equivalent to four distinct 2-processor SMP's for the purposes of balancing communication traffic.

The model assumes a dedicated system, with one process executing concurrently on each processor. To avoid confusion between the applications processes that make up a parallel job and the stochastic processes used to develop the model, the remainder of the chapter uses the term processor in place of the term process when referring to communicating entities.

Using the model, we develop equations relating application performance on a Clump constructed from P-processor SMP's with N NIC's each to performance on a NOW with an equivalent number of processors. The model abstracts applications in terms of NIC utilization on a NOW, a measure of the communication duty cycle. The model treats communication within each process as an indivisible, independent, and continuous operation, ignoring computation overlaps, synchronization dependencies, and discrete messages. These simplifications allow the model to focus on the relationship between application performance and two aspects of the abstract machine: the balance between processors and NIC's within each SMP and the degree of correlation in the communication demands. The resulting equations capture the issues of pooling and fractional scaling of the NIC resources, allowing us to explore some of the potential advantages of the Clump architecture and to understand the effect of adding or removing NIC's from a Clump. Tight coupling affects the NIC utilization of an application, and we incorporate this issue into the model by adjusting that parameter appropriately.

8.2 Queueing Model

We begin with a model of communication as a closed network of queues. The model introduced in this section is nearly identical to an example by Kelly [Kel79], who also provides a good introduction to the underlying theory. Consider P processors numbered 1 through P with shared network interface resources. Each processor alternates between communication and idle (non-communication) states. At any point in time, an *active* processor—a processor in the communication state—receives a fair share of the communication resources.

Model this system as a network of $P+1$ queues: a private idle queue for each processor and a shared communication queue, as depicted in Figure 8.1. To make the

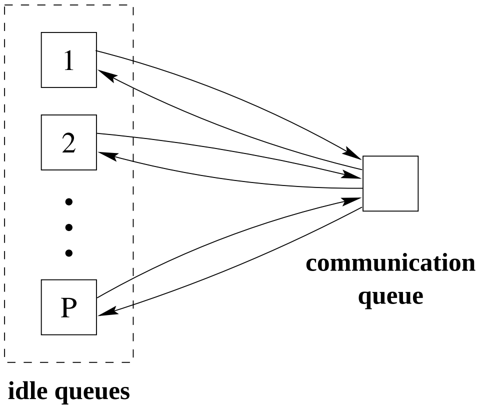

Figure 8.1: Model of network interface sharing. Processors move between private idle queues and a shared communication queue. The communication queue acts as a single-server queue with a server-sharing discipline [Kel79].

model tractable, assume that each processor p operates independently, with a mean service time of R_p in its idle queue and a mean service time of S_p in the communication queue. Assume that both service time distributions are Poisson. Finally, treat the communication queue as a single-server queue with a server-sharing discipline: when Q processors occupy the queue, the transition rate for a processor p to leave the queue is $(QS_p)^{-1}$. In terms of a real system, server-sharing implies that all active processors receive their fair share of service from the NIC.

Let $\sigma(t) \subseteq \{1, \ldots, P\}$ be the set of active processors at time t. Denote by $\sigma(t)$ a Markov process with the transition rate from a state σ to a state τ defined by

$$q(\sigma, \tau) = \begin{cases} \frac{1}{|\sigma|S_i} & \text{if } \exists p \in \sigma \text{ s.t. } \tau = \sigma \setminus \{p\} \\ \frac{1}{R_i} & \text{if } \exists p \notin \sigma \text{ s.t. } \tau = \sigma \cup \{p\} \\ 0 & \text{otherwise} \end{cases}$$

8.2.1 Process reversibility

We want to show that $\sigma(t)$ is reversible—that any evolution of the network has the same probability when reversed in time. A reversible process is stationary (time-independent) and has a unique equilibrium distribution, making a first-cut analysis substantially simpler. Kolmogorov's criteria for reversibility state that a Markov process is reversible if, and only if, for any finite sequence of states $(\sigma_1, \ldots, \sigma_N)$, the products of the transition probabilities around the loop induced by the sequence are the same in either direction. That is,

$$q(\sigma_1, \sigma_2) \ldots q(\sigma_{N-1}, \sigma_N) q(\sigma_N, \sigma_1) = q(\sigma_1, \sigma_N) q(\sigma_N, \sigma_{N-1}) \ldots q(\sigma_2, \sigma_1) \tag{8.1}$$

In demonstrating that these equations hold for our network, we need only consider cycles in which every transition consists of either the addition of a processor not already in the communicating set or of the removal of a processor from the communicating set. Other transition rates $q(\sigma, \tau)$ and their corresponding inverse rates $q(\tau, \sigma)$ are 0 and satisfy (8.1) trivially.

The transition rates for our network consist of two components: the base rates R_p^{-1} and S_p^{-1} and the NIC sharing factor $|\sigma|^{-1}$. The base rates depend only on the processor being added or removed from the set of communicating processors. For any directed cycle, the restriction of transitions around the cycle to those that affect a processor p results in an even-length sequence alternating between additions and removals of p. In the reversed cycle, additions become removals and vice-versa, and the sequence of transitions is exactly the same. The base rates hence contribute factors of the form $(R_p S_p)^{-n_p}$ equally to each side of (8.1), where $n_p \in \{0, \ldots\}$ is the number of addition-removal pairs for processor p in the cycle.

The second component of the transition rates, the NIC sharing factor, depends only on the number of processors in the communication queue. Consider the reduced set of states defined by the equivalence classes

$$C_q = \{\sigma \subseteq \{1, \ldots, P\} \mid |\sigma| = q\} \tag{8.2}$$

Call a transition from a state in C_{q-1} to a state in C_q an upward transition to q. Similarly, a downward transition from q is a transition from a state in C_q to a state in C_{q-1}. Each downward transition from q contributes a factor of q^{-1} to the product of transition rates around the cycle; upward transitions do not contribute. All transitions in a given cycle are either upward transitions from $q \in \{1, \ldots, P\}$ or downward transitions to such a q, and the number of upward transitions to any q is equal to the number of downward transitions from that q. An upward transition to q in the forward cycle becomes a downward transition from q in the reversed cycle, and vice-versa. The number of downward transitions to q is thus the same in the forward and reversed cycles, and the factor contributed to each side of (8.1) is the same. Hence our network is reversible.

8.2.2 Equilibrium distribution

The fact that our network is reversible implies the existence of an equilibrium distribution $\pi(\sigma)$ satisfying the detailed balance equations:

$$\forall \sigma, \tau \qquad \pi(\sigma) q(\sigma, \tau) = \pi(\tau) q(\tau, \sigma)$$

which for our network are:

$$\forall \sigma, \forall p \in \sigma \qquad \frac{\pi(\sigma)}{|\sigma| S_p} = \frac{\pi(\sigma \setminus \{p\})}{R_p}$$

The solution has the form:

$$\pi(\sigma) = B\,|\sigma|! \prod_{p \in \sigma} \frac{S_p}{R_p} \tag{8.3}$$

where B is a normalization constant defined such that the probability over all states sums to unity.

8.2.3 Utilization ratios

The equilibrium distribution given by (8.3) does not depend on the absolute values of the S_p and R_p, but only on the ratios S_p/R_p. Characterize a processor p by its utilization of the NIC resources: $U_p \equiv S_p/(S_p + R_p)$. U_p falls in the interval $(0, 1)$ and corresponds to the communication duty cycle of the processor p when using a private NIC. In terms of U_p, the equilibrium distribution becomes:

$$\pi(\sigma) = B\,|\sigma|! \prod_{p \in \sigma} \frac{U_p}{1 - U_p} \tag{8.4}$$

Processor utilizations of 0 and 1 can be handled through slight modifications to the network of queues. Equivalently, the equations can be changed as follows. Denote the set of processors with utilization 0 by υ_0 and the set of processors with utilization 1 by υ_1. Restrict the state space to $\{\sigma \subseteq \{1, \ldots, P\} \setminus \upsilon_0 \mid \upsilon_1 \subseteq \sigma\}$ and set all transition rates out of this space to 0. Finally, replace the product over processors in σ in the equilibrium distribution with a product over processors in $\sigma \setminus \upsilon_1$.

8.2.4 Service efficiency

A real system incurs overhead when sharing a resource between multiple processors. In the model, we represent this overhead through the inclusion of an efficiency factor

$E(|\sigma|)$ in transition rates of the form $q(\sigma, \sigma \setminus \{p\})$. Note that this change simply extends the notion of the NIC sharing factor and does not affect the reversibility of the network. Defining

$$E_q \equiv \prod_{i=1}^{q} E(i)$$

we incorporate communication service efficiency into (8.4) as

$$\pi(\sigma) = B \, \frac{|\sigma|!}{E_{|\sigma|}} \prod_{p \in \sigma} \frac{U_p}{1 - U_p} \tag{8.5}$$

8.3 Extensions to the Basic Model

In this section, we apply the model developed in the previous section to applications running on a Clump. We begin by studying the effect of using multiple NIC's in an SMP, then improve the model by incorporating aspects of phase structure. Our metric for these evaluations is the slowdown (the reciprocal of speedup) with respect to a system built from 1-processor, 1-NIC machines, *i.e.*, a network of workstations (NOW). For simplicity, we assume in this section that $\forall\ q \in \{1, \ldots, P\},\ E_q = 1$—that the system has efficient communication. Service efficiency reappears in the discussion of the model's limitations.

8.3.1 Incorporating multiple NIC's

For an SPMD program, it is reasonable to assume that NIC utilization is roughly the same for every processor: $\forall p,\ U_p = U$. The equilibrium distribution then simplifies to:

$$\pi(\sigma) = B \, |\sigma|! \, \left(\frac{U}{1-U}\right)^{|\sigma|}$$

which depends only on the number of active processors $|\sigma|$. Recalling the equivalence classes C_q defined by (8.2), we can write an equilibrium distribution $\pi(C_q)$ as:

$$\forall q \in \{0, \ldots, P\} \qquad \pi(C_q) = \sum_{\sigma \in C_q} \pi(\sigma) = B \, \frac{P!}{(P-q)!} \left(\frac{U}{1-U}\right)^q \tag{8.6}$$

$$\text{since} \qquad |C_q| = \binom{P}{q} = \frac{P!}{q!(P-q)!}$$

Suppose that we replace the NIC in an SMP with N NIC's of equal speed. Equivalently, we might employ a single, more powerful NIC. In terms of the model, we increase all transition rates of the form $q(\sigma, \sigma \setminus \{p\})$ by a factor of N, the number of NIC's. Propagating this change through the equations, we rewrite (8.6) as:

$$\pi(C_q) = B \, \frac{P!}{(P-q)!} \, \left[\frac{U}{N(1-U)}\right]^q \tag{8.7}$$

8.3.2 The slowdown metric

An alternative view of the queueing network is useful in understanding the effect of the parameters N and P on application performance. When a processor arrives at a queue, it requires a certain amount of service from that queue; the specific amount is a random variable. Service time in the idle queue accumulates at a rate of 1, and service time in the communication queue accumulates at a rate of N/q. When a processor accumulates enough time to satisfy its requirement, it moves to the other queue. The quantity of interest is then the average service rate $[\bar{X}(P, N, U)]^{-1}$. Given the accumulation rates just mentioned, $\bar{X}(1, 1, U) = 1$, and $\bar{X}$ represents the ratio of a program's execution time when competing with P processors for N NIC's to its execution time with one private NIC. $\bar{X}$ is hence the slowdown of a program with utilization U on a P-processor, N-NIC configuration.

Before we can write $[\bar{X}(P, N, U)]^{-1}$ as a sum over the C_q, we must calculate the probability that a particular processor is active when the network is in a state $\sigma \in C_q$. Without loss of generality, pick processor 1 and let $C_{q,comm}$ be the subset of C_q in which processor 1 is active and $C_{q,idle}$ be the subset of C_q in which processor 1 is idle. Clearly, $C_{q,comm}$ and $C_{q,idle}$ are disjoint, and their union is C_q. Each state in C_q is equally likely at equilibrium, and

$$|C_{q,comm}| = \left(\begin{array}{c} P-1 \\ q-1 \end{array} \right) = \frac{q}{P} \, |C_q|$$

hence

$$\forall q \in \{0, \ldots, P\} \qquad \pi(C_{q,comm}) = \frac{q}{P} \, \pi(C_q)$$

$$\pi(C_{q,idle}) = \frac{P-q}{P} \, \pi(C_q)$$

Using these results, we write:

$$\begin{aligned}
[\bar{X}(P,N,U)]^{-1} &= \pi(C_0) + \sum_{q=1}^{P}\left[\pi(C_{q,comm})\,\frac{N}{q} + \pi(C_{q,idle})\right] \\
&= \pi(C_0) + \sum_{q=1}^{P}\pi(C_q)\left[\frac{q}{P}\,\frac{N}{q} + \frac{P-q}{P}\right] \\
&= \pi(C_0) + \sum_{q=1}^{P}\pi(C_q)\left[1 + \frac{N-q}{P}\right] \\
&= 1 + \sum_{q=1}^{P}\pi(C_q)\,\frac{N-q}{P} \qquad (8.8)
\end{aligned}$$

8.3.3 Correlated and scheduled demands

The queueing model assumes that the processors' communication demands are independent. In real applications, and particularly in programs written in a phase-structured style, communication demands are likely to be partially or even fully correlated. Similarly, a communication-aware compiler or runtime system may avoid some contention through scheduling. We now derive the forms of $\bar{X}(P,N,U)$ for fully correlated and scheduled demands to help illuminate the differences between these cases.

Fully correlated communication presents no difficulty: the processors move in and out of the communication queue as a group, and the network is always in C_0 or C_P. The detailed balance equation is:

$$\frac{N\pi(C_P)}{PU} = \frac{\pi(C_0)}{1-U}$$

$$\text{so} \qquad \pi(C_0) = \frac{N(1-U)}{N(1-U)+PU} \qquad \pi(C_P) = \frac{PU}{N(1-U)+PU}$$

$$\text{and} \qquad \bar{X}_{correlated}(P,N,U) = 1 + \left(\frac{P-N}{N}\right)U \qquad (8.9)$$

Note that the average slowdown caused by correlated communication is linear in U, as we might expect, since all such communication is slowed by a factor of P/N.

Scheduled communication requires more thought. $[\bar{X}(P,N,U)]^{-1}$ is technically an average over time, but can be written as an weighted average over states in a stationary Markov process. However, transitions in a scheduled system are explicitly controlled, and the resulting process is not stationary, forcing us to retreat to an explicit time average. Suppose that, in 1 time unit and with one private NIC, a processor communicates for U time units and idles for the remaining $1-U$ time units. Moving to a system with P processors and N NIC's affects only the time required for communication—the idle time remains a constant $1-U$ time units. If communication service time accumulates at a rate S^{-1}, the execution time is $1-U+SU$, and we can write:

$$\bar{X}_{scheduled}(P,N,U) = \frac{1-U+SU}{(1-U)1+SU\frac{1}{S}} = 1+(S-1)U$$

We define the execution time of a program to be the execution time of its slowest processor; a scheduler gains obtains no benefit by making some processors finish before others. Scheduled systems divide into two domains based on the value of U. For small U, all communication makes use of all NIC's, and communication service time accumulates at a rate of N. For large U, the NIC's are saturated, and the total execution time is equal to the total communication time, PU/N. The boundary value U_s occurs when the idle time for a process is equal to the time spent communicating for all other processes:

$$(P-1)\frac{U_s}{N} = 1-U_s$$

$$\text{or} \qquad U_s = \frac{N}{N+P-1}$$

This result allows us to write

$$\bar{X}_{scheduled}(P,N,U) = \begin{cases} 1-\left(\frac{N-1}{N}\right)U & \text{for } U \le U_s \\ \frac{PU}{N} & \text{for } U \ge U_s \end{cases} \tag{8.10}$$

For both domains, slowdown is linear in U. Notice that $\bar{X}$ increases monotonically to either side of U_s. Writing

$$\bar{X}_{min}(P,N) = X_{scheduled}(P,N,U_s) = \left[1+\frac{N-1}{P}\right]^{-1}$$

we see that a system of P-processor SMP's with N NIC's gives at best a factor of $1+\frac{N-1}{P}$ speedup over a NOW due to sharing of communication resources.

8.3.4 Incorporating program structure

The degree of correlation between processors' communication demands may vary as the processors execute different sections of a program. For a phase-structured program, we can extend our model to include information on the phase structure of the program. Recall that in a phase-structured program, processors move in tandem from one phase to the next but operate independently in each phase.

A program consisting of F phases may present a distinct NIC utilization U_f for each phase. We expect all processors to occupy the same phase at any point in time. For simplicity, we also ignore the effects of transients as the system moves between the equilibrium distributions for successive phases of the program. Such an assumption is reasonable if a system settles into each new equilibrium in a time short compared with the length of the phase. If we know the phase execution times when running on a NOW, denoted by T_f for phase f, we can write

$$\bar{X}(P, N, (U_1, \ldots, U_F)) = \frac{\sum_{f=1}^{F} \bar{X}(P, N, U_f) T_f}{\sum_{f=1}^{F} T_f}$$

8.4 Performance Example

The slowdown formulae quantify the impact of communication correlation due to the phase-structured approach and bound the potential improvement possible with scheduled communication. The formulae provide insight on the issues of communication demand pooling and fractional scaling in Clumps. Tight coupling of resources, demonstrated by the benefit of fast communication to SMP and Clump performance, requires a minor extension to the model. This section presents the necessary extension and illustrates the three technical performance issues—tight coupling, pooling, and fractional scaling—graphically for our experimental platform.

8.4.1 Tight coupling

The SMP memory interconnect serves as a fast path for data traveling between processes within the SMP, coupling the processors with one another above the level of the network. Compared with communication on a NOW, processors passing messages across a cache-coherent interconnect perceive lower latency and higher bandwidth. The empirical results in Chapters 6 and 7 demonstrate this tight coupling effect, but our model of performance has yet to capture the advantages.

We integrate tight coupling into the model by treating it as an independent factor in application slowdown. Effectively, the presence of a multi-protocol communication layer allows each process to transform a portion of its communication time into idle time. In a real system, processes spend the additional "idle" time pulling data across the interconnect, but the model distinguishes activity only on the basis of interactions with the network resources.

Recalling the explicit time-based approach used to derive scheduled communication slowdown, we say that, in 1 time unit and with one private NIC, a processor communicates for U time units and idles for the remaining $1 - U$ time units. Define the tight coupling transformation in terms of two variables: f, the fraction of communication that must cross the network; and V, the cost ratio between local and remote communication. Both of these variables should fall in the interval $[0, 1]$. Tight coupling transforms the U time units of communication into the following:

$$fU + (1 - f)VU \tag{8.11}$$

The first term represents the time spent on the remaining network communication, and the second term represents the time spent on local communication. Adding $1 - U$ time units of computation to (8.11) brings the total to

$$\begin{aligned} & (1 - U) + fU + (1 - f)VU \\ \text{or} \quad & 1 - U(1 - f)(1 - V) \end{aligned}$$

As processes spend less time communicating, we adjust their utilization to incorporate the new timing as follows:

$$U_{adj}(f, V) = \frac{fU}{1 - U(1 - f)(1 - V)}$$

$U_{adj}(f, V)$ is never greater than U. Including tight coupling into application slowdown involves multiplication by a single factor and replacement of U with U_{adj}:

$$\bar{X}_{tc}(P, N, U, f, V) = \bar{X}(P, N, U_{adj}(f, V))\,[1 - U(1 - f)(1 - V)]$$

8.4.2 Pooling

The ability to smooth communication resource demands by pooling the demands of individual processes relies on the independence of those demands. As discussed earlier, phase-structured programming creates interdependencies between the individual demands by enforcing global synchronization before and after communication. These correlated communication demands do not benefit from aggregation. The pooled demands are as bursty as

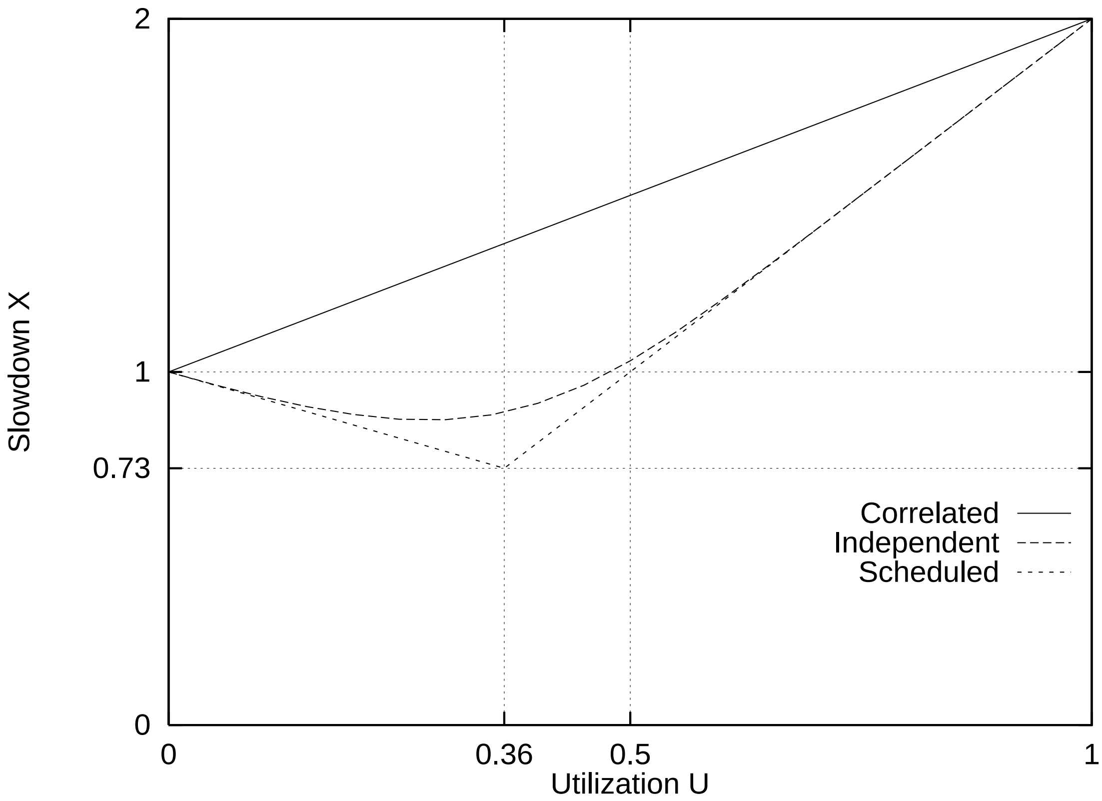

Figure 8.2: Impact of correlation on application slowdown. Each line reports application slowdown on a Clump constructed from 8-processor SMP's with 4 NIC's relative to a NOW for a distinct level of correlation between the processes' communication demands.

the individual demands, and the contention during communication bursts degrades overall performance. The model developed in this chapter captures the effect of communication correlation and further amplifies our understanding with information on the performance of anticorrelated, or scheduled, individual demands.

Figure 8.2 illustrates performance on a Clump similar to the experimental platform used in this thesis. Each SMP contains eight processors and four NIC's. The vertical axis reports application slowdown relative to a NOW. The upper line charts slowdown for fully correlated communication demands, as expressed by (8.9). The middle line graphs slowdown for independent communication demands as calculated with (8.8). The lower line bounds the potential performance by reporting slowdown for scheduled communication, as described by (8.10). Ignoring the advantage of tight coupling within each SMP, we expect no application to execute in less than 73% of the execution time on a NOW. Applications

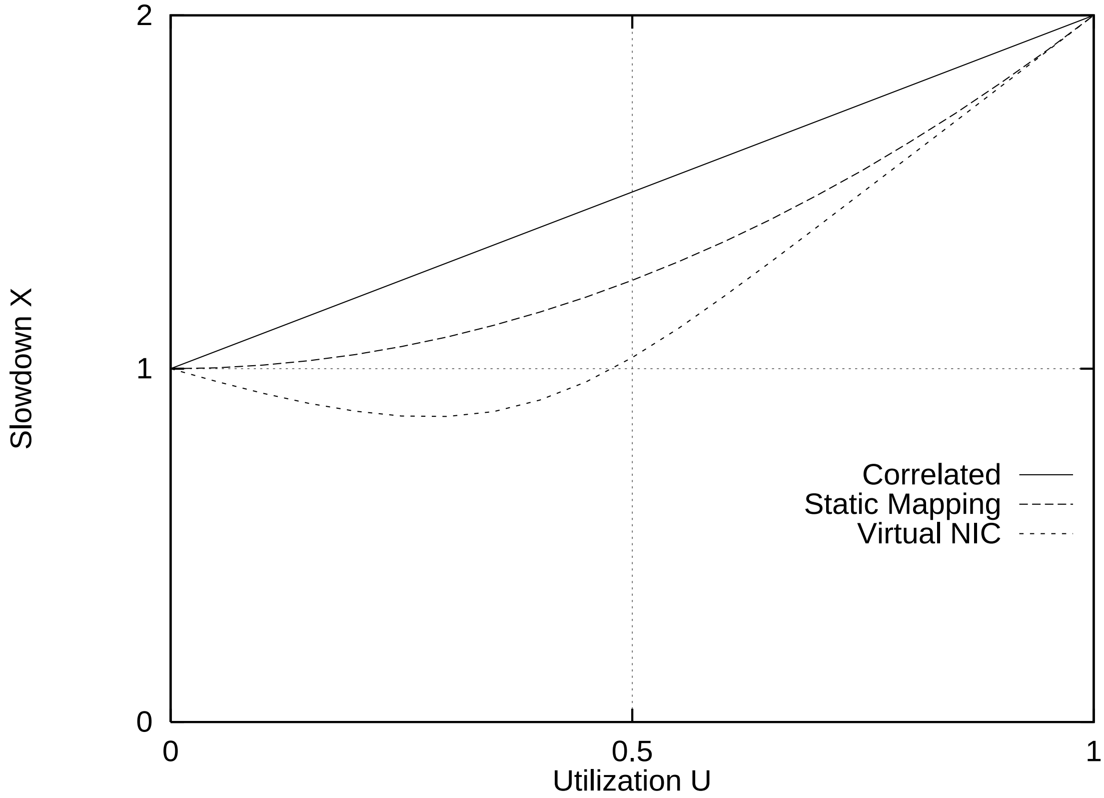

Figure 8.3: Impact of static process to NIC mapping on application slowdown. The upper line reports calculated application slowdown on our system; the lower line reports the same value on a system that efficiently virtualizes the four NIC's as a single device.

without support for communication scheduling can execute no faster than 86% of NOW execution time, the minimum value on the independent line in the figure.

The separation between the correlated and independent slowdown lines demonstrates the advantages of both pooled demands and pooled resources. Resource pooling allows the independent slowdown to drop below one at reasonably low communication utilization. Applications with independent demands and utilization under 0.48 should perform better on the Clump, even though the Clump has only half as many NIC's as an equivalent NOW.

A comparison between our system and a system that virtualizes multiple NIC's allows us to separate the benefits due to each type of pooling. Figure 8.3 graphs slowdown for the two systems, which are both built from 8-processor SMP's with four NIC's each. The lower line replicates the independent slowdown from the previous figure, which assumed the virtualization of the four NIC's as a single device. Such a system pools both demands and resources. The middle line presents independent slowdown for our system, in which the communication demands of two processors are statically mapped to each NIC. The static mapping does not allow resource pooling, and performance is always inferior to that of the NOW, again ignoring the advantage of tight coupling. The upper line presents slowdown for correlated demands on either system, in which neither type of pooling is effective. The static mapping sacrifices about half of the benefit of demand independence by preventing resource pooling.

8.4.3 Fractional scaling

Virtualization of the physical network resources also increases the precision of communication resource scaling. Workstations in a NOW must utilize an integral number of NIC's; many real workstations in fact offer only a single I/O bus. An SMP must also utilize an integral number, of course, but need not employ a multiple of the number of its processors. We can thus take an alternate view of SMP's, allowing P-processor SMP's to scale their network resources in units of $1/P$ per processor. We term this incremental view fractional scaling.

The potential advantage of fractional scaling is illustrated by Figure 8.4. The figure graphs application slowdown for a Clump constructed from 8-processor SMP's with one through nine NIC's, or one-eighth through one and one-eighth NIC's per processor. The SMP can continue to scale in units of one-eighth of a NIC per processor, but a NOW must jump to two NIC's per processor. Pooling allows an SMP with three-quarters of a NIC per processor to perform better than a NOW with two NIC's per processor for applications with utilizations below about 0.5. Considering tight coupling further improves the advantage of the SMP.

8.5 Limitations of the Model

The model developed in this chapter makes a number of simplifying assumptions about the nature of communication in a Clump. We now review those assumptions and discuss their significance for the accuracy of the model. We begin with two abstract issues, variable utilizations between processors and transient states of the queueing model. We

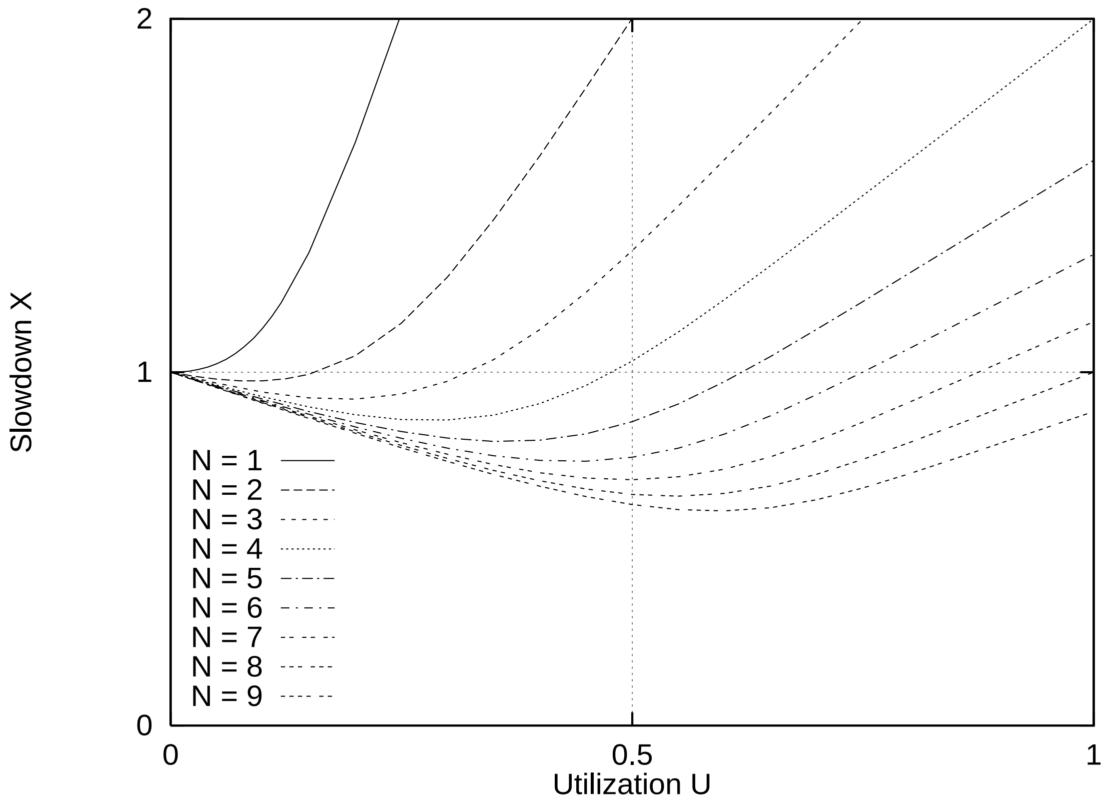

Figure 8.4: Impact of scaling on application slowdown. Each line reports application slowdown on an 8-processor Clump with the specified number of NIC's. Communication demands are assumed to be independent.

then address the assumptions of fairness and efficiency in terms of our own communication layer. The chapter concludes with a discussion of the predictive capacity of the model.

8.5.1 Variable utilization

Even in an SPMD code, processors executing a parallel job can perform different amounts of communication. Although dynamic scheduling can help to balance the communication workload as well as the computation workload, the processors retain some degree of variability in their utilization of communication resources. For applications with static scheduling, the effects are much more significant. With static scheduling, multi-protocol communication can magnify imbalances in the processors' network communication demands.

Unfortunately, although generalizing the solution to incorporate separate utilization factors for each processor is not hard to do in terms of the state space, using the resulting equations to calculate application slowdowns requires an effort exponential in the number of processors. We can generate data for small SMP's, but large SMP's present a problem.

A possible approach to the case of variable utilization with dynamic scheduling is to assume that the utilization for each processor is drawn from the same distribution and that utilizations are narrowly distributed around the mean value. Perturbation theory might then allow calculation of a slowdown distribution.

8.5.2 Transient effects

Transient effects also lead to inaccuracy in the model. At the start of a parallel job, no processor is communicating, and the system state is in C_0 with probability 1. The distribution of states eventually converges to the equilibrium distribution, but the time required to reach that distribution was not discussed. Simulations indicate that the time is fairly long, and some parallel jobs may not converge before they terminate. In such a case, transients between the initial and equilibrium distributions dominate the job's behavior. Phase structure creates additional transient effects by requiring that processors periodically stop communicating to synchronize.

The assumption of Poisson service time distributions for the model does not have much significance for the equilibrium distribution. Any pair of distributions with randomness in at least one of the service times results in similar behavior. Pairs of constant distributions tend to correlate communication by slowing the communicating processors while allowing non-communicating processors to proceed at full speed until they begin to communicate. For transient effects, service time distributions may play a larger role by affecting the time required to converge to the equilibrium distribution.

8.5.3 Fairness and efficiency

The server-sharing discipline used to model the communication queue assumes that NIC's provide equal service to each processor actively communicating. At very small time scales, this assumption cannot hold, as each NIC has only a single connection to the network. At very large time scales, the behavior is easy to achieve. We measured fairness in our layer with multiple pairs of processors to communicating between two SMP's using one NIC in each SMP. Each pair of processors consists of a sender and a receiver split between the SMP's; the sender sends a stream of short messages to the receiver and notes the time at which it receives each reply. From the reply times, we calculate the distribution of times required for successive bursts of a given length, then calculate fairness at that burst length as twice the standard deviation over the mean of the distribution. For bursts of 100 messages or more, fairness is within 10% for up to seven pairs of competing processors. Below 100 messages, fairness degrades quickly for many pairs and more slowly for fewer pairs. None of the processor pairs is systematically favored over others, hence unfairness should average out over the lifetime of the processors. Synchronization dependencies on short bursts may translate unfairly long response times into longer execution times, however.

The benefits attributed to resource pooling implicitly assume that a processor can employ more than one NIC with reasonable efficiency. A straightforward calculation using the LogGP parameters for the AM-II network protocol suggest a potential advantage for up to

$$\left\lceil \frac{g}{o_s + o_r} \right\rceil = \left\lceil \frac{14.51}{3.46 + 8.93} \right\rceil = 2$$

NIC's. This equation assumes that a processor must send a message and receive a reply on one NIC during each period of length g, but can spend the remaining time sending messages through other NIC's. However, the additional cost of polling the second NIC brings the overhead above the gap, eliminating the potential benefit. Resource pooling cannot benefit applications that use primarily short messages with our layer, thus applications can only perform better on the Clump than on the NOW due to the effect of tight coupling. This restriction does not affect applications that use bulk transfers, however, as DMA transfer times are relatively long compared to send and receive overheads.

8.5.4 Predictive capacity

The most significant barrier to the ability of the model to predict performance are the assumptions that communication is indivisible, independent, and continuous. Network communication with DMA and protocol processing support allows an application to hide the bulk of communication latency with computation. Interprocess synchronization encapsulated in messages can lead to dependency chains that dominate overall performance. Finally, the discrete nature of messages and queues can change a non-blocking scenario into a blocking one as flow control code begins operation.

Although the model captures performance issues for Clumps in a qualitative sense, a significantly more complicated model may be necessary to provide a reasonable level of quantitative accuracy. As an illustration of the difficulty of predicting performance on a Clump in a quantitative fashion, compare the level of abstraction embodied by the model developed in this chapter to the level of detail employed in relating the sources of systematic error in Chapter 5. Synchronization interdependencies in a parallel application often magnify the errors encountered at very low levels rather than averaging them into negligible effects. As evidence for this claim, consider the number of references made to the application-level blocking effect in EM3D in Chapters 6 and 7.

Chapter 9

Related Work

We have investigated issues related to efficient message-passing through both shared memory and the network within a Clump. In this chapter, we discuss a range of related work, beginning with studies on message-passing through shared memory and concurrent message queues. After covering other multi-protocol message-passing efforts, we continue with a discussion of distributed shared memory and programming models for Clumps. After a brief section on performance evaluation and metrics for multiprogramming, the chapter closes with discussion of a few less closely related issues.

9.1 Shared Memory Message-Passing

The idea of passing messages through shared memory is not novel. Numerous applications make use of these techniques, as do a number of operating systems. The Mach operating system, for example, uses message-passing between processes to construct a modular kernel [YTR+87]. Most systems, however, avoid placing concurrent access in the critical path of communication by allocating separate data structures for each sender-receiver pair during setup and teardown operations. Cheriton and Kutter [CK96] touches briefly on shared message segments, but only to mention their use in establishing a private segment for client-server communication. Byrd [Byr93] builds optimistic high-level models of non-concurrent shared memory communication to aid in the design of a system that minimizes end-to-end latency. A study by Lim *et al.* [LHPS97] uses shared memory to route network traffic between SMP's through a proxy process, but again the data structures are duplicated for each communicating pair. In Chapter 2, we argued against such an approach, identifying issues of performance, scalability, and memory usage as key reasons for developing efficient concurrent message queues. Our empirical results validate these arguments. Separate queues deliver superior results for one-to-one communication, but can be detrimental for more complex communication patterns. The breakdown of send overhead shows that polling a queue and managing concurrent access incur similar amounts of overhead. Polling queues scales linearly with the number of queues, whereas the communication stress test results demonstrate that concurrent access costs remain relatively constant with increasing numbers of processors. The demonstrated effect of queue length on application performance supports our argument against dividing total queue space between all possible senders, making concurrent message queues more scalable in terms of memory as well. This section discusses a broader cross-section of concurrent access algorithms than we explored empirically and compares our approach to providing hardware support for message-passing with other approaches.

Several studies have addressed both one-to-one and many-to-one queues in the context of reflective memory, which uses hardware such as DEC Memory Channel [GK97], SCI, or Shrimp/VMMC [BLA+94] to establish simplex channels that reflect the contents of memory within one machine into the memory of a second.

9.1.1 Concurrent message queues

We outlined a variety of alternative algorithms for concurrent access in Chapter 6, and the literature offers many more. The spin lock algorithms used in our work—Test & Set, Test & Test & Set, the ticket lock, and the Anderson lock—are drawn from Mellor-Crummey and Scott [MCS91]. Herlihy [Her93] is a good source for the theory of wait-free and non-blocking behavior as well as general implementation strategies. More practical implementations of non-blocking data structures are also abundant. Rudolph [Rud81] describes an implementation of a priority-based scheduler for the Ultracomputer operating system. Michael and Scott [MS97] surveys a variety of algorithms and evaluates their performance on an SGI Challenge. Massalin and Pu [MP91] provides evidence that non-blocking approaches can be efficient when tailored to operating system data structures, and Greenwald and Cheriton [GC96] shows that more general approaches can also be successful in this regime. Both kernel implementations made use of the double-compare-and-swap (DCAS) instruction, which performs simultaneous CAS operations on two independent words and is not generally available.

Lock-free algorithms provide the most competitive alternatives to our approach. We introduced three other lock-free algorithms from the literature in Chapter 6. Two of these algorithms are similar to our own. The first, based on work with the Cray T3D, implicitly links claiming a packet to packet assignment [BCL$^+$95, KC95]. A sender that reaches the end of the queue must wait for the receiver to reset the queue after processing all entries. Our use of an additional synchronization primitive allows us to avoid this boundary condition. The second, based on work with the NYU Ultracomputer, provides a many-to-many queue with explicit support for overflow and underflow [GLR83]. The more general nature of the algorithm adds overhead unnecessary in our system, but the core algorithms are essentially the same. The last algorithm takes a much different approach to managing the packet queue, but requires unaligned CAS primitives [Val94], a feature not supported by any modern architecture.

The use of a request-response communication paradigm makes true non-blocking behavior moot in our system; an endpoint that fails to respond blocks application progress. Non-blocking behavior also requires dynamic storage, a factor that does not mesh well with our model of pre-allocated shared segments. However, by increasing the size of the queue block to the point that no blocking occurs, we can simulate the possibilities with known applications to obtain reasonably accurate data on performance. We now discuss a few non-blocking and wait-free algorithms to illustrate the range of complexity and overhead.

Non-blocking algorithm

Non-blocking operations on a double-ended queue (deque) form the basis of work-stealing in the Cilk multithreaded language [ABP98]. Previous versions of the language used locking constructs [FLR98], but an investigation of performance revealed that the non-blocking algorithm was superior [BP98]. The non-blocking approach also plays a useful role in addressing performance in the face of an adversarial scheduler [ABP98]. The deque operations achieve non-blocking behavior by assuming an infinitely deep queue; a push operation that uses a bounded amount of memory and signals overflow can retain the non-blocking aspect provided that the runtime can handle the failure, perhaps by executing

$\textsc{Swap}(\hat{}\,address, new)$
    $old \leftarrow address\hat{}$
    $address\hat{} \leftarrow new$
    return old

$\textsc{Enqueue}(\hat{}\,q,\ \hat{}\,data)$
    $message \leftarrow \textsc{AllocateMessage}()$
    $message\hat{}.next \leftarrow$ NULL
    $message\hat{}.data \leftarrow data\hat{}$
    $old\_tail \leftarrow \textsc{Swap}(q\hat{}.tail, message)$
    if $old\_tail \neq$ NULL
        $old\_tail\hat{}.next \leftarrow message$

$\textsc{Dequeue}(\hat{}\,q,\ \hat{}\,data)$
    $head \leftarrow q\hat{}.head$
    $next \leftarrow head\hat{}.next$
    if $next =$ NULL
        return UNDERFLOW
    $q\hat{}.head \leftarrow next$
    $\textsc{FreeMessage}(head)$
    $data\hat{} \leftarrow next\hat{}.data$
    return SUCCESS

Figure 9.1: Pseudo-code for pull-based messaging [KC95]. The head element is an empty message buffer and is preserved until another message arrives. The SWAP synchronization primitive performs an unconditional exchange of one value for another.

the thread sequentially. Without such handling, the deque algorithm is lock-free, but not non-blocking.

Karamcheti and Chien [KC95] describes a non-blocking many-to-one queue algorithm based on the SWAP primitive and assuming a single address space. Implemented in the context of the Cray T3D, the algorithm uses dynamic allocation to support an interesting "pull-based" approach on a link-based queue. Pseudo-code for the algorithm appears in Figure 9.1. As shown in the figure, SWAP unconditionally exchanges the value at an address with a new value. Each receiver maintains head and tail pointers to a linked list of messages. A sender enqueues a message by allocating and filling a new message block, then placing it at the end of the receiver's list. The tail pointer is updated before the link in the previous tail message, greatly simplifying the algorithm. A receiver maintains a single empty message block as a sentinel at the head of the list. When dequeueing a message, the receiver frees the previous sentinel block and accepts the message in the next block, leaving the block itself to serve as a new sentinel. The term pull-based refers to the fact that the message data remains cached with the sender until the receiver accepts the message, providing implicit flow control for many-to-one communication patterns. This algorithm may be competitive with ours in an environment based on threads, and may also be a reasonable alternative when using multiple processes with our request-reply paradigm of communication.

The simplifying order of the tail and link update makes the pull-based algorithm non-linearizable [HW90]. Consider the following scenario. A slow sender swaps the tail of an empty queue, then a fast sender inserts a message after that of the slow sender. After the fast sender is done, the queue still appears empty to the receiver, and an attempted message reception returns no message. In selecting a linear order for these events, the fast sender's insertion must appear before the receiver's attempt to accept a message, as the send operation completed before the receive operation. Such an order is incompatible with the receiver's failure to receive the message, however. The FIFO property defined by Gottlieb *et al.* [GLR83] provides a more natural definition for queue semantics by ignoring the effect of failures. In the scenario described above, the pull-based algorithm does guarantee that the receiver accepts the slow sender's message before accepting the fast sender's message.

Michael and Scott [MS96] describes a more complex concurrent queue designed for multiple producers and multiple consumers. The data structures used by the algorithm are almost exactly the same as those used with the pull-based algorithm, but message block pointers include epoch numbers to avoid the ABA problem in practical use. Figure 9.2 presents pseudo-code for a simplified version of the algorithm that supports only a single receiver. The simplified version of the algorithm performs the same operations as the pull-based algorithm, but must verify the success of its modifications and attempt to help previous operations complete their work. The complexity in this case arises primarily from the algorithm's choosing to link in new messages before advancing the tail, the order opposite to that used in the pull-based algorithm. This order was chosen based on correctness and semantic failures (*e.g.*, non-linearizability) in previous work and on an assertion that safety requires that the tail always point to a message in the list. Both senders and the receiver must check that the tail pointer in fact points to the end of the list and try to advance it when it does not.

```
Enqueue(^q, ^data)
    message            ← AllocateMessage()
    message^.data      ← data^
    message^.next.ptr  ← NULL
    while TRUE
        tail ← q^.tail
        next ← tail.ptr^.next
        if tail = q^.tail
            if next.ptr = NULL
                if Compare&Swap(tail.ptr^.next, next, ⟨message, next.count + 1⟩)
                    Compare&Swap(q^.tail, tail, ⟨message, tail.count + 1⟩)
                    return
            else
                Compare&Swap(q^.tail, tail, ⟨next.ptr, tail.count + 1⟩)

Dequeue(^q, ^data)
    while TRUE
        head ← q^.head
        tail ← q^.tail
        next ← head^.next
        if head.ptr ≠ tail.ptr
            data^   ← next.ptr^.data
            q^.head ← next
            FreeMessage(head.ptr)
            return SUCCESS
        if next.ptr = NULL
            return UNDERFLOW
        Compare&Swap(q^.tail, tail, ⟨next.ptr, tail.count + 1⟩)
```

Figure 9.2: Pseudo-code for the non-blocking queue from Michael and Scott [MS96]. The dequeue operation shown is a simplified, sequential form of that given in the paper.

```
FETCH&INCREMENT(^obj)
    toggle                         ← not obj^.announce[MY_PROCESS]
    obj^.announce[MY_PROCESS] ← toggle
    for try ← 1 to 2
        if obj^.cblock^.toggle[MY_PROCESS] = toggle
            if obj^.cblock^.toggle[MY_PROCESS] = toggle
                break
        old_cblock = obj^.cblock
        COPY(new_cblock.ptr, old_cblock.ptr, CBLOCK_LEN)
        for i ← 0 to NUM_PROCESSES−1
            if obj^.announce[i] ≠ new_cblock^.toggle[i]
                new_cblock.ptr^.responses[i] ← new_cblock.ptr^.value
                new_cblock.ptr^.value        ← new_cblock.ptr^.value + 1
                new_cblock.ptr^.toggle[i]    ← obj^.announce[i]
        if COMPARE&SWAP(obj^.cblock, old_cblock, ⟨new_cblock.ptr, old_cblock.count + 1⟩)
            new_cblock ← old_cblock
            break
    return obj^.cblock^.response[MY_PROCESS]
```

Figure 9.3: Pseudo-code for a wait-free FETCH&INCREMENT with NUM_PROCESSES processes. The algorithm is a specific instance of a general approach outlined by Herlihy [Her93]. In theory, the use of COMPARE&SWAP makes this version suffer from the ABA problem.

Wait-free algorithm

Wait-free algorithms guarantee that every process makes progress in a finite number of their own time steps. The CAS operation is sufficient to construct wait-free implementations of arbitrary concurrent data types. These implementations avoid the possibility of starvation inherent to competitive algorithms by having processes cooperate with one another, completing each other's operations before performing their own. Naturally, cooperation incurs significant overhead, a cost that grows linearly with the number of processes. As an example of such an implementation, Figure 9.3 shows pseudo-code for a wait-free version of F&I implemented with CAS.

The F&I data structure consists of a set of process announcement toggles and a copy block pointer. The copy block contains the value of the emulated memory location, a set of process completion toggles, and a set of process return values. As indicated by its name, the copy block must be copied by every operation. The copy block pointer is also extended with an epoch number to reduce the likelihood of the ABA problem.

A process begins the F&I operation by announcing its intent to perform the operation. One-bit serves the purpose, as each process can execute only a single operation at a time. After making an announcement by flipping its bit, the process enters a two-iteration loop to attempt the change. If the change fails twice, another process is guaranteed to have completed the operation, as explained later. At the start of the loop, the process checks that no other process has completed the operation for it; two checks are required, as the copy block pointer may change after being read, leaving the process' completion toggle in the old copy block in a state that falsely confirms the success of the operation. A false positive in the second check implies that the copy block has changed twice, which again implies that another process has completed the operation. If the announcement and completion toggles do not match, the process tries to perform the operation itself, first copying the copy block, then performing all processes' outstanding operations on the new copy, and finally attempting to atomically replace the old copy block with the new copy. If the CAS succeeds, the process' operation is complete, and it can return the response value from the old copy block and reclaim the old block for its next operation. The swapping and reuse of copy blocks keeps the process' response valid until it initiates a new operation, eliminating the possibility of error through accessing an invalid block that has been reclaimed by the memory system and filled with random data.

Two changes to the copy block after a process' announcement guarantee that the process' operation has completed. Let PA denote the process making the announcement, and let P1 and P2 denote the processes that succeed in changing the copy block after a process' announcement. The following four events must happen in order: PA announces its operation, P1 changes the copy block pointer, P2 reads the copy block pointer, and P2 changes the copy block pointer. The ordering between the first two is assumed, and the ordering between the last two is defined by the operation. P2's success in changing the pointer implies that it reads the copy block pointer after P1 changes it. As P2 reads the copy block pointer after PA's announcement, it applies PA's operation to its private copy, implying by P2's success in changing the pointer that the operation is complete.

9.1.2 Hardware support

Most modern commercial SMP's employ a version of the MESI write-invalidate cache coherence protocol [PP84], allowing users only limited and typically coarse-grained control over coherence operations. Chapter 6 examined the cost of message-passing across such interconnects in detail, predicting the effect of current trends on future performance and suggesting a set of instruction set and coherence protocol extensions to better support message-passing. These extensions are designed to provide explicit control over cache coherence, allowing a programmer to override the implicit heuristics embodied by the MESI protocol.

Cheriton and Kutter [CK96] suggested another explicit approach to supporting message-passing in the context of the ParaDiGM multiprocessor, a directory-based CC-NUMA architecture. The work assumes a more traditional model of interprocess communication in which the operating system plays a fairly substantial role in moderating communication. Given the explicit support described by the paper, a sender writes a message into a local cache line, changing a flag within the line last. Writing to the flag signals the memory system to send the message, which moves the packet into one or more prelinked receiver cache lines. Messages are not guaranteed to arrive in the remote lines, nor are they guaranteed to be delivered before another message overwrites them. Those messages that do arrive generate a signal, which may or may not reach the destination process. Signals carry the virtual address of the delivered message, reducing the cost of handling the message. The operating system manages retransmission of messages and regeneration of signals.

Our approach is deterministic, allowing us to avoid operating system intervention. Our techniques also integrate more readily into existing commercial SMP's, as they require less extensive modifications to the memory hierarchy and avoid the need to support timeout and retransmission for shared memory messages in the operating system. A direct comparison of performance is questionable due to differences in the underlying technology. ParaDiGM uses 25 MHz Motorola 68040 processors—a CISC architecture, whereas our results are based on a recent 167 MHz RISC design. Including all optimizations, an RPC on the ParaDiGM system requires at least 47 microseconds, or 1,175 cycles. With none of the optimizations that we suggest, an equivalent RPC on our system requires 5.588 microseconds, or 933 cycles. Although the suggested hardware extension supports concurrent access, the work on ParaDiGM assumes allocation of separate communication memory for each sender, presumably due to the lengthy timeouts and retransmission necessary when senders contend. The operating system's signaling mechanism allows processes to avoid polling multiple message queues, but implies that messages interact asynchronously with a program. In such a model, handlers must either access data structures concurrently with the main program or queue messages for later handling.

The signal-on-write extension might serve as an alternative to burst support in processors that lack large register sets from which to issue a burst. Signal-on-write requires that the operating system and memory system manage the producer-consumer relationships between cache lines, effectively removing them from the control of the usual coherence protocol. Support for these relationships is more complex than that required to expose the ability to generate a coherence transaction with data, a capability already supported by all interconnects.

Other work in the literature examined implicit heuristics that complement our approach. Two studies [CF93, SBS93] considered extending a coherence protocol to dynamically identify migratory data, *i.e.*, data read and written by only one processor at any time but read and written by multiple processors over the lifetime of a program. Data manipulated within a critical section, for example, are often migratory. In the extended protocol, a processor that incurs a read miss on a migratory cache line receives exclusive access to that line with a single coherence transaction. A non-migratory cache line, or any cache line in the unmodified write-invalidate protocol, is first replicated into a shared state. The processor must then incur a second transaction to invalidate the line before modifying it. The utility of these extensions declines with longer cache lines, as the invalidation cost becomes small compared with the cost to move a line, and false sharing between migratory and read-shared data reduces the effectiveness of the dynamic identification [CF93]. At current cache line sizes, use of implicit strategies for identifying migratory data reduces coherence transactions and improves performance.

The implicit techniques do not fully address message-passing, however, as the semantics of message data differ from those of migratory data. A strict producer-consumer relationship governs message data: the producer always produces, and the consumer always consumes. In contrast, the notion of migratory data focuses on ownership: each new owner plays both roles, consuming some of the old data and producing some new data. Our explicit approach addresses message data, whereas the implicit approach addresses migratory data. Our techniques eliminate pollution of a sender's data cache with communication data, prevent the unnecessary transmission of old data, and push rather than pull the data across the interconnect. These benefits are not appropriate or useful for migratory data.

Our approach also benefits from its explicit nature. Data evicted from all caches must be reclassified by an implicit approach, whereas separate instructions serve to classify data in an explicit approach. Also, programmers that explicitly recognize message data also presumably act to eliminate false sharing with non-message data. Finally, an explicit approach requires fewer modifications to the coherency protocol, adding only one signal and no states. An implicit approach builds its knowledge in cache line states and requires that knowledge be transferred between caches with additional signals.

An explicit approach has no benefit unless a programmer makes use of it, however. Although message-passing is generally abstracted in library code rather than rewritten for each new application, the need for explicit recognition can be cumbersome. The two approaches are complementary, however, and can be used in tandem, allowing a programmer to obtain some of the performance benefit implicitly and to optimize as necessary to obtain the remaining performance potential.

Cooperative shared memory [HLRW92] (CSM) offers an explicit approach for identifying migratory and shared data with hints in the program. CSM transforms the notion of reader-writer locks into coherence terms, allowing a programmer to insert directives indicating an intent to read or an intent to modify a particular datum. Another directive announces the completion of the access. The memory system can use these directives to make decisions about migration and replication, and can hide latency through prefetching. The directives are only hints, however; the coherence protocol ensures correct behavior in the presence of conflicting directives. Like the implicit approaches to identifying migratory data, CSM focuses on ownership rather than a producer-consumer model.

Tempest [RLW94] addresses support for efficient shared memory on distributed memory machines through software control and hardware-supported fine-grained access. Tempest treats a portion of each machine's local memory as an L3 cache. Software control allows programmers to define arbitrary consistency semantics on a cache line basis using a complete set of explicit operations. A customized protocol described in [RLW94] is in many ways similar to our explicit protocol extensions, but addresses neither process migration nor pollution of a sender's cache with communication data. These issues are unimportant in the Tempest approach, which offers a large and fully associative L3 cache and does not support process migration.

9.2 Multi-Protocol Messages

The problems associated with integrating multiple communication protocols into a single abstraction have rarely been addressed. The Nexus portable programming system [FKT96] presents a model of communication very similar to our own, but focuses primarily on portability and on support for heterogeneity. It supports arbitrary sets of machines, processes (or contexts, in Nexus terminology) and threads. Nexus generally builds on top of existing communication layers, resulting in somewhat higher overheads than those obtained with active messages. The communication abstractions are similar to those of AM-II, but the style of communication is different. Like AM-II, Nexus has endpoints that define tables of handler routines, but Nexus does not require that communication obey a request-reply paradigm. This flexibility allows Nexus to use endpoint names, or startpoints, to initiate messages. A startpoint can be bound to multiple endpoints, allowing for multicast communication.

Nexus platforms support multiple communication protocols between a startpoint and an endpoint, thus Nexus has explored multi-protocol communication from a more general perspective than have we [FGKT97]. Although shared memory is mentioned in the work, numbers are provided only for more expensive underlying protocols, making a direct comparison impossible. The Nexus multi-protocol paper also notes the wide variance between polling costs for different protocols and presents data for fractional polling strategies. We developed a parametrized, adaptive strategy that allows the communication layer to tune itself dynamically to the underlying architecture.

Recent work on MPI-StarT [HH98] addresses multi-protocol communication in MPI on a Clump of Enterprise 5000's with custom network interface hardware.

9.3 Distributed Shared Memory

Distributed shared memory, also known as shared virtual memory, was suggested as an alternative to message-passing [LH86] long before researchers began to think about the Clump architecture. DSM provides the abstraction of a coherent, shared address space on a platform with physically distributed memories. In early work, the abstraction was completely transparent to a programmer, but later efforts to improve performance led to explicit coherence directives to support new models of data consistency [GLL+90, KCZ92, IDFL96, ISL96a, SB97]. Researcher have now begun to extend DSM notions to Clumps architectures [SDH+97, SB98, SGA98] to take advantage of tight coupling within the SMP's. Recent DSM projects include SVM [ISL96b, JSS97], MGS [YKA96, Yeu98], Brazos [SB97, SB98], Cashmere [KHS+97, SDH+97], Shasta [SGT96, SGA98], and many others.

DSM presents an alternative approach to providing a uniform interface for programming a Clump. In this respect, DSM and this thesis both seek to optimize common techniques in one medium to allow use of those techniques in both. The research community has long debated the relative merits of the shared memory and message-passing models, to the point that many of the arguments have been reduced to zealous religious beliefs rather than objectively substantive claims. Dogmatically speaking, shared memory is easier to program and message-passing delivers better performance. Although at times reluctant to admit it publicly, most researchers recognize that each approach has advantages and disadvantages, and that each view proves more natural and effective than the other for interesting classes of applications. That said, we now explain why we chose to investigate a uniform message-passing model rather than DSM.

A uniform message-passing interface forces a programmer to address the motion of data explicitly and to recognize that moving data is a costly operation. Programmers accustomed to dealing with high latencies will not have difficulty handling lower latencies for some messages. In contrast, a coherent shared memory abstraction manages the motion of data implicitly, in theory allowing a programmer to ignore such issues. Even across the low-latency memory interconnects within SMP's, supporting such an abstraction transparently, *i.e.*, with sequential consistency, incurs very high overheads. Generalizing shared memory to work efficiently with higher latencies is a very hard problem. DSM attempts to tackle this problem, providing coherent shared memory across a communication network, an interconnect with much higher latency. In most existing architectures, the network also presents higher overhead and lower bandwidth than the memory interconnect, but increased latency is the only fundamental difference. An artifact of implementation via the virtual memory protection bits helps to amortize the resulting high cost of exchanging "lines" in the global cache by increasing the line size to that of a virtual memory page. However, false sharing and data fragmentation due to the larger line sizes can significantly impact performance, leading several groups to investigate fine-grained DSM [RLW94, SGT96, KHS+97].

Irrespective of amortization, however, obtaining reasonable performance with high latencies requires that DSM approaches adopt weaker models of data consistency, which in turn require a level of explicit control by a programmer. Typically, programmers insert coherence directives requesting that a local copy of data be updated or that recent changes to data be made globally visible. A few systems treat these directives as hints [HLRW92], using them to improve performance without risking program correctness. Incorrect hints can adversely affect performance, however, by causing memory to thrash between processors. Most systems thus treat the directives as guarantees by the programmer that correct behavior admits inconsistencies in the data [ISL96b]. We claim, and shared memory enthusiasts deny, that once a programmer must explicitly and correctly control data motion within an application, the task of writing that application is at least as difficult with a shared memory model as with a message-passing model. Explicit control renders the shared memory abstraction little more than a convenient aliasing mechanism between message buffers; the added benefit is the ability to dynamically determine and send only the modifications. We view explicit data management within DSM systems as a convergence with the BSP message-passing model [Val90], a phase-structured approach in which messages (shared data) remain outstanding (inconsistent) until the end of a phase (synchronization directive). Like DSM, BSP dynamically tracks and sends only the changes, *i.e.*, the messages sent during a phase.

The central reason for our favoring message-passing over shared memory as a uniform interface, however, is the difficulty of obtaining good performance through shared memory. Programming a flat shared memory model for performance is not as easy as it sounds: although all memory is equally close, the cache is much closer, and ensuring that an application spends little time waiting for data to move between caches can be challenging. Shared data must be laid out very carefully, typically using hard-won experience, to minimize coherence transactions. Algorithmic changes are not uncommon. Consider, for example, that the shared memory protocol developed in this thesis is little more than a data structure and supporting code in a shared memory program, then imagine expending an equivalent effort for every data structure. The shared virtual memory literature [ISL96b] is a good source of information on the necessary performance optimizations, as building shared memory across relatively slow interconnects forces researchers to pay careful attention to performance issues. For DSM systems, the relative cost of the protocols makes data layout even harder. Algorithmic changes are common.

The optimization process is even more difficult. A programmer optimizing for a DSM must not only understand the issues for the program, but must also be fairly knowledgeable about the heuristics and decision-process employed by the DSM runtime in order to invert those decisions when tuning a program for performance. Furthermore, false sharing and data fragmentation tie shared memory performance to the artificial granularity of access used to support the coherence mechanisms. In terms of optimization, this dependency couples otherwise unrelated portions of a program. Redistributing data to optimize one procedure may degrade performance in another portion of the code that accesses the same data in a different fashion. Although such coupling is relatively unimportant for benchmarks with only a few thousand lines of code, it can easily dominate the complexity of the optimization process in applications with hundreds of thousands or millions of lines of code. The explicit data management necessary with message-passing is indeed harder to construct than the arbitrary schemes possible with shared memory, but such an investment is more than worthwhile if in return a programmer gains the ability to perform local optimizations using a simple cost model.

Returning to a more objective viewpoint, we might consider integrating the shared memory and message-passing models. Combining the better aspects of the two may lead to an effective programming model, but merely making both available is not adequate. Support for both models is clearly useful in the hardware, and many architectural research efforts address simultaneous support for both message-passing and shared memory, including Alewife [Kub98], FLASH [HGDG94], and StarT-Voyager [ACR$^+$98]. Tempest [RLW94] provides support for both models on a distributed memory machine with a small amount of additional hardware. Both message-passing and shared memory are fairly difficult to use in isolation, however, and using them in tandem or incrementally switching subsets of data between them is not a simple task. The hierarchical nature of future Clump architectures further complicates matters by introducing dynamic tradeoffs. One model may deliver adequate performance for processors arranged in one hierarchy but not in a second hierarchy.

Compiler and runtime support is necessary to handle the dynamic aspects, but pushing the issues into these systems begs the question of what abstractions a programmer actually uses to program a Clump. A compiler and runtime system capable of applying a complex global analysis of source code based on a simple abstraction to obtain a high fraction of available performance may not be able to maintain that level of performance as the levels of the data hierarchy shift underneath it. Equally hopeless in such a case is the notion that a programmer will unravel the transformation to optimize performance.

The programming model for these systems must thus embody both a reasonable level of simplicity to allow a programmer to write correct code and a reasonable level of clarity to allow the programmer to optimize that code. Neither message-passing nor shared memory is particularly satisfying in this regard.

9.4 Programming Model

A variety of programming models have been suggested for Clumps. Some extend a combination of shared memory and message-passing with library routines for collective communication, while others propose to extend models developed on SMP's or on NOW's to Clumps. Several other models also look promising.

9.4.1 Clumps models

The p4 programming system [BL94] was probably one of the first systems to recognize Clumps as a platform. It provides mechanisms start multiple threads on one or more machines and to communicate between such threads using either message-passing or shared memory constructs. The programmer must explicitly select the appropriate library call. The library also provides a number of useful reduction operations.

SIMPLE [BJ97] provides functionality similar to p4, but extends the library with broadcast operations and a variety of tuned, many-processor communication methods. SIMPLE also attempts to lighten the programmer's burden by offering functions that involve all processors, all processors in an SMP, one processor in each SMP, and so forth.

A paper by Fink and Baden [FB] attacks the problem of load balance in phase-structured algorithms by rebalancing computation and communication for a regular problem within an SMP. Given a 2D domain partitioned in one dimension between SMP's in a Clump, the paper calculates a non-uniform partitioning of the domain within each SMP such that the time spent in a phase is roughly equal for each processor. The analysis gives processors on boundaries less computation to balance the cost of communication.

KeLP, by the same authors, seeks to simplify the process of application development. Recent extensions to KeLP [FB97] add new functionality to support applications on Clumps. With KeLP, a programmer expresses data decomposition and motion in a block-structured style. The runtime system then employs inspector-executor analysis to overlap communication with computation. No global barriers are used; interprocessor synchronization occurs only through communication dependencies.

9.4.2 SMP models

Bulk-synchronous programming, or BSP [Val90], is a phase-structured approach in which messages sent during a phase do not arrive until the end of that phase, at the next global synchronization. BSP may be better than less restrictive phase-structured approaches because it allows a runtime system to dynamically schedule communication onto a virtual NIC. As illustrated by our model in Chapter 8, communication scheduling has a large potential impact on performance. The difficulty lies in rewriting applications in a way that exposes scheduling opportunities; a phase in which processors solely communicate, for example, cannot be improved by scheduling.

Autoscheduling analyzes parallel loop dependencies to generate a hierarchical task graph, allowing the runtime system to dynamically schedule loop iterations across processors [MP94]. This dynamic approach to load balancing aspect addresses the problem of the uneven benefits of fast communication by rebalancing the workload based on the platform architecture. Autoscheduling has been demonstrated on DSM platforms as well.

Cilk [FLR98] dynamically schedules compiler threads generated through fork-join parallelism onto an SMP, using work-stealing to distribute the threads. Each process maintains a set of ready threads. A fork operation allows several new threads to execute in parallel, and these threads are placed into the local set to be executed one at a time. When a process runs out of threads, it steals work (threads) from another process, effectively balancing the load. With efficient implementation support for compiler threads [Gol97] and effective thread scheduling [NB98], an extension of Cilk to a Clump architecture might address the problems of correlated communication and uneven fast communication benefits without introducing significant overhead.

9.5 Performance Evaluation

We presented a suite of benchmarks for measuring the performance of shared memory message-passing. Our approach enables a more complete perspective on the problem than do most approaches found in the literature, examining performance across the possible range of contention and highlighting a variety of applications. In this section, we focus on developing a methodology for measuring the performance of applications under multiprogramming, a metric that has remained elusive because of the complexity of the general problem and the natural inconsistency of independent attempts at simplification. We compare two previous multiprogramming approaches with our own methodology and suggest that a more detailed study is necessary to capture performance on a time-shared system.

Wisniewski *et al.* [WKS95] and Michael and Scott [MS97] simulate multiprogramming to evaluate preemption-safe locking and to compare concurrent algorithms by using one processor within an SMP as a scheduler and binding one process to each other processor. The scheduler periodically signals the application processes to stop and start executing their local code. While stopped, the processors wait idly for the next start signal. Blumofe and Papadopoulos [BP98] measure multiprogramming in the context of Cilk by allowing the number of user-level threads in a program to exceed the number of physical processors. On the experimental platform used in [BP98], a Sun Enterprise server running Solaris, all thread scheduling in such an environment occurs at user-level, eliminating the need for context switches. Both approaches outlined here allow for fractional multiprogramming, an aspect that our own results lacked due to time constraints on the availability of the machines. However, these methodologies produce overly optimistic results, as they neglect several detrimental effects, including the motion of data between caches, context switches between jobs, and variable levels of cooperation from competing jobs.

As outlined in Chapter 6, yielding a processor potentially incurs overhead of three types: operating system scheduler invocation, including context switches; cache pollution by an intervening process; and data transfer between caches. The approach taken in the concurrent algorithm work eliminates these costs, allowing performance to remain high as the level of multiprogramming rises. In fact, in the absence of cache effects, reduced contention under multiprogramming leads to improved performance for several of the algorithms studied. The Cilk methodology avoids the first cost by executing only a single process on the machine, and reduces the effect of the second by executing the same code on all processors. We do not mean to imply that multiprogramming need be measured using several processes within a parallel job, only that competing jobs should not be in the same process.

The multiprogramming results in our work demonstrate the effect of variable levels of cooperation: other parallel jobs try to coschedule themselves implicitly, and are likely to yield processors unless their sibling processes are also running; sequential processes do not yield, and reduce the performance of parallel jobs that must compete with them. The concurrent algorithm work allows a simulated intervening process a randomized time slice with a constant mean, corresponding more closely to the effect of competing with sequential processes. The Cilk work competes with its own threads, which yield their processors when they run out of work, corresponding to competition with parallel jobs.

A measure of multiprogramming performance should reflect the performance available from a time-shared system. Without a methodology that captures the effect of resource interactions with competing jobs and operating system overheads, however, multiprogramming results are hard to interpret in this manner. Our multiprogramming methodology attempts to obtain a fair approximation of time-shared performance by evaluating application in competition with a range of parallel and sequential jobs. A few simplifying assumptions are necessary to limit the amount of data. A simple sequential program that generates memory traffic and pollutes the caches serves as the model of a sequential process. Unrelated copies of a parallel application serve as the model of a parallel job. Extending our methodology with fractional multiprogramming is also potentially interesting.

9.6 Other Issues

We conclude with two issues related less directly to the work described by this thesis: integrating message-passing communication more tightly into the memory hierarchy and routing network traffic through a proxy process.

9.6.1 Integrating communication

A variety of work focuses on integrating network communication devices more tightly into the memory hierarchy, placing them directly on the memory interconnect, integrating them into the processor board, or even exposing them as control registers in the processor. Mukherjee and Hill [MH97] surveys the literature on the advantages of such integration. Joerg and Henry [JH92] describes a network interface architecture and evaluates its performance at several levels of integration, including register-mapped and on- and off-chip cache implementations. The Alewife machine uses a modified processor and a custom "communications and memory management unit" to provide efficient, user-level message-passing and coherent shared memory support [Kub98]. The work demonstrates the performance potential of systems built from the ground up to support efficient communication. StarT-Voyager [ACR+98] also places a custom network interface device on the memory interconnect, but is designed to support multiple processors. Four types of user-level messages offer a variety of tradeoffs between length and startup cost. A programmable service processor and control message capability between the service processor and application processors allows StarT-Voyager to readily support DSM protocols as well. As with the AM-II network layer, StarT-Voyager virtualizes communication queue resources, paging them dynamically between the network interface memory and the host.

The empirical work in this thesis focuses on implementations based on current commercial systems. Tighter levels of integration in the swiftly moving computer industry require higher levels of trust between companies developing processors and individual machines and companies developing network interfaces, but we are optimistic about the possibilities demonstrated by the work mentioned above. For multi-protocol communication, a more balanced relationship between protocol overheads might eliminate the need for an adaptive polling strategy, although variations in latency will maintain the value of optimizing using locality information.

As with most modern SMP's, our experimental platform executes synchronization primitives at the level of the L2 cache. Several architectures [Rud81, KDL$^+$93] with uncacheable portions of memory support atomic synchronization primitives at the memory itself. Processors send messages to the memory to perform the primitives. For a general class of read-modify-write operations, such messages can be combined in the network between the processors and the memory to allow the architecture to scale to very large numbers of processors [KRS86]. The latency across these networks represents a major drawback, but approaches that explicitly recognize the operations that generate such messages might be able to hide some or all of it.

9.6.2 Message proxies

A significant body of Clumps-related literature addresses only platforms with a single network device per SMP, perhaps as a simplifying assumption. Our results and model suggest that the view of multiple network interface devices as a single virtual device can improve performance on a Clump, but retaining message overheads close to those available in hardware requires that the virtualization be done in a distributed fashion. An alternative approach directs network traffic through a local proxy, often running on a dedicated processor.

An interesting study by Lim *et al.* [LHPS97] investigates this approach as a means to provide multiple users with protected access to a single network resource and provides a breakdown of costs in the proxy approach. Our system sidesteps the question of protected access by taking advantage of an SMP's virtual memory system to grant direct access to a subset of network resources. An intelligent NIC plays an essential role in our approach. Falsafi and Hill [FW97] provides a more thorough survey of proxy-based approaches and compares their performance with systems in which all processors communicate through the network.

Chapter 10

Conclusions

Lest this thesis become, like the world, too much with us, we now retreat to a pleasant lea and survey our work from a distance. This chapter expresses our view of the thesis against a background of mercurial progress, relating both its defenses and its weaknesses against the lone and level sands of Time. In concrete terms, we outline our philosophy.

10.1 Hierarchical Systems

Future architectures will present complex hierarchies of data to maintain the rate of growth in processor power, requiring that applications carefully manage the motion of data through the system. Effectively addressing such a platform requires a combination of automatic control and application-specific knowledge. Automatic control handles the bulk of the work, applying heuristics grounded in general principles and system parameters. Application-specific knowledge allows a program to make more effective use of the system, integrating the application's needs with the capabilities of the architecture.

An ideal abstraction gives a programmer ready access to both techniques by providing both a straightforward initial approach and a clear optimization path. The initial approach relies on the implicit heuristics, managed by the compiler and runtime and supported by the hardware, to obtain a reasonable level of performance. The optimization path allows a programmer to explicitly override the heuristics, applying application-specific knowledge to improve performance. The means of explicit control are incremental and require only local decisions, implying that the heuristics do not obscure the relationship between a program and its performance.

A uniform interface is a good approach to developing such an abstraction. This thesis describes a uniform message-passing interface that closely couples performance to that of the underlying hardware, tackling the requirements outlined above for interprocess communication. In a broader sense, it demonstrates the value of using such an interface. Applications distributed across processors in a hierarchical system must at some level address the variance in their ability to share data. Clearly no programmer wants to undertake the wide-ranging and painstaking level of effort required to develop the individual protocols and integration support described here for each new application. Our multi-protocol communication layer demonstrates that runtime software can provide the automatic control necessary to address the Clump hierarchy in the common case, thereby reducing the necessary effort for every application that builds on it. A programmer need consider only a simple cost model in an initial approach to an application: invoking a remote function is expensive, and should be overlapped with computation. Using the simple cost model, a programmer can ignore the hierarchy and obtain reasonable performance. Improving performance requires that a programmer refine the cost model to recognize that functions invoked on another SMP take longer than functions invoked within the same SMP. Locality information is readily available, and performance depends primarily on the algorithms employed in communication, allowing a programmer to handle hierarchical aspects selectively and incrementally.

Two key components enable the success of the multi-protocol approach: the lock-free concurrent queue algorithm and the adaptive polling strategy. Our original shared memory protocol used Posix mutexes to protect access to the queue data. With little prior shared memory experience, Posix mutexes seemed the concurrency mechanism of choice; why else are they the standard? Compared with the more heavyweight network protocol, microbenchmark timings with Posix mutexes looked very fast, painting a rosy picture for the advantages of a Clump over a NOW. Selling SMP's is not our business, of course, but we had hypothesized improved performance and felt that we might soon confirm the hypothesis, possibly even in a way that allowed quantification. Success was not immediate, however. Applications generated vast and widely varying numbers of context switches with the original protocol, leading one member of our project to joke that SMP's made excellent random number generators. The search for an algorithm that delivered good performance at all levels on both dedicated and multiprogrammed machines eventually led to the lock-free algorithm.

Lock-free algorithms are in general a good way to approach the problem of designing a concurrent data structure. The literature perhaps overemphasizes the importance of contention. High levels of contention for individual instances of a data structure do not generally occur in well-written applications. Most programmers know that the average arrival rate of processes at a data structure should not exceed the service rate of the operations. Interactions with the operating system scheduler depend on the lengths of critical sections, which are generally short, but the relative length of a time slice makes such interactions important. Time slices are growing to amortize larger total cache reload costs; faster clock times make critical sections shorter. As demonstrated by the performance of Posix mutexes on a dedicated machine, informing the operation system of every synchronized access is also costly. A lock-free algorithm offers most of the advantages of the more costly non-blocking property, avoiding both forms of operating system interactions. They trade a slight possibility of blocking for reduced overhead in the common case.

The adaptive polling strategy enables a single binary built on our layer to perform well on a wide range of Clump architectures, including the extremes of a NOW and a single SMP. This component allows us to compile applications for a generic Clump rather than a specific configuration. The need for an adaptive polling strategy became apparent when we put the two protocols together and found that our microbenchmark and application performance on an SMP had dropped through the floor. We knew that we had to poll the network, and we knew that reading from NIC memory is costly, but we did not understand the implications until the numbers were in front of us. Fractional polling strategies seemed like a fairly natural solution, and did provide adequate levels of performance, but we felt that we should be able to do better. The first version [LMC97] employed a simple adaptive heuristic that tracked only network traffic. The adaptive bounds were selected by hand from an exploration of application performance across the space of possible bounds. After some thought, we developed a deeper understanding of both the approach and the important constraints, allowing us to derive the more general approach described in this thesis. The new strategy maintains the traffic estimates with very little overhead, and the parameters allowed us to fully automate the process of collecting performance data for tuning. The resulting strategy is much more effective, obtaining superior levels of performance on all forms of Clumps.

Adaptive polling is as much an example of good engineering as it is one of research. It corrects no errors, adds no functionality, and explains no mystery. It does provide a way to address a performance problem in a very efficient and flexible manner, in a regime in which a complex solution can easily incur as much overhead as the problem. Using simple measures of past success as the basis for future decisions is a technique that might be applied successfully to a wide range of problems, from selecting amongst several sources of information to deciding on which disk to store a new block of data.

10.2 Programming Models

Efficient support for passing messages is an excellent building block for a myriad of higher-level interfaces. In spite of our preference for message-passing over shared memory, we recognize that passing messages is no panacea. Those of us who develop applications as benchmarks secretly care more about getting fast results than about getting correct ones.[1] Message-passing is great for speed and awful for correctness. The benchmark applications used in our measurements rarely pass messages, however. Split-C provides a non-coherent shared address space abstraction and explicit control for updates and visibility. The aliasing mechanism provided by explicitly controlled shared memory really is very convenient. Unfortunately, it does little to eliminate the potential for incorrect behavior.

As does almost everyone writing message-passing or shared memory codes, the authors of our applications simplified their tasks by applying a phase structure. Phase-structured programs are much easier to contemplate, much easier to design, and much easier to understand. The phases provide a hierarchical framework that greatly reduces the difficulty of informally verifying correctness. The phases themselves are essentially a sequential program, and each phase is a small parallel program. People can manage small parallel programs; they have a lot of trouble with large ones. The structure makes writing applications easier.

On a Clump, that same structure hurts application performance. Forcing processes to wait for other processes to catch up prevents an application from reaping the full benefits of tight coupling within SMP's and pooling of network devices. Correlating processes' communication demands within each phase eliminates any benefit of pooling those demands.

[1] People who tell you that they enjoy writing irregular message-passing codes are pathological liars, bad programmers, or computer scientists. A fourth possibility is one of those people who enjoys writing proofs with hundreds of cases.

Dynamic load balancing may offer a solution, but brings problems of its own, including nondeterministic scheduling and the possibility of thrashing the memory system by ignoring issues of cache affinity. Dynamic load balancing also addresses a third performance issue. The variability of a workload distributed between processors can be reduced by dynamically sharing that work, but moving a task between address spaces (or across the network in a DSM) may take more time than executing the task locally. In a Clump, dynamic load balancing reduces the variability by sharing tasks within each SMP.

More than a programming model is required to properly address Clumps. We need a programming environment: a language, a compiler, a runtime, and strategies for debugging. An object-based approach that provides fork-join parallelism with dynamic scheduling, allows a programmer to reason about data affinity, and admits effective approaches to debugging sounds attractive provided that all of these things can be done without quadrupling execution time through self-analysis in the runtime.

10.3 Importance of Clusters

Looking at the big picture, one wonders whether cluster computing will have any significance in the future. Joy's law of processor performance scaling implies a similar reduction in the number of processors necessary for a given application. In fact, the reduction is more rapid than the improvement, as parallelism is almost never efficient. Decoding a compressed video clip in real time may have required a cluster of several workstations several years ago, but today a low-end laptop can decode several while recalculating a spreadsheet, and in a few years only antique watches will lack video support. Pessimists might further point out that the problem already exists today: universities and laboratories with large cluster systems are having trouble finding enough applications to keep the machines busy. What can we possibly do with all of the cycles we will have in the future?

Few people in the history of computers have been able to answer that question without the aid of retrospect. When cycles becomes available, people find ways to use them. Clusters are complicated, and making them truly available requires an effective, general-purpose software infrastructure. Large research clusters are currently underutilized because that infrastructure still needs work. A great deal of progress has been made, however, much of it based upon work with the large clusters, and this thesis represents another contribution towards the end goal. To lend weight to this argument, consider the success of clusters designed to run only very specific applications. The largest internet service providers use clusters to manage connectivity and to drive their virtual environments and entertainment divisions. The entertainment industry makes heavy use of clusters to produce special effects for films, sometimes even eliminating the actors completely. Any company that runs critical business applications on a Sun Enterprise 10000, an IBM S/390 Parallel Sysplex, an IBM SP-2, an SGI Challenge Array, an SGI Origin, or one of many other commercial machines, makes use of cluster software technology in some form. Cast in a more optimistic light, the question remains nearly the same: what can we do with all of the cycles we will have in the future?

The appearance of low-end SMP's may fuel the effort to develop an effective infrastructure by placing parallelism in the path of the software industry. In the past, multiple processors were rarely available to a single application, or were coupled only through high-latency, low-bandwidth interconnects and separate security domains that rendered application-level parallelism difficult to obtain. Once low-end SMP's find their way to the desktop, application vendors will be forced to exploit all of the processors to remain competitive with other vendors. This same phenomenon occurred many years ago with databases, and today many commercial databases run on either SMP's or clusters, using all of the processing power available and driving all of the disks. Some vendors might survive for a while by improving sequential software, but finding coarse-grain parallelism in the computationally intensive portions of a program is usually easier than obtaining the same speedup from algorithmic improvements or cache-tuning. Expressing and debugging that parallelism efficiently, particularly as it becomes more fine-grained, will require better abstractions.

This shift of focus may also force low-end SMP's to mimic their more expensive counterparts. The design of the high-end server platforms has been driven to a large extent by the needs of I/O-intensive applications, and the effect of this market on their architecture becomes quite clear when one compares them with low-end systems. Current low-end machines are little more than personal computers with an extra processor stuck on the memory interconnect. Neither the interconnect nor the memory provides enough bandwidth for applications with large working sets, and the I/O subsystem is typically restricted to a single bus. Some point out that the I/O bus bandwidth in these systems is matched to the memory bus bandwidth. In the same vein, one might consider slowing down a processor to match the capabilities of an attached disk. The technology clearly exists to provide more effective coupling between processors and between boxes, but cost defers only to widespread demand in architectures designed for the home and small business markets.

Clusters offer a great deal of potential for scalability and availability, and Clumps are a natural extension of the data hierarchy within a cluster. The question to what extent that potential can be harnessed to our benefit remains open.

10.4 Etymology

In closing, I offer a bit of etymology. The term Clump began as an attempt at humor. I had been thinking about clusters of SMP's and developing a few analytic notions on performance. David Culler asked me to write up my thoughts to include in an initial proposal that he and Horst Simon (of NERSC) were writing to Sun Microsystems, and I jokingly labeled the platform "CLOMP's" for CLusters Of MultiProcessors. The name then diverged: David named our platform and project "Clumps," while the project at NERSC became "COMP's." I take credit for both terms, of course, although I've never verified either claim. A few months into the project, several of the members began to refer to the individual machines as "Lumps." Fortunately, I was able to deliver the *coup de grâce* to that term by publishing the project's first paper without it. I hope.

Bibliography

[ABP98] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures*, pages 119–29, Puerto Vallarta, Mexico, June 1998.

[ACPtNT95] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[ACR+98] B. S. Ang, D. Chiou, D. L. Rosenband, M. Ehrlich, L. Rudolph, and Arvind. StarT-Voyager: A Flexible Platform for Exploring Scalable SMP Issues. In *Proceedings of SC98: High Performance Networking and Computing*, Orlando, Florida, November 1998.

[ADCM98] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the SIGMETRICS'98/PERFORMANCE'98 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 233–43, Madison, Wisconsin, June 1998.

[ADV+95] R. H. Arpaci, A. C. Dusseau, A. H. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–78, Ottawa, Canada, May 1995.

[AISS95] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer towards a Realistic Model for Parallel Computation. In *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, California, July 1995.

[And90] T. E. Anderson. The Performance of a Spin Lock Alternative for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[AP97] E. Anderson and D. A. Patterson. Extensible, Scalable Monitoring for Clusters of Computers. In *Proceedings of the 11th Systems Administration Conference (LISA'97)*, pages 9–16, San Diego, California, October 1997.

[BA97] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring '97*, pages 81–6, San Jose, California, February 1997.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet—A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–38, February 1995.

[BCL+95] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*, pages 42–53, Santa Barbara, California, July 1995.

[BDF+95] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[BJ97] D. A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMP's). Preliminary version available via http://www.umiacs.umd.edu/research/EXPAR, May 1997.

[BK94] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 858–67, Cancun, Mexico, April 1994.

[BL94] R. Butler and E. Lusk. Monitors, Message, and Clusters: the p4 Parallel Programming System. *Parallel Computing*, 20(4):547–64, April 1994.

[BLA+94] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual-Memory-Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–53, Chicago, Illinois, April 1994.

[BLM91] G. Blelloch, C. Leiserson, and B. Maggs. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, South Carolina, July 1991.

[BP98] R. D. Blumofe and D. Papadopoulos. The Performance of Work Stealing in Multiprogrammed Environments. Technical Report CS-TR-98-13, University of Texas at Austin, May 1998.

[Byr93] G. T. Byrd. Models of Communication Latency in Shared Memory Multiprocessors. Technical Report CSL-TR-93-596, Stanford University, December 1993.

[CC97] K. Connelly and A. A. Chien. FM-QoS: Real-time Communication using Self-synchronizing Schedules. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, California, November 1997.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 1993*, pages 262–73, Portland, Oregon, November 1993.

[CDMS93] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. *Fast Parallel Sorting under LogP: from Theory to Practice*, chapter 4, pages 71–98. John Wiley & Sons, 1993.

[CF93] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98–108, San Diego, California, May 1993.

[CK96] D. R. Cheriton and R. A. Kutter. Optimized Memory-Based Messaging: Leveraging the Memory System for High-Performance Communication. *Computing Systems*, 9(3):179–215, 1996.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

[CLMY96] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[CMC98] B. N. Chun, A. M. Mainwaring, and D. E. Culler. Virtual Network Transport Protocols for Myrinet. *IEEE Micro*, 18(1):53–63, February 1998.

[CPWG97] A. Charlesworth, A. Phelps, R. Williams, and G. Gilbert. Gigaplane-XB: Extending the Ultra Enterprise Family. In *Proceedings of Hot Interconnects V*, pages 97–112, Stanford, California, August 1997.

[CRD+95] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 12–25, Copper Mountain Resort, Colorado, December 1995.

[CSwAG98] D. E. Culler and J. P. Singh with A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1998.

[DO87] F. Douglis and J. K. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987.

[FB] S. J. Fink and S. B. Baden. Partitioning of Finite Difference Methods Running on SMP Clusters. Available at http://www-cse.ucsd.edu/users/baden.

[FB97] S. J. Fink and S. B. Baden. Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters. In Y. Ishikawa *et al.*, editor, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 1–8. Springer-Verlag, 1997.

[FGKT97] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing Multiple Communication Methods in High-Performance Networked Computing Systems. In *Journal of Parallel and Distributed Computing*, volume 40, pages 35–48, January 1997.

[FKT96] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, August 1996.

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of Programming Language Design and Implementation*, pages 212–23, Montreal, Quebec, Canada, June 1998.

[FW97] B. Falsafi and D. A. Wood. Scheduling Communication on an SMP Node Parallel Machine. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 128–38, San Antonio, Texas, February 1997.

[GC96] M. Greenwald and D. Cheriton. The Synergy between Non-Blocking Synchronization and Operating System Structure. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 123–36, Seattle, Washington, October 1996.

[GC98] K. Ghosh and A. J. Christie. Communication Across Fault-Containment Firewalls on the SGI Prigin. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 277–87, Las Vegas, Nevada, February 1998.

[GK97] R. B. Gillett and R. Kaufmann. Using the Memory Channel Network. *IEEE Micro*, 17(1):19–25, February 1997.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[GLR83] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–89, April 1983.

[Gol97] S. C. Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California at Berkeley, June 1997.

[GPR+98] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software—Practice and Experience*, 28(9):929–61, July 1998.

[Her88] M. P. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the 7th Symposium on Principles of Distributed Computing*, pages 276–290, Toronto, Ontario, Canada, August 1988.

[Her93] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–70, November 1993.

[HGDG94] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, San Jose, California, November 1994.

[HH98] P. J. Husbands and J. C. Hoe. MPI-StarT: Delivering Network Performance to Numerical Applications. In *Proceedings of SC98: High Performance Networking and Computing*, Orlando, Florida, November 1998.

[HLRW92] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–73, Boston, Massachusetts, October 1992.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–92, July 1990.

[IBM98a] IBM S/390: The Defining Standard of Enterprise Computing, Today and Tomorrow, International Business Machines Corporation. Document #GF22-5043-00. Available at http://www.s390.ibm.com/marketing/gf225043.html, May 1998.

[IBM98b] S/390 Parallel Enterprise Server and OS/390 Reference Guide, International Business Machines Corporation, May 1998.

[IDFL96] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 14–25, San Jose, California, February 1996.

[ISL96a] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures*, pages 277–87, Padua, Italy, June 1996.

[ISL96b] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 122–33, Philadelphia, Pennsylvania, May 1996.

[JH92] C. F. Joerg and D. S. Henry. A Tightly Coupled Processor-Network Interface. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–22, Boston, Massachusetts, October 1992.

[JSS97] D. Jiang, H. Shan, and J. P. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proceedings of the 6th Symposium on Principles and Practice of Parallel Programming*, pages 217–29, Las Vegas, Nevada, June 1997.

[KC95] V. Karamcheti and A. A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 298–307, Santa Margherita Ligure, Italy, June 1995.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Queensland, Australia, May 1992.

[KDL+93] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner. The Cedar System and an Initial Performance Study. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 213–23, San Diego, California, May 1993.

[Kel79] F. P. Kelly. *Reversability and Stochastic Networks*. John Wiley & Sons Ltd., 1979.

[KHS+97] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 157–69, Denver, Colorado, June 1997.

[KLCY97] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected Components on Distributed Memory Machines. In Sandeep Bhatt, editor, *Parallel Algorithms: Third DIMACS Implementation Challenge*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. 1997.

[KRS86] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient Synchronization on Multiprocessors with Shared Memory. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pages 218–28, Calgary, Alberta, Canada, August 1986.

[Kub98] J. D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, February 1998.

[KY96] A. Krishnamurthy and K. Yelick. Analyses and Optimizations for Shared Address Space Programs. *Journal of Parallel and Distributed Computing*, 38(2):130–44, November 1996.

[LC95] L. T. Liu and D. E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Proceedings of the Intel Supercomputer Users Group Conference*, June 1995.

[LC96] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308–17, Philadelphia, Pennsylvania, May 1996.

[LH86] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pages 229–39, Calgary, Alberta, Canada, August 1986.

[LHPS97] B.-H. Lim, P. Heidelberger, P. Pattnaik, and M. Snir. Message Proxies for Efficient, Protected Communication on SMP Clusters. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 116–27, San Antonio, Texas, February 1997.

[LKC95] S. S. Lumetta, A. Krishnamurthy, and D. E. Culler. Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines. In *Proceedings of Supercomputing 1995*, San Diego, California, December 1995.

[LL92] M. J. Litzkow and M. Livny. Making Workstations a Friendly Environment for Batch Jobs. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 62–7, Key Biscayne, Florida, April 1992.

[LL97] J. Laudon and D. E. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–51, Denver, Colorado, June 1997.

[LLG+90] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–59, Seattle, Washington, May 1990.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–11, San Jose, California, June 1988.

[LMC97] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, California, November 1997.

[Mad92] N. K. Madsen. Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Non-Orthogonal Unstructured Grids. Technical Report 92.04, NASA RIACS, February 1992.

[Mar94] R. P. Martin. HPAM: an Active Message Layer for a Network of HP Workstations. In *Proceedings of Hot Interconnects II*, pages 40–58, August 1994.

[MC96] A. M. Mainwaring and D. E. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report #CSD-96-918, U. C. Berkeley, October 1996.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[MCSW97] A. M. Mainwaring, B. N. Chun, S. Schleimer, and D. Wilkerson. System Area Network Mapping. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures*, pages 116–26, Newport, Rhode Island, May 1997.

[MH97] S. S. Mukherjee and M. D. Hill. A Case for Making Network Interfaces Less Peripheral. In *Proceedings of Hot Interconnects V*, pages 21–6, Stanford, California, August 1997.

[MP91] H. Massalin and C. Pu. A Lock-free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, June 1991.

[MP94] J. E. Moreira and C. D. Polychronopoulos. Autoscheduling in a Distributed Shared-Memory Environment. In K. Pingali *et al.*, editor, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 453–69. Springer-Verlag, 1994.

[MPI97] MPI-2: Extensions to the Message-Passing Interface. Available from the Message Passing Interface Forum at http://www.mpi-forum.org/, July 1997.

[MS96] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Symposium on Principles of Distributed Computing*, pages 267–75, Philadelphia, Pennsylvania, May 1996.

[MS97] M. M. Michael and M. L. Scott. Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 267–73, Geneva, Switzerland, April 1997.

[NB98] G. Narlikar and G. Blelloch. Pthreads for Dynamic and Irregular Parallelism. In *Proceedings of SC98: High Performance Networking and Computing*, Orlando, Florida, November 1998.

[NMCB97] J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen. S/390 Cluster Technology: Parallel Sysplex. *IBM Systems Journal*, 36(2):172–202, 1997.

[Ous82] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, Miami/Fort Lauderdale, Florida, October 1982.

[PAC+97] D. A. Patterson, T. E. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM: IRAM. *IEEE Micro*, 17(2):29–38, April 1997.

[Pfi98] G. F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing.* Prentice Hall, 2nd edition, 1998.

[PH98] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, 2nd edition, 1998.

[PLC95] S. Pakin, M. Lauria, and A. A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing 1995*, San Diego, California, December 1995.

[PP84] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 348–54, Ann Arbor, Michigan, June 1984.

[PT98] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In José Rolim, editor, *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 472–85. Springer-Verlag, 1998.

[PTVF93] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2nd edition, January 1993.

[RK79] D. P. Reed and R. K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–23, February 1979.

[RLW94] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325–36, Chicago, Illinois, April 1994.

[Rud81] L. Rudolph. *Software Structures for Ultraparallel Computing.* PhD thesis, New York University, December 1981.

[Saa93] R. H. Saavedra. Micro Benchmark Analysis of the KSR1. In *Proceedings of Supercomputing 1993*, pages 202–13, Portland, Oregon, November 1993.

[SB97] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the USENIX Windows NT Workshop*, pages 95–106, Seattle, Washington, August 1997.

[SB98] E. Speight and J. K. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 312–22, Las Vegas, Nevada, February 1998.

[SBC+96] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of Hot Interconnects IV*, pages 41–52, Stanford, California, August 1996.

[SBS93] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 109–18, San Diego, California, May 1993.

[SC87] M. L. Scott and A. L. Cox. An Empirical Study of Message-Passing Overhead. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 536–43, Berlin, West Germany, September 1987.

[SDH+97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 170–83, Saint Malo, France, October 1997.

[SGA98] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 125–36, Las Vegas, Nevada, February 1998. Also available as DEC WRL Research Report 97/3.

[SGT96] D. J. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–85, Cambridge, Massachusetts, October 1996. Also available as DEC WRL Research Report 96/2.

[SME97] UltraSPARC-I Data Sheet: First Generation SPARC v9 64-Bit Microprocessor with VIS, July 1997. Document #STP1030A. Available at http://www.sun.com/microelectronics/datasheets/stp1030a/.

[SS95] K. E. Schauser and C. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 140–9, Santa Barbara, California, April 1995.

[TBG+97] D. Teodosiu, J. Baxter, K. Govil, J. Chapin, M. Rosenblum, and M. Horowitz. Hardware Fault Containment in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 73–84, Denver, Colorado, June 1997.

[TM94] L. Tucker and A. M. Mainwaring. CMMD: Active Messages on the CM-5. *Parallel Computing*, 20(4):481–96, August 1994.

[Val90] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–11, August 1990.

[Val94] J. D. Valois. Implementing Lock-Free Queues. In *Proceedings of 7th International Conference on Parallel and Distributed Computing Systems*, pages 64–9, Las Vegas, Nevada, October 1994.

[vEABB95] T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-latency Communication over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–53, February 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–66, Gold Coast, Queensland, Australia, May 1992.

[VIA97] Virtual Interface Architecture Specification, Version 1.0, published by Compaq, Intel, and Microsoft. Available via http://www.viarch.org, December 1997.

[WBvE97] M. D. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-Level Communication. In *Proceedings of 3rd International Symposium on High-Performance Computer Architecture*, pages 332–42, San Antonio, Texas, February 1997.

[WKS95] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 199–206, Santa Barbara, California, July 1995.

[Yeu98] D. Yeung. *Multigrain Shared Memory.* PhD thesis, Massachusetts Institute of Technology, February 1998.

[YKA96] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 44–55, Philadelphia, Pennsylvania, May 1996.

[YSP+98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Proceedings of the Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 63–76, Austin, Texas, November 1987.