Implementing Distributed Shared Memory on an Extensible Operating System

by

Joseph Adler
MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Masters of Engineering

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

© Massachusetts Institute of Technology 1997. All rights reserved.

Author

Department of Electrical Engineering and Computer Science
May 23 1997

Certified by

M. Frans Kaashoek
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Implementing Distributed Shared Memory on an Extensible Operating System

by

Joseph Adler

MIT Laboratory for Computer Science

545 Technology Square

Cambridge MA 02139

Submitted to the Department of Electrical Engineering and Computer Science on May 23 1997, in partial fulfillment of the requirements for the degrees of Masters of Engineering and Bachelor of Science in Computer Science and Engineering

Abstract

The adoption of parallel computing systems for networks of workstations has been not only been limited by a lack of appropriate hardware and application software, but has also been limited by a lack of adequate operating system support. A version of the C Region Library (CRL) has been constructed for a system on a standard UNIX operating system and on an extensible operating system. These experiments have demonstrated that it is exceedingly difficult to get adequate communication performance using an off-the-shelf operating system. CRL is a distributed shared memory system designed to facilitate the programming of multiprocessors that share information through message passing. Performance tests comparing CRL on UNIX and XOK have indicated that CRL in conjunction with an extensible operating system allows user applications to achieve better communication performance on similar hardware.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Computer Science and Engineering
Contents

1 Background and Motivation ........................................... 9
  1.1 Structure of this Thesis ....................................... 10
  1.2 Introduction to Parallel Computing ............................. 10
  1.3 Characterization of Parallel Computers, DSM, and NOW work ... 11
  1.4 Discussion of Different Parallel systems. ....................... 13
    1.4.1 Parallel Computing Hardware .................................. 13
    1.4.2 Parallel Computing Software ................................... 15
  1.5 The Operating System Problem ................................... 16
  1.6 Contributions of this Thesis ................................... 17
  1.7 Related Work ..................................................... 18
    1.7.1 Networks of Workstations ..................................... 18
    1.7.2 Distributed Shared Memory ..................................... 19
    1.7.3 Operating Systems and Fast Networking ....................... 20

2 The Design of CRL for an Extensible Operating System ............ 21
  2.1 Introduction to CRL ............................................. 21
    2.1.1 Overview of The CRL protocol ............................... 22
  2.2 History of NOW CRL ............................................. 24
    2.2.1 Early Implementations ....................................... 24
    2.2.2 Protocol Optimizations ....................................... 24
    2.2.3 The Operating System Problem ............................... 26
  2.3 Exploiting Extensible Operating Systems ......................... 28
    2.3.1 The Exokernel Network Interface ............................ 30
  2.4 An Active Message Layer for Unreliable Networks ............... 33
    2.4.1 The Active Message Layer Interface .......................... 33
    2.4.2 Using Active Messages ....................................... 33
    2.4.3 Active Message Design ...................................... 35
A Protocol Details

A.1 Protocols .................................................. 75
    A.1.1 Broadcasts ............................................. 75
    A.1.2 Reductions ............................................. 76
    A.1.3 Active Message ......................................... 77

A.2 Correctness Arguments ........................................... 79
    A.2.1 Active Message Protocol ................................ 79
    A.2.2 Broadcast Protocol ..................................... 80
    A.2.3 Reduction Protocol ..................................... 81

B Preliminary Results ............................................... 82
    B.1 Barnes-Hut .................................................. 82
    B.2 FFT ........................................................ 83
    B.3 Blocked LU ................................................ 84
List of Figures

2-1 Invalidating a region without the Three Way Invalidate Protocol. ................. 25
2-2 Invalidating a region with the Three Way Invalidate Protocol. .................. 25
2-3 An illustration of applications using ASHes and Fast Upcalls for communication. . 31
2-4 The XOK Active Message Interface ................................................. 34
2-5 An illustration of Stop and Wait ARQ. ............................................ 37

3-1 The Message Format for the Active Message Layer ............................... 43

4-1 TCP and XOK synthetic benchmark results for sixty-four 64-byte regions ...... 63
4-2 TCP and XOK synthetic benchmark results for two hundred fifty six 64-byte regions 64
4-3 TCP and XOK synthetic benchmark results for two hundred fifty six 1024-byte regions 64

B-1 Barnes-Hut time and speed-up on 16384 bodies ................................. 82
B-2 Barnes-Hut time and speed-up on 4096 bodies .................................. 83
B-3 FFT time and speed-up on a matrix of size 1024 with block size 128 .......... 83
B-4 FFT time and speed-up on a matrix of size 1024 with block size 128 .......... 83
B-5 Blocked LU on a matrix of size 1024 with block size 32 ...................... 84
B-6 Blocked LU on a matrix of size 1024 with block size 256 .................... 84
# List of Tables

3.1 Region sizes for selected LU parameters ............................................ 52
3.2 Communication scaling with different matrix sizes and node counts ............. 52

4.1 Raw Active Message Round-trip Latencies ............................................ 56
4.2 Latencies on OpenBSD with TCP .......................................................... 57
4.3 Latencies on OpenBSD with UDP .......................................................... 58
4.4 Latencies (in microseconds) on XOK .................................................... 59
4.5 CRL message rates with TCP/IP and indexing by Region ID ......................... 61
4.6 CRL message rates with TCP/IP and indexing by Region Pointer .................... 62
4.7 CRL message rates with XOK and indexing by Region ID ............................ 62
4.8 CRL message rates with XOK and indexing by Region pointer ....................... 63
4.9 Times to complete the traveling salesman problem application, measured in seconds 65
4.10 LU Decomposition on various problem sizes with UDP ............................. 66
4.11 LU Decomposition on various problem sizes with TCP ............................. 66
4.12 LU Decomposition on various problem sizes on XOK ............................... 67

B.1 Experimental bandwidth and latency results for different region sizes, using two nodes. 84
Acknowledgments

I would first like to thank everyone in PDOS for the help that they have extended to me while working on this thesis. I greatly appreciated the advice, encouragement (and proofreading) I received from Frans Kaashoek. From Frans, I have learned an enormous amount about computer systems in general, and operating systems in particular. Most of all, he has taught me that building computer systems is hard, and that you can’t always predict how long it will take to get something working. I would especially like to thank Greg Ganger, Russell Hunt, Tom Pinckney, and Hector Briceno for leading me through setting up XOK machines and helping me find odd problems with my system.

I would also like to thank my office mates, Ulana Legezda and Xiaowei Yang for putting up with the pile of papers, coffee cups, and PCs at which I worked (my “desk”), and for listening to my problems with hard to track performance bugs.

My appreciation extends also to my apartment-mate Andrew Begel, for allowing me to bounce design ideas off of him, and for listening to my complaints about obscure x86 floating point bugs. It also extends to my fraternity, Alpha Epsilon Pi, without which I would almost certainly have had fewer friends and less fun at MIT (and probably would not have ended up in course VI).

Most of all, I would like to thank my Jenna, for helping to relieve my stress and comforting me when I was unable to find bugs in my project. Without her support and encouragement, it would have been much harder for me to write this thesis.
A large amount of research has been done on the importance of low cost communication for parallel computing, and much work has been done on building fast hardware to achieve fast communication, or writing efficient algorithms that work well despite slow communication. For a long time, the only way to achieve adequate communication performance for a large body of algorithms was to build complicated specialized machines for parallel computation. However, many people have noted that the aggregate computing power of a cluster of ordinary workstations often equals or exceeds the computing power of specialized multicomputers, and have also noted that modern local area networks have become as fast as, and in some cases faster than, the networks in traditional multicomputers. Additionally, big multicomputers turned out to be too expensive for research groups, large corporations, and even government agencies to afford. So, much recent work has been focused on utilizing inexpensive networks of workstations to perform parallel computation.

Networks of workstations work well for parallel programs which do not require a large amount of communication. However, the network performance of workstations with standard networks and operating systems is not good enough for running programs with fine-grained parallelism. Much research has focused on building specialized interconnections for workstations. We believe that there are cheap and readily available alternatives to these customized networks, such as Fast Ethernet or ATM, which should be able to achieve comparable performance to more elaborate networks. Unfortunately, standard operating systems often do not let user applications take full advantage of the performance of this hardware.

We have examined the operating system problem by exploring the benefits or using an extensible operating system, the XOK Exokernel [25], to implement a fast communication layer for the C Region Library system [34]. Our data shows that with the same hardware and user software, giving the application more control over the network interface can result in significant performance improvements.
1.1 Structure of this Thesis

In the remainder of chapter 1, I will describe the operating system problem by introducing the idea of running parallel programs on networks of workstations, and describing related work on Distributed Shared Memory, operating system performance, and NOWs. In chapter 2, I will introduce the C Region Library system and the XOK Exokernel, and will describe a design for a suitable communications layer for CRL on XOK. In chapter 3, I will give specific details about the design of this system, and how it was implemented on the Exokernel. In chapter four, I describe the benchmark applications I used to test performance, and give some data indicating how well the system performed. Finally, this thesis in chapter five giving some indications on what further work could be done to improve the system's performance. An appendix is included which gives detailed descriptions of the communication protocols used in this project.

1.2 Introduction to Parallel Computing

Parallel computing is an old idea; J. Von Neumann invented the first model of parallel computing almost fifty years ago. [17]. The basic notion of parallel computing is that sharing work among more processors lets you finish a job more quickly. By dividing a workload between multiple nodes, it is possible to accomplish more work in less time; parallel computing is only useful as a way to make computers faster.

There are a number of good arguments for parallel computation. First, any problems can be solved efficiently in parallel on multiple nodes. The degree to which a program can be split is called its average parallelism, and is specified as a function of the program's input size. Many important algorithms can be parallelized and run on multiple machines, and many theoretical results exist showing that important problems have a high degree of parallelism.

Secondly, the performance gains achieved by parallelizing a program often far exceed the performance gains achievable by purchasing a faster serial computer. It might not be possible to make a program run ten times as fast if it needed to be executed sequentially. But, if the program had a high degree of parallelism, then it might be possible to execute the program on ten interconnected serial machines and run about ten times faster.

Unfortunately, taking advantage of these theoretical speed-ups can often be difficult. First, it can be difficult to take a serial program and change it so that it runs on multiple nodes simultaneously. It is often difficult to rewrite a program as a set of parallel tasks which communicate by sending messages or sharing objects.

Secondly, there are costs associated with sharing information between nodes; it often takes a significant amount of time for a node to locate and retrieve a datum needed for a calculation. Some simplistic models of parallel computation (like the Parallel Random Access Machine, or PRAM...
model) ignore these costs and focus on the fundamental algorithm. More recent work, such as [19], has given us parallel models which take these communication costs into account. The degree to which communication is required to solve a problem is described by the problem’s granularity. Some applications, such as graphic rendering or factoring prime numbers, can be effectively coded as very coarse grained problems, which require only a small amount of communication to divide the problem on initiation and assemble the answer at the end. However, other applications, such as the Barnes-Hut simulation of the n-body problem, are very fine grained, and require a large amount of communication while running. For very course-grained problems communication costs are not a big problem, because there isn’t very much communication. But, for very fine-grained problems, communication costs can dominate many other factors in the program’s running time.

An important goal of parallel computing research is to figure how to minimize communication time so that fine-grained programs can run efficiently. Massively parallel supercomputers use exotic and complex networks to achieve low latency and high bandwidth communication. Additionally, many theoretical and practical results have focused on trying to figure out how to provide programmers with fast communication primitives. Communication costs are a big problem on networks of workstations, because workstations often use standard operating systems and standard network hardware. Though NOWs can easily be used for rough-grained problems, they are ill-suited for many problems which require frequent communication.

1.3 Characterization of Parallel Computers, DSM, and NOW work

There are many different types of parallel machines, and many different ways in which these machines can be programmed. Machines differ in control and data flow, in communication methods or memory consistency models, and in the degree of hardware and software support for parallel operations.

There are several different ways to characterize the flow of control on a parallel machine. In the 1960's, Flynn concocted a scheme which is still used today. He divided machines into four basic categories:

- **Single Instruction, Single data stream (SISD) machines** This is a fancy way of referring to serial machines.

- **Single Instruction, Multiple Data Stream (SIMD) machines** On these machines, multiple calculations are issued at a single time, but there is a single flow of control. For instance, several processors may be called upon to add different sets of numbers simultaneously. A large number of computers actually fit this category, including vector supercomputers, and even machines like the Intel Pentium with MMX (which can execute an instruction on several
pieces of data simultaneously). This is a very powerful computation model, as described in [11].

- **Multiple Instruction, Single Data Stream (MISD) machines** This is a fairly obscure type of machine; it is much more limited than MIMD machines, and could be easily simulated by them should a programmer choose to do so.

- **Multiple Instruction, Multiple Data Stream (MIMD) machines** These are the most general parallel machines. Many parallel computers fit this model, from massively-parallel machines like the Thinking Machines CM-5, to the popular IBM SP-2, to bus-based symmetric multiprocessors like Sun Workstations, to network of workstation environments.

Parallel systems also differ in the mechanisms they use to communicate between nodes. The simplest way to communicate is by **Message Passing**. A program communicates between different parallel threads (running on different nodes) by explicitly sending and receiving messages. Message passing parallel systems are by far the most popular because they are the easiest to build. Essentially, any set of processors connected by a network can be regarded as a message-passing parallel computer. Unfortunately, it can be very difficult to program a message-passing machine. In order to write a parallel application, a programmer needs to be carefully coordinate the sending and receiving of messages on different nodes. Additionally, the programmer needs to implement a mechanism for keeping track of where information is located on different nodes.

An alternative to message passing systems are **Shared Memory** systems. These systems allow a programmer to communicate between nodes by simply reading and writing data from shared regions of memory. Some systems provide a global address space while others share objects at different degrees of granularity. It is generally agreed that it is easier to program using shared-memory than by using message-passing, but it can be much harder to design and build efficient shared memory systems. Some shared memory systems share physical memory off the same memory bus, but bus-based systems do not scale well to large systems. Systems which distribute memory across multiple, but can be programmed using one large address space (or using shared objects), are called **Distributed Shared Memory** or DSM machines. These systems use more complicated networks and protocols to keep memory consistent.

At some level all shared memory systems send messages between nodes (or between memory hardware). There are many different ways to implement shared memory systems. Kirk Johnson divided these up as follows in [33]:

- **All Hardware** An all-hardware shared memory system provides all mechanisms for sharing in hardware. An example of this type of machine would be the Kendall Square Research KSR-1.

- **Mostly Hardware** Some systems rely heavily on hardware support, but to achieve efficiency,
generality, or correctness, may require software intervention occasionally. An example of this type of machine would be the MIT Alewife machine.

- **Mostly Software** Other systems are implemented mostly in software, but use some hardware mechanisms like virtual memory systems to help in efficiently providing sharing. Some page based DSM systems like SHRIMP fit into this category.

- **All Software** An all software system relies completely upon software to provide synchronization. These systems include page-based systems like Treadmarks, as well as object based systems like Emerald.

1.4 Discussion of Different Parallel systems.

Many of the parallel systems that other people have built are excellent systems which have been either fast, portable, easy to program, or cheap. However, very few systems achieve a good balance between these features. We will examine different approaches to building parallel systems.

1.4.1 Parallel Computing Hardware

**Traditional Parallel Systems** Perhaps the best known set of parallel systems are the SIMD vector machines, such as Cray supercomputers. Despite a restricted computational model, it is possible to get excellent performance out of these machines. However, only applications with rigid control structures can take advantage of vector parallelism. These machines are best for mathematical or scientific calculations, such as weather prediction or fast Fourier transforms, and do not work very well for more abstract tasks like playing chess.

Another important set of parallel machines are massively parallel computers such as the Thinking Machines CM-5, the IBM SP-5, and the Intel Paragon. These machines offer a more flexible programming model than the vector processors, because they allow multiple threads of control. Most of these systems require the application programmer to build programs by explicitly sending messages between nodes, requiring the application writer to carefully craft a protocol in order for the different nodes of the machine to communicate.

Some companies and researchers have built hardware based distributed shared memory machines, such as the Stanford Flash multicomputer, and the MIT Alewife machine. It can be extraordinarily difficult to build these machines; getting the memory consistency hardware to work correctly has proved to be much more difficult than researchers had hoped; and fixing hardware bugs can be very time-consuming and expensive.

The biggest problem with big parallel systems, however, has not been their limited programming interfaces, or the difficulty in construction, but in their high cost. A multi-million dollar supercom-
puter is beyond the means of almost everyone, including educational and research institutions, large corporations, and most government bodies. Even machines that have sold well, such as the IBM SP-2 multicomputers, have been sold primarily in small configurations of 4 or fewer nodes; IBM initially intended to sell over half its machines with over 100 processors.

**Small Multiprocessors** The last few year have seen the proliferation of bus-based multiprocessors, such as multi-processor Sun workstations, and some SGI multiprocessors. A bus-based multiprocessor is a computer with several processors that share physical memory through a common memory bus. These machines work well for many tasks, such as graphic-rendering and some mathematical calculations. However, performance of these machines is limited by the bandwidth of the shared memory bus. These machines work well for jobs which are not memory intensive, and can take advantage of each processor's cache. Unfortunately, the benefits of multiprocessing disappear after more than a handful processors are wired to the machine. Some recent research has focused on building larger parallel systems using these smaller nodes as a building block, and exploiting locality properties to get high performance.

**Networks of Workstations** The earliest systems for parallel programming on a Network of Workstations emerged at the same time as the earliest local area networks. During the 1970's, Birrell and Nelson invented the Remote Procedure Call [10] interface as a way of running computations on a remote machine. Since this time, many systems have developed for using the combined computing resources of a network to solve computational problems.

There are excellent arguments for constructing parallel computing systems using ordinary workstations, as spelled out more explicitly in [5]. First, real multiprocessors cost a lot of money for several reasons. The market for workstations or desktop computers dwarfs the market for parallel computing hardware. Moreover, parallel computers are often a good deal more complex and difficult to design than serial machines. Finally, there are enormous economies of scale in constructing computer hardware. The high costs coupled with low demand and a lack of economies of scale often make parallel computing systems disproportionately expensive.

Secondly, because parallel computers are difficult to construct, it often takes much longer to design and build a parallel machine. The development time required to design a parallel computer can be significantly longer than the time required to design a workstation. In many cases, by the time many of these machines are ready to use, the performance of many of their components lags behind the state of the art.

Third, modern workstations and local area networks often beat "real" parallel machines in processor speeds, network speeds, and storage space. Thus it is possible to purchase a set of workstations and a fast network that can theoretically outperform a monolithic parallel machine, even on fine-grained problems requiring frequent communication.

Finally, many organizations already have large local area networks in place, consisting of many
workstations that spend much of their time idle. Rather than wasting these machine cycles, many people have suggested stealing processing power from idle machines. Many such cycle-stealing systems have been constructed, which donate machine time on idle machines to running parallel processes.

1.4.2 Parallel Computing Software

Message Passing Systems

Despite the significant advances in hardware for parallel computing, the software for parallel computing has not improved as rapidly. The two most prevalent systems for parallel computing, the Parallel Virtual Machine (PVM) system [28], and the Message Passing Interface (MPI) [20], are based on a large set of obscure and complex commands. To efficiently program these systems, a programmer may need to learn hundreds of different communication modes!

Moreover, because these machines are based on message passing, they can be extremely difficult to program. In addition to keeping track of the algorithms of his program, a programmer needs to keep track of how the nodes are communicating and coordinate the message passing facilities of all of the nodes in the system. Message passing systems can be a poor choice for expressing complex problems such as physical simulations.

However, these two systems still remain enormously popular. There are two principal reasons for this. First, programs written for either of these two systems are extremely portable. A programmer can move his program from a network of three PCs running Linux to an 8000 node Intel paragon with only minor modifications. Additionally, these environments provide support for heterogenous computing so that a program can simultaneously run on a Thinking Machines CM-5 and a Cray Y-MP.

Secondly, these systems are popular within a small segment of the computing population: scientists. Although these systems are not well suited for abstract programming, they work beautifully for complex mathematical calculations. Moreover, programmers often don't have to worry about doing any parallel programming with these systems, because excellent libraries of commonly used functions are readily available—a programmer doesn't have to know how to code an efficient parallel implementation of a fast Fourier transform to use one in his application.

Distributed Shared Memory Systems

Currently, there are several distributed shared memory systems available for easily programming a network of workstations as a parallel machine, but most users still cling to message-passing libraries. There are two reasons why many users prefer PVM and MPI to systems with nicer programming models. First, systems with better programming models are often inefficient or don't run on readily
available hardware. Secondly, the performance of PVM and MPI is pretty good on many of the types of problems which scientific users want to solve. These are mostly rough-grained problems; it is fairly easy to divide a matrix computation among a set of parallel machines. It is probably not true that there is a large community of users running sophisticated programs on networks of workstations; they are actually running course grained problems which aren't affected by poor networking performance, or are not too complicated to code in message-passing style. Users who need to run more complicated applications are stuck using expensive parallel machines or even running sequential programs on ordinary workstations. The lack of a cheap, easy, efficient system for fine-grained computation is a significant obstacle in the adoption of parallel computing by a larger audience.

1.5 The Operating System Problem

The problem with many proposed systems to run fine-grained parallel programs is that they rely on expensive specialized hardware. If we want to run a program on cheap and readily available hardware, like PC workstations, it can be very difficult to achieve good performance. Much of the research on running parallel programs on networks of workstations has focused on building a small amount of specialized hardware to add to readily available hardware. Most of the time, this involves building a new communication interface.

However, there are a number of good, inexpensive, and readily available options for fast networks. ATM network cards and switches have inherently high bandwidth and low latencies. Fast (100 MB/s) Ethernet cards and switches are almost as fast as ATM, and cost much less. Additionally, networks like Myranet provide networks with bandwidth in the gigabit per second range for only a little more money. It should be possible to achieve very good performance using this readily available hardware.

Unfortunately, much of this performance is not available to users of standard operating systems. Programmers are stuck with expensive general purpose abstractions which deny them the full power of the available interfaces. It can often take a long time to make a system call (to check for messages, for example, or to handle an interrupt), because the operating system often needs to do a full context switch whenever a call is made. It may take a long time for a network interrupt to propagate to a user process. Finally, it may be difficult to send receive messages cheaply. Implementing network abstraction in the kernel is often done with complicated multi-level protocol stacks. With a multi-level protocol stack like TCP/IP, a user's message needs to move through multiple "levels" of code to reach its destination. For example, if we used TCP/IP for CRL, a CRL message would first be encapsulated in an active message packet. It would then be encapsulated in a TCP packet, then in an IP packet, and finally in an Ethernet packet. This is usually done by creating a linked-list structure (called "membufs") of packet components in the kernel, which are assembled into a single
packet when the packet is copied to the Ethernet card's DMA buffers [37]. It has been shown that this complicated hierarchical structure often leads to poor network performance [45].

Some systems have tried to circumvent these problems by giving user processes unprotected and direct access to network facilities through memory mapping ([41], [66]). However, these systems do not allow a network event to cheaply interrupt an executing process, but instead require parallel programs to explicitly poll for messages. Moreover, some user-level systems are not designed for a multi-user environment. This means that a bad program can corrupt critical data structures, thus interfering with other programs.

1.6 Contributions of this Thesis

We hope to accomplish several things with this thesis. First, we hope to demonstrate that it is possible to obtain good performance on ordinary, unmodified computer hardware by using the correct software. We present evidence of this by including results of a prototype distributed shared memory system built on an inexpensive PC workstations with an inexpensive Fast Ethernet network.

Secondly, we demonstrate that operating system performance is a great obstacle in obtaining good performance on a network of workstations. It is necessary to have good computer hardware, and good software for managing program complexity, but it is also necessary for the operating system to support fast communication between nodes. We believe that a properly designed and implemented set of abstractions will greatly aid the construction of efficient parallel software systems, while ordinary operating system abstractions will hinder this construction. In order to accomplish this, we chose an extensible operating system because the proper set of abstraction can not be efficiently implemented on a traditional operating system.

Finally, we hoped to design and build a system to fully exploits the power of the underlying hardware without sacrificing the system's safety or flexibility. This thesis details the process of designing and evaluating such a system, and our experiences implementing a DSM system with an extensible operating system.
1.7 Related Work

1.7.1 Networks of Workstations

The most popular and widely available systems for programming a network of workstations as a single large parallel machine are PVM [28] and MPI [20], two message passing systems which run on a large number of different machines. Both of these systems can run in a heterogeneous environment, so that a program can be simultaneously run on, for example, Sun workstations, and PCs running LINUX. These systems also work on MPPs such as the Thinking Machines Corporation CM-5 and Kendall Square Research KSR-1, or SMPs such as the SGI Onyx, or the Sun Ultra Enterprise servers. Despite the difficulty of programming these machines, they have become very popular in the scientific community because of their portability, and the availability of high performance libraries for many mathematical functions. Neither system requires operating system modification.

The Berkeley NOW project [5], [18], was one of the first large projects devoted to using a network of workstations a parallel machine. The NOW project first used Sun workstations, and experimented with ATM and Myranet for networking. The NOW project made use of Globally Layered Unix (GLUNIX) [58] as an abstraction. The GLUNIX layer allowed easy job control over the network. Several NOW efforts ([15], [41]) have address the issue of communication latency. The project made liberal use of Active Messages [60], [3], [59] to allow efficient communication. Of particular interest is the Fast Sockets implementation [48], which bypasses the common TCP/IP network path to give much better LAN performance in the common case.

The SHRIMP project [14], [21] at Princeton built a parallel machine using commodity Pentium PCs and an Intel Paragon backplane to provide communications. The SHRIMP project used some extra caching hardware to improve the system’s performance. SHRIMP allowed low cost communication using user-level instructions. Before using this model, users need to map shared memory using a pair of primitives for sending and receiving messages. While giving user processes the ability to directly communicate with network hardware yields high efficiency, it limits the security of the system, and makes it somewhat more difficult to run multiple threads on each node, or to use an interrupt-driven system.

The MOSIX project [2], [1], at the Hebrew University also constructed a NOW from readily available components. They used a set of Pentium PCs and a Myranet network. The MOSIX project did investigate operating system support, but focused mainly on scheduling questions and code mobility, and not on cheap communications.

The Parastation project [64] focused on building a system that allowed fast communications through PVM or MPI using specialized network hardware. NASA's Beowulf project linked together a set of Intel motherboards running UNIX with Fast Ethernet, and achieved far slower communication than in our system.
Treadmarks [4] was one of the earliest distributed shared memory systems to be assembled on a network of workstations. Treadmarks provides a simple set of programming abstractions and uses UDP packets on an Ethernet network, or AAL3/4 packets on an ATM network for communication. The Treadmarks system uses the machines virtual memory mechanisms to provide more efficient communications. Treadmarks also made heavy use of UNIX signals, and did not investigate lower cost communication mechanisms.

1.7.2 Distributed Shared Memory

A large amount of research has been done in the implementation of distributed shared memory systems, and many of these systems have been modified to run on networks of workstations. Much of this research has focused on the pros and cons of various programming models or of page based versus object based models. We briefly describe some systems which can be run on a network of workstations environment.

The Treadmarks [4] system described above used a novel consistency model, called Release consistency, in order to improve performance. The SHRIMP system [14] used a page based system as well, and included extra memory hardware (called Automatic Update) to improve performance. The Midway system [8] developed at Carnegie Melon was run on several machines, including an ATM based cluster. The Midway system achieved good performance using Entry consistency, but required some compiler support. Midway does not use a page based strategy, but instead asks the user to explicitly specify the relation between data and synchronization objects.

The CVM [57] system is another Distributed Shared Memory system written as a user level library using UDP for communication between nodes. CVM uses a variant of release consistency as its consistency model, and allows multiple threads on the same node.

The CILK language and run-time system [12], [32] developed at MIT includes an implementation for a Network of Workstations, and provides a novel consistency model called DAG-consistency [13], which is a derivative of its multithreaded, divide-and-conquer based model. CILK uses a provably efficient run time system to achieve good program performance for many applications. The CILK team has expressed some frustration with the non-extensible operating systems that they have used, and are interested in investigating the benefits of securely giving the CILK system control over memory mapping and communications.

Stanford's SAM [51] is another distributed memory system, which provides sharing at the level of objects. SAM is implemented a user level library, and run on a standard UNIX operating system using PVM.

The Shasta [50] system developed at DEC WRL is a software based distributed shared memory system, implemented on a network of Alpha workstations connected by an ATM network or a Memory Channel network. Shasta works by converting executables compiled for a hardware shared
memory system to a form that runs in a distributed shared memory system.

Additionally, several object based systems using specialized languages have been constructed. These include the Orca object based system [54], the emerald system [53] and several other.

1.7.3 Operating Systems and Fast Networking

Several projects have been devoted to finding ways to provide fast, though not necessarily secure, communication for user level processes. The HPAM [41] implementation of Active Messages on a network of HP workstations (using a FDDI network) got very low latency reliable communication by using some simple user level structures. However, the implementation did not provide security against malicious processes.

Several projects at the University of Arizona [44], [30] have focused on building an operating systems for efficient communication. In [45], they outline specific reasons why a traditional TCP/IP protocol stack often yields poor latencies, and propose several solutions (often at the instruction level) to optimize network performance.

The U-net project, as described in [66], [65], [6], was designed to give user level processes direct access to network interfaces to provide low latency communication. They implemented prototype systems using ATM and Fast Ethernet networks.

The Spin project is another extensible operating system, which achieves safe extensibility through the use of type safe languages [7]. This project has investigated fast network interfaces [26]. The Scout project: [44] is another extensible operating system, and its designers have also investigated fast network interface [45].

Within our own research group, Deborah Wallach has explored mechanisms for fast communication in depth on the Aegis Exokernel [63], [62]. Additionally, several projects have focused on using extensible operating systems to build fast servers, or other applications requiring low latency.
Chapter 2

The Design of CRL for an Extensible Operating System

We have implemented a parallel computing system on the Exokernel using the C Region Library system to provide communications to the programmer. The C Region Library uses a specialized active message layer to allow high bandwidth, low latency, reliable communication between nodes.

2.1 Introduction to CRL

The C Region Library is an all-software distributed shared memory system developed by Kirk Johnson, Deborah Wallach, and M. Frans Kaashoek. CRL provides the programmer with an elegant way to program a message-passing parallel machine. It gives the programmer a simple and intuitive interface for efficiently communicating between nodes.

A programmer using CRL shares information between the nodes of the parallel machine using special areas of memory called regions. Operations are provided to create regions, to map regions to a node's local address space, and to begin and end read and write operations on regions. The interface to regions is analogous to the normal C memory allocation operations, with some constructs added to assure that region operations appear atomic. An operations on a region is enclosed between region_start_op and region_end_op markers. Each operation is considered to be a single atomic action from the perspective of the other nodes. This leads to a consistency model similar to the memory consistency model of a hardware based system of individual loads and stores. The programmer references unmapped regions using unique region identifiers, which CRL can use to obtain a current copy of the region.
2.1.1 Overview of The CRL protocol

CRL uses a fixed-home, directory based invalidate protocol that closely resembles the protocols used by many hardware based DSM systems. Implementations of CRL on the MIT Alewife machine, and on the Thinking Machines CM-5 have been shown to be competitive with both all-hardware DSM implementations; and CRL applications have been shown to be competitive with applications that directly use the underlying message passing primitives. The CRL Library is largely self contained and requires neither a special language, special compiler, or special hardware to run.

To accomplish this functionality, CRL associated a state machine with each region that takes messages from the local user or the network as input, and sends messages or changes state as its output. The user interacts with CRL as a library, using simple intuitive commands to access memory. CRL communicates between nodes by sending active messages between nodes.

There are two principle kinds of state machines used by CRL for each region, depending on whether the region is on a home node or a remote node. The home node is (usually) the node on which the region is created, and all other nodes are remote nodes. In turn, each node can be in a set of states, depending upon what CRL operations have occurred on the region.

The states for the home node are as follows:

- **HomeExclusive** The home node has the only valid copy of the region and no CRL operations are in progress.

- **HomeExclusiveRip** The home node has the only valid copy of the region and a read operation is currently in progress on the home node.

- **HomeExclusiveWip** The home node has the only valid copy of the region and a write operation is currently in progress on the home node.

- **HomeShared** The home node and some other node or nodes have valid copies of the region. A read operation may be in progress on some other node.

- **HomeSharedRip** The home node and some other node or nodes have valid copies of the region. A read operation is in progress on the home node, and may be in progress on some other nodes.

- **HomeIip** The home node has a valid copy of the region, but some other node has requested that the home node's copy be invalidated.

- **HomeIipSpecial** The home node has an invalid copy of the region, but the local node has requested read access, or some other node has requested shared access to the region.

- **HomeInvalid** The home node has an invalid copy of the region. Some other node has a valid copy.
The states for remote nodes are very similar:

- **RemoteInvalid** This node does not have a valid copy of the region.
- **RemoteInvalidReq** This remote node does not have valid copy of the region, but has requested either shared or exclusive access to the region.
- **RemoteShared** This node has a valid copy of the region and some other nodes may also have a valid copy.
- **RemoteSharedReq** This remote node has a valid copy of the region and want to get exclusive access to the region so that it can begin a write operation.
- **RemoteSharedRip** This node has a shared copy of the region and is currently in the middle of a read operation.
- **RemoteModified** This node has the only valid copy of the region, but no operation is currently in progress.
- **RemoteModifiedRip** This node has the only valid copy of the region and is currently in the middle of a read operation.
- **RemoteModifiedWip** This node has the only valid copy of a region and a write operation is currently in progress.

If the Three-way Invalidate and Floating Home Node optimizations are active, then there are two additional states which a node can be in:

- **RemoteBecomeHomeReq** This node is currently a remote node, but the user has requested that it become the home node. The home node is currently being contacted.
- **HomeBecomingRemote** This node is currently the home node, but has received a request from a remote node to transfer “home” status to the remote node.
- **HomeThreeWayRipSpecial** The home node is in an invalid state, some other node has exclusive access, and a third node wishes to obtain shared access to a region. The home node is forwarding the invalidate request from the node requesting shared access to the node with exclusive access. (See below for an explanation of this optimization.)

A region’s state may change depending upon what actions have been executed on the local node, or what messages have been received from remote nodes. Safety and liveness properties for the CRL protocol have been proven in an unpublished (?) paper by Eddie Kohler and Nancy Lynch (and others?), although I don’t believe that they have been proven correct for the protocol enhancements made by Sandeep Gupta. For a complete description of the CRL protocol messages, see [29] and [33].
2.2 History of NOW CRL

2.2.1 Early Implementations

CRL was designed to work on the Thinking Machines CM-5. The CM-5 is a massively parallel supercomputer, with a Sparc processor and a vector unit at each node. The nodes are connected by a high speed network which allows small messages to be sent with low latency [38]. The performance on the CM-5 was good, achieving near linear speedups on a number of different types of problems. Kirk Johnson indicated in [33] and [34] that the performance of CRL on this machines would be similar to what could be achieved on a network of workstations with a modern network.

With this in mind, Sandeep Gupta and I adopted CRL to a network of Sun workstations. We implemented a reliable active message layer using Sun OS 4.2. We used Unix signal mechanisms to detect when a message was waiting, and used TCP to provide reliable transport.

On these machines, we achieved fairly good performance. However, it quickly became apparent that the performance of this system would not scale well to a larger set of computers or to workstations with faster processors. When we moved this application to a different network (a cluster of a dozen PCs running Linux), the poor performance of the underlying network became a bigger problem.

2.2.2 Protocol Optimizations

Sandeep Gupta added several optimizations to CRL in the version designed for the IBM SP-2 [29]. Based on the observation that most of the cost of the CRL protocol was due to the cost of transporting messages, he made several changes to the CRL protocol to cut down on unnecessary communication. The first optimization was the "Three-way Invalidate" protocol change. To see how this optimization works, we consider the following situation, illustrated in figure 2-1. Node A has a writable copy of a region, which is currently being shared with nodes B, C, and D. In order to get a valid copy of the region, node A sends a MsgExclusiveRequest message to the home node, which sends invalidate messages to nodes B, C, and D. The home node next waits for nodes B, C, and D to respond to the request, and finally sends a writable copy of the region back to node A. Both node A and the home node are stuck waiting for nodes B, C, and D to finish. Additionally, an extra message is required.

The optimization proposed by Sandeep is illustrated in figure 2-2. After the home node invalidates the copies held by nodes B, C, and D, nodes B, C, and D forward their acknowledgments directly to node A. This way, the home node is free to return to whatever work it was doing before. Additionally, fewer messages are required by the protocol, so less time is wasted processing messages.

The second optimization proposed by Sandeep was the "Floating Home Node" optimization. Suppose that some node n needs to make use of a region R for several consecutive operations. If
Figure 2-1: Invalidating a region without the Three Way Invalidate Protocol.

Figure 2-2: Invalidating a region with the Three Way Invalidate Protocol.
node $n$ were not the home, but instead some node $m$ was the home node of $R$, then several messages would need to be sent before each read or write operation was completed. If another node needed to utilize the same datum while node $n$ was in the middle of this sequence, then this could require a more expensive set of operations in order to invalidate and replace cached copies of the region. In order to speed up these operations, Sandeep proposed a new CRL operation which allowed a node to become a region's home. This optimization significantly reduced the number of messages needed for communication in many cases.

2.2.3 The Operating System Problem

The protocol changes reduced the number of messages needed by the protocol, resulting in much improved performance. However, we found that it was still hard to achieve acceptable performance on all systems because of a variety of operating system problems.

In order to implement CRL, we needed several simple things:

- **Reliable Message Delivery** The CRL protocol assumes that the underlying network guarantees reliable and at-most once message delivery.

- **Asynchronous Communication** CRL applications need to get messages off the network fairly regularly, to prevent other nodes from waiting too long. This requires that either application writers explicitly poll for protocol messages, or the system interrupts the application when messages arrive. If interrupts are used, they need to be very light-weight because they are bound to occur frequently.

- **Fast Delivery of Small Messages** Most messages sent by the CRL protocol are only about 20 bytes long; to guarantee good overall performance, we need to guarantee that small messages are delivered with low latency.

- **Reliable Broadcasts and Reductions** CRL applications rely on broadcasting and reduction primitives to share some data (such as region names), and to synchronize the application.

Unfortunately, it was very difficult to satisfy all of these requirements well on a straightforward (or even tricky) UNIX implementation.

First, UNIX limits a process to some arbitrary number of open connections. This would not have been a problem if it were not necessary to dedicate a connection for each link between a pair of machines when using a connection oriented protocol. This made it awkward for a host to determine the source of an incoming message without making an extra system call (like `select`). The other problem we faced was the lack of a reliable broadcast protocol. To broadcast a message to a set
of $n$ nodes, we were forced to send $\Theta(n)$ \(^1\) separate messages. To compute a global reduction, it was necessary to exchange at least $2n$ messages. (We do this in $\lg n$ rounds as follows: at each round, we divide the remaining nodes into pairs $(i, j)$. Node $i$ computes $f(i, j)$, and node $j$ leaves the computation for a while. After $\lg n$ steps, there is one node left who has computes the global reduction value. This node then distributes the value to another node. Each of these nodes then distribute the value to another node; this process continues for $\lg n$ steps until every node has received the reduction value. This procedure required $2n - 2$ messages to be sent, and takes at least $\lceil \lg n \rceil$ steps to finish.) Because network bandwidth is limited, and on Ethernet only one message can be sent at a time, the extra communication required for these schemes is prohibitive. We needed a way to multiplex and demultiplex packets within the CRL application, and to provide reliable broadcasting.

Secondly, low latency message delivery was extremely difficult to achieve on UNIX. A fast response time was critical for good performance. On UNIX, it proved to be extremely difficult to provide an efficient mechanism for asynchronous communication. The first problem we had was that the mechanisms for this communication were inefficient and had variable semantics.

To activate asynchronous communication, a process explicitly specifies that it wants a socket to use this feature. Next, the process declares a specific handler, which will be executed when a message arrives. To protect critical pieces of code, it is necessary to stop these interrupts. If a program finds that it is waiting during a time when interrupts are off, it may choose to explicitly poll for messages.

We had several problems with these signal mechanisms. The first problem is that the semantics of the system calls for setting up signals (fcntl, signal, and sigaction) varied greatly between the three flavors of UNIX that we tried (Sun OS, Linux, and OpenBSD). Some calls, such as ioctl and select, behaved inconsistently or incorrectly much of the time, forcing us to use several inelegant methods to assure correctness. Secondly, the signal mechanisms all required very expensive system calls. (At one point, I discovered that half the time spent in one of the CRL applications on the Sparc V machines was spent turning on and off signals!) Additionally, it could take a very long time for a signal to propagate from the TCP/IP stack to the user process.

The TCP/IP network protocols used by most UNIX systems work beautifully for certain tasks, like remote terminals, file transfers, and can even be adapted for certain real-time communication like video and audio. These protocols work well even when the network is prone to lose packets, or deliver them with errors, or when the two communicating hosts send messages through multiple networks halfway across the world. However, there is a great cost for providing these services. First, it takes a lot of time to provide reliable communications. Secondly, many of these features result in

\(^1\)When we say that a function $g(n) = \Theta(f(n))$, we are saying that $g(n)$ "behaves like" $f(n)$ for large values of $n$. Here, we mean that the number of broadcast messages increases proportionally to the number of nodes. For more details on this notation, see [17].
needless additional complexity.

With the TCP/IP version of CRL, we were unable to adequately exploit special knowledge held by the application about how messages were going to be used. On a UNIX system, the operating system needs to figure out where a message is supposed to go, make copies of the message in case it needs to be sent again, and construct a complicated set of headers to encapsulate the information. If we had access to the raw interfaces, we could avoid all this overhead entirely by caching ready-to-send headers. Much of this copying can be avoided if the application is given access to the underlying hardware, so that it can cache copies of packets and copy them only once to the network card’s DMA buffers.

Another problem with using TCP is that we occasionally waste bandwidth with extra acknowledgments. If CRL is in the middle of sending a packet to a node, or is composing a message to send to that node, then it would be better to implicitly acknowledge a packet than to explicitly acknowledge the packet.

Finally, it was difficult to exploit special knowledge about the underlying network when we used CRL with TCP/IP. The TCP protocol uses some fairly expensive checksums. This series of expensive checksums is completely superfluous on a system where we know that the underlying network can be trusted not to deliver packets with bit errors. (This effectively changes the types of errors encountered from “Byzantine faults” to “stopping faults.”) These checksums cost a lot of application performance, but are very difficult to avoid in a UNIX environment.

(We could actually avoid some of these problems with UDP. But, providing this functionality at user level requires more systems calls, and a more clumsy interface. As we demonstrate later, it is very hard to beat the performance of an in-kernel protocol like TCP with a custom protocol at user level on UNIX.)

2.3 Exploiting Extensible Operating Systems

To address these problems, we chose to build a system using an extensible operating system, the XOK Exokernel. The Exokernel is a revolutionary new operating system design, described by Dawson Engler, Frans Kaashoek in [25]. Traditional operating systems are monolithic programs which present the programmer with a specific set of abstractions for process control, communication, and storage; often, they hide memory management, scheduling, and other tasks from user level applications. For some applications, these abstractions are adequate, and can even by helpful in decreasing complexity and increasing portability.

However, some of these abstractions are ill suited for constructing many modern applications. Many of these abstractions were originally designed for very different machines which were used in very different ways. BSD Unix was originally written for VAX minicomputers, where each ma-
chine was shared between many users at the same time. Microsoft Windows was designed for non-networked PC users. It is not surprising that the same set of abstractions that worked well for corporate accounting programs, software development tools, or spreadsheets are poorly suited writing fast web servers, graphic rendering software, or parallel programming systems.

The TCP/IP protocols and Berkeley Socket network abstractions described above were originally designed for very different purposes than what we wanted to use them for. They were intended to be used for file transfers, remote logins, and electronic mail service to take place over a large geographical area. They were built to be robust systems, which would reliably work in the event of damaged, lost, delayed, or out of order packets. They were not designed for high speed communication between physically close machines, and are not particularly well suited for this purpose. TCP/IP requires the construction of multiple headers, obtaining routing information, and copying and checksumming packets multiple times; it is not well suited for building DSM systems. Unfortunately, it is difficult or impossible to sidestep these mechanisms on an unmodified UNIX system, and still get a program which performs acceptably efficiently. (For example, our system requires reliable message delivery, but we used an unreliable network. To assure that messages are reliably delivered, we needed to periodically retransmit unacknowledged packets. We could only accomplish this by either running a second process which periodically re-sent messages, using UNIX interval timers and signals to tell us when to re-send messages, or lazily re-broadcasting messages whenever library calls were made. All of these operations were very expensive.)

The Exokernel does not provide a monolithic, abstract, and complex interface to user processes. Instead, it provides a thin veneer which safely multiplexes system resources. Abstractions are still available to the programmer in the form of libraries. Specifically, the ExOS library gives the programmer the option to use many UNIX-like interfaces. However, a programmer is not stuck with a set of inappropriate abstractions. The programmer is given access to primitives very close to hardware level, should he decide to use them. For most applications, the programmer will pick certain library abstractions and reject other abstractions. The “80-20” or “Pareto rule” applies to Exokernel programming: a program spend 80% of its time in 20% of its code. An application often needs to only the pieces of the library which affect the 20% of the code that is taking the most time. Thus, a programmer will choose to optimize only the most important modules. (For our CRL implementation, we borrowed many functions from the ExOS library, and only re-implemented crucial network calls.)

By giving the user process access to fast network primitives and fast interrupt mechanisms, we felt that we could achieve performance far better than what we achieved with UNIX. It seemed likely that we would be able to achieve a level of performance which approached the predicted hardware performance. Other experiments with the Exokernel, such as Greg Ganger’s fast web server Cheetah [35], and Deborah Wallach’s work with CRL on the Aegis Exokernel [63], [62], had
shown that an order of magnitude performance difference could be achieved by managing resources at the application level.

2.3.1 The Exokernel Network Interface

An Exokernel should use the minimum amount of abstraction necessary to securely multiplex systems resources. The network interfaces of the XOK Exokernel was designed with this in mind. A user-level application is given an interface that is as close to the hardware level as can securely be given. Below, we briefly describe the mechanisms and structures which give a user process low cost access to the network hardware:

User-level DMA Buffers

Fast sending is accomplished by giving user level applications direct access to DMA buffers [23]. The user process can copy information directly into these buffers, and use a lightweight system call to tell the operating system when to send the message.

Dynamic Packet Filters

The DPF interface [23], [22] provides a very fast way to associate packets with user processes, by using dynamic compilation techniques to build filters. To use dynamic packet filters, a user process gives the kernel a safe description of characteristics which describe incoming packets. (For example, a TCP packet is identified by its Ethernet packet type, its IP packet type, and its source and destination port numbers.) The kernel then dynamically compiles a trie structure, so that incoming packets can be filtered in an amount of time proportional to the maximum description length.

Packet Rings and ASHes

When a packet is received, the XOK Exokernel provides two low-cost ways to give the packet to the application. When a packet needs to be dealt with immediately, whether or not the associated user process is running, Application Safe Handler, or ASHes [62], are the appropriate choice. An ASH is a small piece of code which is allowed to execute for a limited time on packet arrival. ASHes are also guaranteed to be safe, because they execute in kernel mode. The Aegis Exokernel used sand-boxed code to assure that downloaded user code could be executed safely and efficiently. The XOK Exokernel uses the x86 memory segmentation hardware to isolate ASH code.

Alternatively, when a packet arrives it can be placed in a user-specified packet ring. Packet rings allow the user process to tell the kernel exactly where the user process wants incoming messages to go, so that the process can process messages without making more system calls. To use this mechanism, the user process allocates buffer space for incoming packets and sets up a linked-list of
individual packet buffers within this space. The user process then tells the kernel to place packets associated with some filter into this space. When a packet arrives, the kernel copies it from the incoming DMA buffer into the appropriate packet ring. The user application is then free to read the packet whenever it needs to do so.

**Upcalls from ASHes and from the Kernel**

ASHes are not an appropriate mechanism for speeding up communication operations in all applications. A message may require more processing than can be accomplished given the limited time and memory resources allowed for ASHes. As an alternative, we provide simple mechanisms for exporting network interrupts to user level, which closely resemble the mechanisms used for handling page faults and IPC on XOK (and the network upcalls on Aegis). We have implemented two mechanisms for user upcalls: one from ASHes, and one from packet rings. The mechanism for ASHes is designed so that an ASH can try to deal with a packet completely within the ASH, but fall back to the user process if necessary. We felt that this was a good system design technique: ASHes can be used to make the common case fast, but upcalls can be used if ASHes fail. Alternately, if we don't believe that ASHes offer any benefits for an application, then we allow a process to bypass ASHes and instead use an in-kernel fast upcall mechanism.

Our design requires the kernel and user process to agree upon three pieces of information. The
first is the address of the user-level interrupt handler. The second is a location to store the value of
the user process instruction pointer at the time of the interrupt. The final piece of information is a
mutual-exclusion lock which tells the kernel whether or not it is safe to interrupt the user process. For
convenience, we store these three pieces of information in the user XOK user environment structure.

The user process sets up the upcall by declaring an interrupt handler address. With the ASH
based mechanism, the ASH asks the kernel to change the user’s instruction pointer to a specific
interrupt handler value each time the upcall is executed, so no additional set-up is required. With
the kernel based mechanism, the user makes a system call which tells the system that the user
process would like to be interrupted when messages arrive.

If the ASH mechanism is being used, then when a packet arrives, the kernel executes the ASH
code with the received packet. The ASH then decides whether or not to make an upcall. If it wishes
to make an upcall, then it checks the lock to make sure that the user program is not in a critical
section. If it is not in a critical section, then it makes the upcall. Otherwise, the ASH indicates to
the user process that an interrupt has arrived.

If the packet-ring mechanism is being used when a packet arrives, the kernel copies the packet
into the ring. It then checks whether the user process wants interrupts to be exported. If so, the
kernel next checks the lock to see if interrupts are OK. If interrupts are OK, then the kernel stores
the process’s old instruction pointer, moves the process instruction pointer to the interrupt handler
address, and restores the user environment. Otherwise, the kernel simply restores the environment,
and expects the user process to poll the packet ring for new packets when it leaves the critical
section.

The current implementation of upcalls on the XOK Exokernel does not provide any support for
preemptive scheduling; if a message arrives while an application is not running, the application does
not learn about the message until it is next swapped in by the kernel. In many cases, there can be
benefits to preemptive scheduling, but that is beyond the scope of this thesis. For more information,
see [63].

**XIO**

To simplify the interface to many of these mechanisms, Greg Ganger has written an Extensible I/O
abstraction (XIO) to simplify the process of reading packets from packet rings, and sending packets
through DMA buffers. We made use of this library because it provided some useful abstractions to
the XOK network interface.

**Wake Predicates**

The Exokernel also allows user processes to download predicates into the kernel, asking to be awakened
if an event occurs. Our current implementation does not make use of wake predicates. However,
it would be straightforward to modify CRL to use this feature, as it is likely to result in better performance when multiple applications are run on the same machine.

2.4 An Active Message Layer for Unreliable Networks

For the high-performance NOW version of CRL, we chose to keep the underlying message transport mechanism used by the CM-5, Alewife, SP-2, and Sun OS versions of CRL. CRL manages regions by sending Active Messages [60] between nodes. Active messages are an excellent tool for protocol design; they have been shown to provide an order of magnitude performance improvement over other message passing tools.2

2.4.1 The Active Message Layer Interface

An Active Message consists of two pieces of information. The first part specifies a handler to process the information. The second part consists of some data to be given as arguments to the handler. Most of the time, the handler is specified by simply giving the address of a procedure in the receiver’s address space.

A program interacts with the CRL active message interface through several procedures, described in figure 2-4. These can be divided roughly into active messages, broadcasts, and reductions. Additionally, there are some procedures to find address information (like node identifiers), and procedures to turn interrupts on and off. A program sends an active message (which consists of a handler address and some data) with the am_send procedure. A node receives active messages either through explicit polling using am_poll if interrupts are off, or is interrupted by the handler if interrupts are on. Some additional procedures which provide broadcasting and global reductions are also included in this layer.

2.4.2 Using Active Messages

Active messages are usually used as a sort of asynchronous Remote Procedure Call [10]. RPCs are an abstraction developed in the 1970s to allow simple communication between computers on a network. Remote procedure calls were designed to look like normal procedure calls, so that a programmer could easily program network applications, without worrying about the specifics of the network interface. A remote procedure call has two pieces. On the sending end, the programmer calls a procedure which sends a message to the receiver. On the receiver, there is a “stub” which receives the message, does some stuff, and returns a value to the sender. The idea of remote procedure calls

2What we have implemented here is mostly just the idea of active messages. Mainwaring and Culler have produced a set of specifications for an active message layer [3]. Our implementation does not try to meet this specification, but simply borrows some ideas from active messages to allow our nodes to efficiently communicate.
/* some data types: */
typedef unsigned long nonce;
typedef double (*commutative_func)((double, double));
typedef void (*handler)((u_short sender, u_long size, void *data, nonce sn));
struct am_addr {
    u_short pid; /* “process number” */
    u_short nid; /* “node” number within process */
};

/* interface procedures for active messages*/
nonce am_send(struct am_addr, handler, int, char *, int);
void am_poll(void);
void clock_or_io(timertype *t); /* wait for a message or for t time units */

/* address information */
ushort am_num_nodes(void);
ushort am_node_id(void);

/* reductions and barriers */
double am_reduction(double x, commutative_func f);
int am_barrier(void);

/* broadcast interface */
void am_broadcast_recv(int length, char *data);
void am_broadcast_send(int length, char *data);

/* to protect atomic sections */
void AMEnterCritical(void);
void AMExitCritical(void);
int AMCriticalDepth(void); /* number of nested AMEnterCritical() calls */

Figure 2-4: The XOK Active Message Interface
was to provide a simple programming interface for network communications. Most implementations also involve the use of servers which tell where certain services are and process incoming messages.

Active messages are used in a similar way. Much of the time, a node $a$ wants to get a piece of information from node $b$. Node $a$ sends a message to node $b$, specifying the address of the stub procedure, which generates a response. Node $a$ then polls for new messages, until it receives a response from node $b$. Node $b$, meanwhile, is interrupted when the message arrives. (Or, node $b$ has exited a critical section, and is polling for messages.) Node $b$ formulates a response, which calls another stub on node $a$. Node $b$ then returns to whatever it was doing. The message from $b$ then reaches $a$, and the stub on node $a$ is called. When node $a$ is done processing the message, it discovers that the desired message has arrived and continues with its computation.

### 2.4.3 Active Message Design

The original UNIX version used a simple set of primitives for sending active messages. We built this active message layer on top of TCP operations, and simply hid the signal mechanisms and socket multiplexing and de-multiplexing from the main CRL code. However, it was necessary to design a more elaborate protocol for the UDP and XOK versions of CRL, because the underlying network was prone to drop packets, deliver packets out of order, or deliver duplicate packets.

In order to provide reliable and fast communications, we implemented an active message library which runs over a generic unreliable network. We make two assumptions about the underlying network in our design. First, the library assumes that the underlying network is not error prone; if a message is received by the library, we assume that it is error-free. Secondly, we do not worry about the possibility that a node will crash. Both of these problems occur in real systems, and if we were trying to run distributed shared memory applications on multiple hosts on different continents we would need to consider this possibility. However, we have instead focused on how to write an efficient system for use on a LAN. Our belief is that most people trying to run parallel applications on a network of workstations would try to do so on a network of physically close machines, connected by an inexpensive high performance LAN. We composed our design with this basic idea in mind.

The protocol we used to implement active messages is essentially a "Go Back n ARQ Protocol." In order to implement reliable messages, the protocol needs to keep some state information. The state information can be divided into global state information and link specific state information. Global state information includes recently received broadcasts and the current reduction epoch. Link specific state information includes outgoing and incoming sequence numbers for packets, incomplete fragments of incoming messages, cached address information, and some cached messages (like acknowledgments).
Reliable Message Delivery

The protocol that we implemented uses an ARQ (Automatic Repeat reQuest) system to guarantee reliable, in-order message delivery (see figure 2-5. An ARQ protocol works as follows: When the sender has a message to send, it transmits that message to the receiver, tagged with a unique identifier, or nonce. There is a total ordering to the set of nonces, so that if the last packet received by node the receiver had nonce $n_1$, then the receiver expects the next packet it receives to have some nonce $n_2$ which succeeds $n_1$ in the total ordering. If node the receiver gets a message which it is expecting, then it sends an acknowledgment to the sender. If the receiver gets the wrong packet, or fails to receive a packet (as in the figure), then it sends a negative acknowledgment to the sender, specifying the packet for which node it is searching. When the sender transmits a packet, it expects that packet to be received with some period of time. If the sender does not receive an acknowledgment for the packet from within this period of time, then it attempts to retransmit the packet.

Unfortunately, this scheme leads to inefficient communications. The problem is that there can be only one packet in transit at any given time; and we require one acknowledgment per packet. If a message required, 10 packets to send, this would require 10 acknowledgments. When transmitting a large number of sequential messages, or transmitting a message too large to fit in a single packet, the network quickly becomes filled with acknowledgment packets, leading to poor performance.

Optimizations

In order to make this scheme more efficient, there are several optimizations which can be made to the protocol. The first optimization is to allow multiple unacknowledged packets to be sent at a given time. (This change is called “go-back-n arq.”) It is possible to make the window large enough so that each node can effectively take advantage of the entire bandwidth of the network.

However, this optimization results in an increase in complexity. In order to implement the “go back n” optimization, it is necessary for the sender to store up to $n$ packets in memory at any time, to be resent if necessary. We implemented this mechanism by using a ring of pointers. As there were at most $n$ packets on the line at any time, we knew that we only needed to keep $n$ different packets in memory. We directly referenced a packet by its nonce; spot $sn \mod n$ would correspond to packet $sn$. To avoid copying, we kept only pointers to data buffers in this location.

Because this ring of packets was specific to each link, we created a queue of unacknowledged messages. On the UDP implementation, we used UNIX signals and interval timers to periodically resend unacknowledged packets, and on the XOK version we added a call in the prologue code to do the same. As an optimization, we allowed the user to specify whether or not the buffer used in the message was to be freed after the send was done. This released the CRL code from waiting for a message to be acknowledged before returning, or copying the whole message from the CRL
Figure 2-5: An illustration of Stop and Wait ARQ.
buffer. When the \texttt{am\_send} command returns, it guarantees that a message will eventually reach its
destination (in the correct order, and without errors), but it does not guarantee that the message
has already reached its destination.

The reason why this optimization is not a problem is because the CRL protocol is already
designed to allow messages to be delayed in transit between nodes. A lost packet may result in a
much longer delay than would occur on the CM-5 or Alewife, but the underlying protocol is designed
for arbitrary length delays, so they should not present a problem.

The second optimization is the construction of a non-blocking resend. When a packet is sent, it
is placed into a queue of unacknowledged packets, along with the time when it was sent. Whenever
the sending side calls an AM operation, the active message interface checks this queue, removing
acknowledged packets and re-sending packets which have not been acknowledged in the correct
amount of time.

There are additional protocol optimizations possible, which further reduce the number of mes-
sages which need to be sent, or the time to send messages. A simple optimization (discussed above)
is not to acknowledge a packet explicitly if a send is in progress, because the packet can be implicitly
acknowledged by the outgoing message. Another important optimization is to reduce the number of
acknowledgments sent for a long message. If a message \( m \) requires a series of \( l \) packets \( m_1, m_2, \ldots m_l \)
to transmit the message, it is not necessary to separately acknowledge each packet \( n_i \). Instead, if
our window size is \( n \), and our window size \( n \) has been tailored so that the time required to transmit
\( n \) packets is similar to our retransmission delay \( D_R \), we will send out an acknowledgment for each
packet \( m_i \) where \( i \equiv -1 \mod \frac{n}{2} \), or where \( m_i \) is the last packet in the sequence.

2.5 Reliable Broadcasts and Global Reductions

The second challenge we faced was to provide a mechanism for reliable and efficient broadcast and
reduction messages. A broadcast message is a message sent by one node, and received by every
other node. A global reduction is the performance of some (commutative) function \( f : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R} \)
on a set of values residing on each node. More formally, we assign a value \( v_i \) to each node \( i \). We can
describe a global reduction more formally as follows. We use the notation \( a \cdot b \) to represent \( f(a, b) \).
Now, we define the result of the global reduction to be \( v_1 \cdot v_2 \cdot \ldots \cdot v_n \). At the end of a reduction
operation, all of the nodes should form a consensus about the result of the reduction operation. One
particularly important reduction operation is the global barrier. A barrier operation is simply
a stopping point in the code which every node must reach before any node continues beyond that
point. A programmer inserts a barrier at any point that he wants all of the nodes to synchronize,
for example at the end of a program.

When we implemented broadcasts and reductions, we had two goals in mind. The first goal was
to assure that our scheme was reliable and correct. The second goal was to assure that our scheme would be efficient. Our design was influenced by these criteria, and also by our knowledge of the network on which it would be run. Ethernet is a good broadcast medium: the cost of sending a packet to every node is the same as sending it to a single node. Thus, it was desirable for us to try to broadcast as many messages as possible. Secondly, the cost of assembling or processing an Ethernet packet is not as great as the cost of sending a packet. The reason for this has to do with the way that modern I/O systems are designed. Most modern network interfaces use direct memory access, or DMA, to communicate between the network card and the processor. Packet sending and receiving occurs asynchronously; the operating system places a packet in a special buffer when sending, or reads a packet from a special buffer when receiving. Because CPU time is only required to write to or from these buffers, the performance of the CRL library will not be directly tied to the cost of transmitting a packet. (This only applies to receiving messages, or to non-blocking sends.) Thus, in the design of our protocol, it seemed wise to waste a little bit of per processor bandwidth if this reduced the overall number of messages needed by the application.

2.5.1 Broadcast Protocol

We chose a very simple implementation of broadcast messages. Let node $a$ be the node sending the broadcast, and $b$, $c$, and $d$ be the nodes receiving the broadcast. To send a message, node $a$ transmits a broadcast packet on the network containing the message and a short identifier, or epoch which describes the number of broadcast messages sent over the network. After this message is sent, nodes $b$, $c$, and $d$ read the message from the network. At this point, each node places the broadcast message into a queue of waiting broadcast messages, which can be read at the application's convenience. Finally, each receiving node sends an acknowledgment to the sending node $a$, after a somewhat randomized delay. After node $a$ has received all of these acknowledgments, it increments its broadcast epoch, and continues execution. If node $a$ has not heard from some node, say $d$, after some delay, it attempts to rebroadcast the packet. Upon receipt of this packet, node $d$ will accept the broadcast information (if it lost the first packet), and reply to node $a$ unconditionally. In this way, we assure that every node receives the broadcast packet. Additionally, this is a very low cost set of operations. If we assume that no packets are lost, then a broadcast on a network of $n$ nodes requires exactly $n$ messages to be sent. However, it is possible that packets may be dropped by the network. We assume that the probability that a given packet is dropped by the network is bounded by some probability $\alpha$, and try to find the expected number of messages required to reliably broadcast a message.

To show that this expected number of messages required by this protocol is linear in the number of nodes, we set up a recurrence to describe the expected number of messages. For the case where
one node is broadcasting to \( n \) other nodes:

\[
T(n) = 1 + \alpha T(n) + (1 - \alpha) \sum_{i=0}^{n} \binom{n}{i} (1 - \alpha)^{n-i} \alpha^i (T(i) + n)
\]

We omit the proof that this recurrence is \( T(n) = \Theta(n) \).

### 2.5.2 Reduction Protocol

Our implementation of a protocol for global reductions tried to balance the same set of tradeoffs as the broadcast protocol. It is possible to compute a global reduction in \( O(\lg n) \) steps for \( n \) nodes, by pairwise computing the operation. However, this mechanism requires a total of \( O(n) \) messages. For a bus-based network where processing single received messages is inexpensive, there is a lower cost strategy. In our implementation, a global reduction is computed (essentially) as follows. Each node sends out the value which it is contributing to the computation. Also, each node receives the values from the other \( (n - 1) \) nodes from the network. Finally, each node computes an identical value for the result of the global reduction.

Unfortunately, it is still possible for messages to be lost on the network. So, each node (blindly) retransmits a message after some predetermined (but reasonable long) amount of time, until that node has received a message from every other node. This assures that every node will eventually receive every message. Moreover, it is important that no node moves on in the program beyond the global reduction before every node has reached the reduction. This is assured in the same way: the reduction call will not return until a message has been received from every node. If a message has been received from every node, then we are sure that every node has reached the reduction. Thus, no node will pass the reduction before every node has reached the reduction.

There is one other problem which we prepare for: the final message before a reduction may be lost. We need a mechanism to distinguish packets querying about one reduction message from packets querying about another reduction message. To do this, we a unique \textit{epoch} to each reduction message. The epoch is incremented after each complete reduction operation. In this way, we distinguish between old and new epoch messages.

If a node receives an old epoch message, this implies that some node has not received the last message about the previous epoch. So, one node will reply to this message. Specifically, suppose that the message is sent by node \( i \). Then, node \( i \)'s reply will be sent by node \( i + 1 \mod n \). In the event that node \( a + 1 \mod n \) has also not received the message, it does not reply. However, in time, node \( a + 1 \mod n \) will send out a request for the old epoch value.

More precisely, we know that for some \( j \) node has reached the next epoch. Let \( j \) be such a node where \( j - 1 \) has not received the message. Then, eventually, node \( j - 1 \mod n \) will request the epoch value, and node \( j \) will reply. Eventually, with non-zero probability, node \( j - 1 \) will receive
the message, and the program can proceed to the next step. In this way, we assure that eventually every reduction will be completed, and the program will make progress. Moreover, we see that if the probability of packet loss is suitably small, then a reduction can be completed by sending as few as $n$ messages! This is a significant improvement for us over the $n \lg n$ messages required to compute a reduction pairwise. (It is worth noting that if we were to switch to a switched Ethernet network, or to an ATM network, where point-to-point communications were possible, then it would be better to modify this protocol.

For a complete description of the various protocol messages, the appropriate actions, and short proofs of the correctness of these protocols, see appendix A.
Chapter 3

Implementation on the Exokernel

We implemented the protocol described above using UDP on a network of 166 Mhz Pentium PCs running OpenBSD, and achieved performance slightly worse than the performance of TCP/IP. It was somewhat surprising that the performance using a specially designed and implemented protocol was essentially the same as the performance of TCP/IP. We theorized that the reason for this was that with UDP we were stuck with some of the same overhead as with TCP/IP (like the same large protocol stack), and a need to make even more systems calls to send and receive message. Furthermore, we needed to use UNIX signals and interval timers to indicate when packet retransmissions were necessary, causing additional overhead. In order to get performance closer to what the machines were capable of, it was necessary to implement this protocol as close to the hardware as possible.

For our high performance XOK implementation, we used a specialized packet format for our messages, encapsulated by raw Ethernet packets and not UDP packets. (See figure 3-1). We also chose to use several innovative features of the Exokernel. First, we took advantage of the fast Dynamic Packet Filter interface [22] to quickly demultiplex packets for CRL processes. The DPF interface provides a very fast way to associate packets with user processes by using dynamic compilation techniques to build filters.

We used the user-level DMA buffer access to send assembled packets with almost no copying. One of the reasons why we couldn’t achieve optimal performance on the UNIX implementation was because of the complexity of passing messages into the kernel data structures.

Additionally, there is some overhead associated with IP fragmentation and reassembly. On a local area network, most of the overhead associated with TCP, UDP, and even IP was not useful. Instead, our protocol does packet disassembly and reassembly at the user level, allowing us to cut down on the required packet movement.

Third, we took advantage of the slightly more accurate timers available to user processes on
struct am_pkt {
    u_long cs;                  /* checksum */
    internal_handler h;        /* message handler */
    u_long debug_nonce;        /* unique identifier, for debugging */
    struct am_addr src;        /* the packet's source */
    struct am_addr dest;       /* the packet's destination */
    u_long size;               /* the message size */
}
union am_payload {
    struct red_pkt reduction;
    struct data_pkt data;
    struct bcst_pkt broadcast;
} msg;
};

struct bcst_pkt {
    nonce epoch;    /* the current broadcast epoch */
    u_long set;     /* hosts who have acknowledged the broadcast */
    u_long length;  /* length of broadcast message */
    uint8 payload[0]; /* broadcast message */
};

struct red_pkt {
    nonce epoch;    /* the current reduction epoch */
    u_long set;     /* nodes which have reached the barrier */
    double value;  /* the value contributed by the sender */
    double rvalue; /* the composite value of the set */
    double ovalue; /* the final value from epoch - 1 */
};

struct data_pkt {
    nonce sn;        /* id for this packet */
    nonce rn;        /* next id expected by this node */
    handler h;       /* handler to process the data */
    u_short flags;   /* more info on this packet */
    u_short bytes_this_msg; /* the size of the payload */
    u_long bytes_sent_so_far; /* for long messages */
    u_long total_bytes_to_send; /* for long messages */
    uint8 payload[0]; /* the message */
};

Figure 3-1: The Message Format for the Active Message Layer
the Exokernel and the user-level prologue and epilogue code. We used the Pentium performance counters because they allowed us to get far more precise timings than were available with UNIX time-reading system calls. (This was helpful in correctly timing when to resend packets.) Additionally, the Exokernel allows user processes to specify pieces of code to be run at the beginning and end of a process’s quantum. Rather than setting up expensive and complicated interval timers with signals, we hard-wired a call to our retransmission procedure into the epilogue code to guarantee that it was run once every quantum (which meant once every 10 ms).

### 3.1 The ASH Mechanism

With ASHes, small, safe message handlers can be installed in the kernel by applications. On the Aegis Exokernel (running on Decstations), safety was assured by “sandboxing” ASH code [62]. Before installing and executing the code, the operating system would make sure that the code did not access or jump to arbitrary memory locations (or make up code to do the same thing). On the XOK Exokernel (running on the PCs), ASH safety is assured by using the PC segmentation hardware. Each ASH is limited to a short running time and a small amount of address space. When a message arrives for a specific process, the associated ASH is executed and allowed to immediately take some action. (For example, a quick acknowledgment can be computed at this time, or an internal structure can be modified.)

We initially took advantage of ASHes on the XOK Exokernel. The idea behind ASHes is that it is sometimes necessary or useful to perform certain computations within the kernel when a message arrives. If an application is not currently running, or if a response is simply needed quickly, then a standard operating system will have to schedule the target process before the message can be processed.

Unfortunately for our purposes, the address space provided by the ASH was not generally enough to house all the data needed by CRL. Thus, many requests could not be satisfied from directly within ASH space. If a sandboxing mechanism were written for the Exokernel, then such a thing might be possible. But, because the implementation relied on x86 hardware protection mechanisms, we were limited to 4 MB of virtual address space. So, instead, we were forced to design a different mechanism to achieve reasonable performance without placing arbitrary restrictions on application memory usage.

### 3.2 Fast Upcalls from ASHes

Instead of trying to satisfy all request from within ASHes, we used a quick and simple upcall mechanism, similar to the one described in [63]. If the target application is currently running, fast
upcalls preempt the running process and, if it is not in a critical section, force it to execute a handler immediately. If a process is not running, the upcall occurs immediately when the process is next given CPU time. A simple mechanism allows an ASH to request that when the program continues executing it will begin by immediately executing a special handler. This mechanism results in very low-latency message delivery.

To correctly implement this interrupt mechanism it was necessary to do two things. First, certain procedures needed to execute atomically without being interrupted. Data structures needed by the AM or CRL protocols could end up in an inconsistent state if interrupts occurred while they were being modified. For example, the sending and retrieving code could not be interrupted or critical data structures would become corrupted. Secondly, once a message was processed, the process needed to be able to continue to execute in the place where it left off.

In order to make sure that ASH upcalls would not interfere with executing processes, we implemented a simple locking mechanism which allowed the programmer to declare that he was executing a critical section of code, where interrupts could cause race conditions. Upon exiting a critical section, the user process quickly checks to make sure that no messages arrived during the critical section. If it is discovered that messages arrived during the critical section, the program tries to immediately process these messages. Simple "mutex" locks are used to mark critical sections. (Because it is possible to have nested critical sections, a variable critical_depth is used to denote the number of nested critical sections. This value is incremented on entering each critical section, and decremented on leaving each critical section. Only if the depth is zero do we know that it is OK to process arbitrary messages.)

The interfaces for the kernel and ASH upcall mechanisms are very similar. The user application begins by setting up a piece of handler code to deal with the incoming message. (If the ASH mechanism is being used, nothing more needs to be done. But, if packet rings are used, then the user calls the function sys_dp_set_intr to associate a packet filter (and hence a packet ring) with an interrupt handler.) When a message arrives, the processor receives an interrupt, telling it that there is information waiting to be read on the network card. If the target application is executing, then the kernel saves the process's registers in a trapframe so that its state can later be restored. Otherwise, the target application is not running, and its state is already saved in a trapframe. If ASHes are being used, then the kernel calls the appropriate ASH to process the packet. The ASH then processes the packet, and may make an upcall. The ASH does this by checking if the target process is in a critical section. If it is not, then it calls the sys_ash_upcall function to tell the kernel to change the target process's instruction pointer to the handler procedure, and save the old instruction pointer so it can later be restored. The ASH then marks the process as being in a critical section, so that recursive upcalls are not made. Recursive upcalls are not allowed unless a user process specifically asks for them. This serves two purposes. First, it eliminates the complexity
required in keeping a long list of instruction pointers from a handler which has been interrupted several times. Secondly, it helps guarantee that a procedure can continue to make progress in processing messages, and does not become so overwhelmed by new messages that it fails to deal with old ones.

If packet rings are being used, the kernel then checks to see if the packet ring has interrupts turned on. If interrupts are on, the kernel looks in the user’s environment structure to see if the process is in a critical section. If the process is not in a critical section, then the kernel modifies the target procedure’s trapframe so that its instruction pointer points at the handler procedure. The kernel next saves the old instruction pointer in the user environment so that it can later be restored by the target process. Additionally, the kernel will mark the target process as being in a critical section to avoid the problem of recursive interrupts described above.

On the process end, we coded a procedure which duplicates the actions taken by the kernel interrupt procedure, with a couple of minor variations. The implemented handler borrows the user process’s the interrupted process’s stack, rather than using a special interrupt stack. So, the upcall handler begins by dressing up the user stack to look like a procedure call was made to the handler procedure. (This allows us to easily and simply restore the procedure state after the handler is finished.) Next, the handler procedure saves all of the processor state so that it can be restored after the handler is finished. Finally, the handler executes procedures to process the received packet. When the processing is finished, the processor restores the program state and leaves the program in the same place where it was when the first interrupt occurred.

It was necessary to adopt this mechanism because periodically polling for new messages yielded poor performance. Exporting the kernel interrupts to the user process improved the system’s performance by a good margin. Unfortunately, our first try at the mechanism was somewhat clumsy because we were using upcalls within ASHes without taking any advantage of the special features of ASHes. The current version of CRL does not make use of ASHes, but instead uses packet rings and fast upcalls.

3.2.1 Debugging the Library’s Performance and Correctness

In implementing this protocol on different types of machines (SPARCstations and Pentium PCs), and different operating systems (SunOS 4.2, Linux, OpenBSD, and XOK), we got a number of very different bugs. One bug involved an arbitrary change in argument order in the version of CRL with the Three-way Invalidation and Floating Home Node optimizations enabled. Another bug involved asymmetrical network byte order conversions. (The network byte order conversion routines were null macros on the Suns, but had an effect on the Intel machines.) One bug involved the use of the wrong size structure as an argument to an Exokernel routine (and resulted in destroying the calling stack frame). We had numerous problems due to improperly locking variables. And, it took a long
time to tweak the performance of the library to be close to optimal.

With many of these bugs, the place where the error was made and the place where the error appeared were often very far apart in the application code. The effect of this was to make normal debuggers such as gdb almost useless. Instead, we had to rely heavily on two techniques: sanity checks, and logging protocol messages and arguments.

Throughout the program, there are a large number of invariants which need to be maintained by the Active Message library. For example, a ShortMessage message needs to have a size less than the size of a single Ethernet packet. The Active Message library code makes liberal use of assertions to assure that these properties are maintained. (We also made these assertions switch-able, so that when we were evaluating performance, we wouldn't be wasting time with assertions.)

A second technique we used liberally were switch-able debugging messages, similar to techniques used in writing the Amoeba distributed system. Different parts of the program were assigned logical debugging levels—protocol messages were separated from locking calls, for example—which could be easily turned on or off with compiler arguments, depending upon where an error was found to be. We did this by using C macros to code switch-able debugging statements to save on typing. Each debugging message would print the line number, node id, and procedure name where the debugging message was printed. The principle debugging messages we used were as follows:

- **dmark** We used this to easily mark the beginnings of procedures.
- **ddone** We used this to mark the end of procedures.
- **dprintpacket** We used several different types of messages of this sort to print out the contents of an entire packet for inspection, including a neatly formatted header.
- **dprintf** This allowed us to print arbitrary information.

By saving these messages to a log as they were printed out, we could reconstruct an exact trace through the CRL code to an arbitrary level of detail. By analyzing these traces, it was possible to trace the most nefarious bugs\(^1\)

Performance debugging was somewhat trickier for a network protocol. Normal profiling revealed some problems (like an excessive number of cycles wasted in UNIX signal code), but it was often hard to connect the cause of a performance problem with its effect. In order to analyze the performance of our system, we used several additional techniques. First, we began to collect statistics on the distribution of packets. This allowed us to find how many messages were lost, how many were

\(^1\)These messages turned out to be useful in another way. Because the CRL and AM code was littered with debugging messages defined by a small set of macros, it was possible to insert all sorts of assertions into these macros. When I had a problem with a variable being trashed at some mysterious and hard to find place in the code, I added an assertion to one of the debugging messages so that I could stop the program as soon as the variable was changed to a wrong value.
duplicates, how many were reduction messages, etc. It also gave us a specific set of parameters for which we could target improvements. For example, we implemented an increasing delay between the broadcast of reduction packets, after we discovered that an unusually large number of packets were being sent by the protocol. Additionally, we were able to isolate the number of explicit acknowledgments, and determine average message sizes. This led us to optimizations which reduced the frequency of acknowledgments when we knew that this was safe to do.

Another technique we used was to test performance on a set of very simple benchmarks, and to try to minimize latency and maximize bandwidth. We supplemented this technique by getting basic block counts, and by timing how long certain communication operations took. This allowed us to estimate how much time the CRL code was spending in certain sections.

3.2.2 Unresolved Bugs

Packet Errors

There are still several bugs which remain unresolved in the XOK CRL implementation. The first problem is that approximately one in one thousand packets received is corrupted. We believe that this problem may come from a defective network card, but it is more likely a side effect of a software bug. We think that an interrupt at the wrong moment may be wiping out a processor register, and causing packets to be copied slightly incorrectly.

To work around this problem, we have implemented a very simple checksum mechanism. We calculate a checksum by XOR-ing the entire contents of the packet in 16 bit pieces, and include the 16-bit result with each packet as a checksum. (We also implemented an Internet style cyclical checksum, but decided that the XOR checksum was somewhat more efficient, and worked just as well for this random error.)

Floating Point Errors

Some of our applications (Barnes-Hut and water-nsquared) have been difficult to run because we non-deterministically get floating point traps in the parallel applications on both OpenBSD and XOk. There could be many reasons for this; faulty user level code or incorrect context switching are the most likely causes. There is also a chance that we have found a hardware bug, but this is exceedingly unlikely.

Overflowing Buffers

The current implementation does not have flow control. Unfortunately, it is possible for a single machine to be completely flooded with requests from other machines; we can easily prevent a machine from making progress by flooding it with incoming messages. In order to make CRL a practical and
reliable system, we need to implement a mechanism for flow control. (Doing so would require either some kernel level support if we wished to use packet rings, but would be straightforward if we used ASHes.)
3.3 Applications and Performance Predictions

3.3.1 System Parameters

We have evaluated the performance of CRL on several different sets of machines. Our primary interest was to figure out how well CRL would perform on a set of PCs connected as follows:

- **Processors** Each node was a PC workstation, with a 200 Mhz Pentium Pro processor.
- **Memory** Each node had 64 MB of RAM.
- **Network** Each node was connected to a 100 MB/s fast Ethernet. We used an SMC Fast Ethernet switch interconnect the nodes. The use of the switch should not have greatly affected the latency, because it used cut-through routing. Note that the minimum packet size allowed on this network is 64 bytes, and the maximum is 1520 bytes. Each Ethernet packet requires a 14-byte header, and a 6 byte footer. This led to payload sizes of between 44 and 1500 bytes.

3.3.2 Synthetic Benchmarks

In order to measure the performance of CRL, it was important to start by measuring the latency and bandwidth associated with sending small and large messages using CRL, the underlying protocol, and the hardware.

**Latency Measurements**

To measure latencies, we borrowed a benchmark program used by Kirk Johnson in evaluating the CM-5 and Alewife versions of CRL. The benchmark works by allocating 64 regions on a home node. The benchmark then sets up a set of nodes for a set of possible CRL scenarios: starting to read on a node with a cache hit, ending a read operation, starting to read on a node with a cache miss and varying numbers of invalidations needed, etc. The program does each of these tests for all numbers of regions between 1 and 64, then computes the linear regression of the time needed for these operations. The slope of the line then corresponds to the time required for each of these operations. This allows us to get a feel for how quickly the CRL system processes application requests.

Additionally, we wrote several simple benchmarks to try to estimate the average latency and bandwidth of the Active Message library on the Exokernel, and to try to find the best latencies and bandwidths achievable with the Exokernel. We did this by simply timing a large number of round trip messages, using the Pentium performance counters. To estimate latencies, we used small messages. To estimate bandwidths, we sent a number of large regions in sequence, in an attempt to maximize our usage of the buffers on each node.

It is our belief that the bandwidth available on the Exokernel should be very close to the maximum bandwidth which the hardware is capable of.
Synthetic Workload

We also tested the synthetic workloads written by Kirk to better understand our performance data. The synthetic workload creates a user-specified number of regions of a user-specified size, then proceeds to perform 1024 random CRL operations on these regions (reading and writing these regions), interspersed by some time wasting activity. The benchmark varies the ratio of reading to writing operations, and returns values which describe the rate at which operations occur. (The rate is specified in the number of CRL operations per second.) This workload can be used to determine the sensitivity of CRL operations to network parameters (like latency and bandwidth), or to give a sense about the characteristics of applications which will perform well with CRL.

3.3.3 Applications

To evaluate the performance of CRL on a network of workstations, we used a similar set of benchmark applications to the set of applications used to evaluate the performance of CRL on monolithic multiprocessors. We borrowed a set of applications from the Stanford Parallel Applications for Shared Memory (SPLASH-2) suite [67] for our testing. The SPLASH suite consists of a set of applications for shared memory multiprocessors which have been selected because they present challenging workloads which are representative of the types of applications people would like to run on multiprocessors.

LU Decomposition

LU decomposition is a technique for factoring a dense matrix into the product of a lower triangular matrix \( L \) and an upper triangular matrix \( U \). The algorithm for doing this is fairly straightforward, and is based on several algebraic properties. For a matrix of size \( n \times n \), the running time is \( O(n^3) \), and the average parallelism is \( O(n^2) \). This implies that we should be able to easily run this program on a large number of nodes.

The SPLASH-2 version works by dividing an \( N \times N \) matrix into a set of \( B \times B \) blocks. For a block size argument of \( b \), each transferred block will be of size \( \text{sizeof(double)} \cdot b^2 \) bytes. This implies that for most practical block sizes (16 by 16 or larger), our program requires at least 2 Ethernet packets for transmission. The LU matrix decomposition is a good benchmark application because the problem size and granularity can be easily varied, changing the communication load. We have achieved reasonable speed-ups for this application, indicating that there is a good ratio between CRL operations and useful work. A sample set of results are depicted in figures B-5 and B-6.

The LU application breaks a dense matrix into smaller blocks that are distributed across the processors. For a block size argument of \( b \), each transferred block will be of size \( \text{sizeof(double)} \cdot (\text{blksz}^2) \), which means that it is about \( 4 \cdot b^2 \) bytes. The number of blocks per matrix is proportional
The calculations here are based on slightly different problem sizes than the problem sizes in the plots listed above. Table 3.1 illustrates the region sizes for different matrix and block size parameters.

The next table gives the frequency with which blocks move between nodes. According to [46], communication scales with $\frac{n^{\frac{1}{2}}}{p}$. Communication scaling values are given for different matrix sizes and node counts in table 3.2.

The ratio of computation to communication decreases as the number of nodes increases; less work per computation is accomplished for each additional processor.

**Fast Fourier Transform**

The Fast Fourier Transform application is an implementation of the 3-dimensional formulation of this important problem. This problem uses a radix of $\sqrt{n}$ to minimize the amount of communication required. This application runs in $\Theta(n \lg n)$ time.
Barnes-Hut

This application is an instance of a program to simulate an n-body problem. An n-body problem is a simulation of the interaction of a set of n bodies under the influence of some force. The Barnes-Hut application simulates the interaction of a system of galaxies influenced by the force of gravity. The complexity of this problem is potentially large (there are $\Theta(n^2)$ interactions among n bodies), so the Barnes-Hut application uses several techniques to simplify the model. Because gravity is inversely proportional to the square of the distance between two objects, it is possible to approximate the forces experienced by distant objects. In the situation where one body is far away from a set of other bodies which are near each other, Barnes-Hut lumps together the distant set into a single aggregate body. This effectively reduces the complexity of the problem to $\Theta(n \lg n)$ time, and $\Theta(n)$ parallelism.

However, to accomplish this, Barnes-Hut uses a specialized and complicated data structure. Space is divided into eight quadrants, and then each quadrant is subdivided into further quadrants. These are represented in memory using an octree data structure. To compute the force acting on a body, the application traverses this tree to find the forces of its neighbors; however, the application does not visit every node. Instead, the application only visits nodes along the path to the desired body. This reduces the problem complexity significantly, but requires frequent accesses to a complex and distributed data structure. Because the octree structure is stored in a large number of regions, the Barnes-Hut application frequently requires obtaining distant regions to calculate the force acting on a body. Thus, Barnes-Hut is an unusually challenging and computationally-intensive application for a parallel machine, and especially for a network of workstations.

Water Simulation

The water program is another challenging application for CRL. The application simulates the interactions between a set of water molecules over time. A set of forces and potentials are associated with each molecule; these are computed using a $\Theta(n^2)$ algorithm and a “predictor-corrector” method. The water application is another computationally challenging application for the CRL/NOW system because it also requires a lot of communication.

Traveling Salesman Problem

The TSP program is a simple program that finds an optimal traveling salesman tour on a weighted graph of twelve nodes. One node acts as the master and controls the other “slave” nodes. Each slave is given an approximately equal slice of work. Eventually, all promising paths through the graph are checked by the application, and the best length path is found. We expect this application to take about the same amount on one or two nodes, because on one node there are no slaves, and
on two nodes, there is one slave which does all the work. Thus, we expect that the performance of the application on 4 nodes will be about 3 times as fast as on two nodes.
Chapter 4

Measurements

4.1 Measurement with Extensible and Standard Operating Systems

To evaluate the performance of our system, we present three sets of measurements for some benchmark applications. We present the performance of CRL using TCP/IP running on OpenBSD, to show what a straightforward implementation can accomplish. Next, we present the performance of CRL using the new active message layer and UDP on OpenBSD. Finally, we evaluate the performance of the system using our active message protocol on the Exokernel.

We had some problems getting certain applications to run on some problem sizes. In some cases, we were unable to allocate enough memory for CRL and the application on the machines, and in others we had the floating point errors described above. These cases are marked with a † to indicate that we were unable to either run or accurately measure these applications. However, it seems that most of these problems were not directly related to our shared memory system. Instead, it appears that the problems were related to the way the active message library interacted with other libraries on the Exokernel; CRL and the scientific applications we tried used the XOK Exokernel in different ways than it had been used before. The problems that surfaced during the performance tests do not invalidate the results presented here. Thus, we omit results for Barnes-Hut, Water-nsquared, and FFT in this section because the system was not stable enough to make meaningful measurements.

4.1.1 Active Message Layer Latencies

In table 4.1.1, we list round average measured round-trip latencies for several packet sizes on the different systems. These measurements were taken with a simple program which bounces a packet back and forth between two nodes. For these tests, we have deliberately turned off acknowledgments, to get a sense of how much time is required per packet. We find that it takes approximately 65
μ-seconds for a CRL active message packet to travel between nodes, which works out to be about 13000 processor cycles. We notice that for larger message sizes, the gap between TCP and XOK times narrows. This indicates that for large messages the bottleneck in efficient communications is not overhead, but is instead network bandwidth.

### 4.1.2 CRL Latency Data

Based on the raw latencies described above, we expect the performance of CRL to be much better on XOK for small message sizes, and similar for larger message sizes. This is consistent with the results we obtained.

In tables 4.2, 4.3, and 4.4, we see measurements of the latencies of varies CRL primitives on several different region sizes. These measurements are the most revealing measurements, and tell us the most about why CRL performs well or badly on a specific platform. As noted above, for programs like LU decomposition, there is a CRL operation once every few thousand instructions, while with programs like Barnes-hut and water-nsquared, CRL operations occur much more frequently. Often, these requests can be processed without using the network. (The times for these operations are given as Map hit [a], Map hit [b], Unmap [a], Unmap [b], Start read hit [e], Start Read hit [f], End read [g], End read [h], Start write hit, and End write.) We notice that each of these operations is consistently fast and most take less than 5 microseconds. None of these operations require sending any data, so we expect them to be consistent across packet sizes, and find that for the most part they are. (The UDP times are slightly longer. This extra time can be explained by our use of interval timers and signals for retransmission. These operations are also interrupted in the TCP and XOK versions, but the signal mechanism is very expensive, and changes the average time.) This implies that most of the overhead of the CRL protocol comes from communication costs, and not from other parts of the CRL library.

When we compare the times with UDP packets and TCP packets, the UDP times consistently lose. There are several reasons for this. First, the UDP implementation explicitly acknowledges messages from user space. Thus, each CRL event requiring communication requires as many a four system calls to send messages, compared to only two with TCP. Secondly, as discussed above, the UDP version uses UNIX signals and interval timers for retransmission, adding to the cumulative running time.
<table>
<thead>
<tr>
<th>event</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map miss</td>
<td>473.12</td>
<td>471.01</td>
<td>497.14</td>
<td>568.93</td>
</tr>
<tr>
<td>Map hit [a]</td>
<td>.40</td>
<td>1.01</td>
<td>.49</td>
<td>.68</td>
</tr>
<tr>
<td>Map hit [b]</td>
<td>.31</td>
<td>.39</td>
<td>.76</td>
<td>.90</td>
</tr>
<tr>
<td>Unmap [c]</td>
<td>.13</td>
<td>.14</td>
<td>.32</td>
<td>.45</td>
</tr>
<tr>
<td>Unmap [d]</td>
<td>.29</td>
<td>.33</td>
<td>.60</td>
<td>.79</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 0 copies</td>
<td>651.61</td>
<td>535.40</td>
<td>718.73</td>
<td>1127.48</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 1 copies</td>
<td>494.67</td>
<td>533.69</td>
<td>719.37</td>
<td>1196.41</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 2 copies</td>
<td>495.15</td>
<td>535.99</td>
<td>719.25</td>
<td>1128.70</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 3 copies</td>
<td>496.39</td>
<td>534.56</td>
<td>760.35</td>
<td>1131.18</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 4 copies</td>
<td>498.90</td>
<td>533.05</td>
<td>721.17</td>
<td>1130.20</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 5 copies</td>
<td>496.47</td>
<td>534.97</td>
<td>720.11</td>
<td>1136.44</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss, 6 copies</td>
<td>494.39</td>
<td>533.18</td>
<td>718.54</td>
<td>1097.69</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit [e]</td>
<td>.46</td>
<td>.47</td>
<td>.55</td>
<td>.74</td>
</tr>
<tr>
<td>Start read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit [f]</td>
<td>.47</td>
<td>.47</td>
<td>.54</td>
<td>.80</td>
</tr>
<tr>
<td>End read [g]</td>
<td>.53</td>
<td>.52</td>
<td>1.11</td>
<td>.75</td>
</tr>
<tr>
<td>End read [h]</td>
<td>.55</td>
<td>.53</td>
<td>.66</td>
<td>.82</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 0 inv</td>
<td>494.61</td>
<td>529.43</td>
<td>718.78</td>
<td>1125.82</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 1 inv</td>
<td>779.87</td>
<td>757.98</td>
<td>944.50</td>
<td>1349.58</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 2 inv</td>
<td>795.26</td>
<td>853.24</td>
<td>1099.32</td>
<td>1449.48</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 3 inv</td>
<td>830.59</td>
<td>851.55</td>
<td>1037.21</td>
<td>1378.69</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 4 inv</td>
<td>833.61</td>
<td>851.68</td>
<td>1034.52</td>
<td>1352.17</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 5 inv</td>
<td>832.27</td>
<td>850.08</td>
<td>1032.21</td>
<td>1348.92</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss 6 inv</td>
<td>832.51</td>
<td>954.37</td>
<td>1033.67</td>
<td>1414.22</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>modify</td>
<td>465.57</td>
<td>543.79</td>
<td>468.02</td>
<td>45.80</td>
</tr>
<tr>
<td>Start write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td>.49</td>
<td>.50</td>
<td>.51</td>
<td>.67</td>
</tr>
<tr>
<td>End write</td>
<td>3.55</td>
<td>3.55</td>
<td>3.72</td>
<td>3.92</td>
</tr>
</tbody>
</table>

Table 4.2: Latencies on OpenBSD with TCP
<table>
<thead>
<tr>
<th>event</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map miss</td>
<td>647.17</td>
<td>683.48</td>
<td>656.47</td>
</tr>
<tr>
<td>Map hit [a]</td>
<td>.44</td>
<td>.44</td>
<td>.48</td>
</tr>
<tr>
<td>Map hit [b]</td>
<td>.37</td>
<td>.41</td>
<td>1.44</td>
</tr>
<tr>
<td>Unmap [c]</td>
<td>.12</td>
<td>.17</td>
<td>.26</td>
</tr>
<tr>
<td>Unmap [d]</td>
<td>.31</td>
<td>.31</td>
<td>.35</td>
</tr>
<tr>
<td>Start read miss, 0 copies</td>
<td>801.48</td>
<td>786.9</td>
<td>1004.10</td>
</tr>
<tr>
<td>Start read miss, 1 copies</td>
<td>578.24</td>
<td>796.10</td>
<td>963.94</td>
</tr>
<tr>
<td>Start read miss, 2 copies</td>
<td>701.12</td>
<td>803.11</td>
<td>1028.57</td>
</tr>
<tr>
<td>Start read miss, 3 copies</td>
<td>810.92</td>
<td>814.66</td>
<td>994.83</td>
</tr>
<tr>
<td>Start read miss, 4 copies</td>
<td>729.68</td>
<td>814.42</td>
<td>1059.53</td>
</tr>
<tr>
<td>Start read miss, 5 copies</td>
<td>717.09</td>
<td>712.97</td>
<td>1005.45</td>
</tr>
<tr>
<td>Start read miss, 6 copies</td>
<td>726.29</td>
<td>856.65</td>
<td>1095.35</td>
</tr>
<tr>
<td>Start read hit [e]</td>
<td>10.21</td>
<td>9.42</td>
<td>9.75</td>
</tr>
<tr>
<td>Start read hit [f]</td>
<td>14.34</td>
<td>12.82</td>
<td>12.21</td>
</tr>
<tr>
<td>End read [g]</td>
<td>9.84</td>
<td>9.49</td>
<td>8.97</td>
</tr>
<tr>
<td>End read [h]</td>
<td>13.07</td>
<td>12.26</td>
<td>12.55</td>
</tr>
<tr>
<td>Start write miss 0 inv</td>
<td>737.74</td>
<td>657.25</td>
<td>944.86</td>
</tr>
<tr>
<td>Start write miss 1 inv</td>
<td>1962.3</td>
<td>1538.43</td>
<td>3688.22</td>
</tr>
<tr>
<td>Start write miss 2 inv</td>
<td>2043.27</td>
<td>2155.89</td>
<td>1614.53</td>
</tr>
<tr>
<td>Start write miss 3 inv</td>
<td>1459.50</td>
<td>1765.79</td>
<td>2088.43</td>
</tr>
<tr>
<td>Start write miss 4 inv</td>
<td>1672.78</td>
<td>2314.45</td>
<td>1521.35</td>
</tr>
<tr>
<td>Start write miss 5 inv</td>
<td>1289.21</td>
<td>2243.74</td>
<td>2041.07</td>
</tr>
<tr>
<td>Start write miss 6 inv</td>
<td>1855.71</td>
<td>1929.92</td>
<td>1829.14</td>
</tr>
<tr>
<td>Start write modify</td>
<td>850.02</td>
<td>780.46</td>
<td>869.25</td>
</tr>
<tr>
<td>Start write hit</td>
<td>9.47</td>
<td>9.60</td>
<td>9.60</td>
</tr>
<tr>
<td>End write</td>
<td>13.27</td>
<td>12.12</td>
<td>12.82</td>
</tr>
</tbody>
</table>

Table 4.3: Latencies on OpenBSD with UDP
<table>
<thead>
<tr>
<th>event</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map miss</td>
<td>130.83</td>
<td>129.99</td>
<td>132.56</td>
<td>133.84</td>
</tr>
<tr>
<td>Map hit [a]</td>
<td>.39</td>
<td>.37</td>
<td>.39</td>
<td>.27</td>
</tr>
<tr>
<td>Map hit [b]</td>
<td>.31</td>
<td>.31</td>
<td>.44</td>
<td>.69</td>
</tr>
<tr>
<td>Unmap [c]</td>
<td>.12</td>
<td>.13</td>
<td>.18</td>
<td>.35</td>
</tr>
<tr>
<td>Unmap [d]</td>
<td>.26</td>
<td>.33</td>
<td>.34</td>
<td>.51</td>
</tr>
<tr>
<td>Start read miss, 0 copies</td>
<td>153.67</td>
<td>199.97</td>
<td>399.48</td>
<td>899.27</td>
</tr>
<tr>
<td>Start read miss, 1 copies</td>
<td>151.13</td>
<td>201.02</td>
<td>400.46</td>
<td>900.74</td>
</tr>
<tr>
<td>Start read miss, 2 copies</td>
<td>152.40</td>
<td>202.07</td>
<td>402.13</td>
<td>901.53</td>
</tr>
<tr>
<td>Start read miss, 3 copies</td>
<td>151.55</td>
<td>198.93</td>
<td>401.74</td>
<td>782.55</td>
</tr>
<tr>
<td>Start read miss, 4 copies</td>
<td>148.54</td>
<td>200.37</td>
<td>399.56</td>
<td>850.24</td>
</tr>
<tr>
<td>Start read miss, 5 copies</td>
<td>151.25</td>
<td>191.01</td>
<td>404.40</td>
<td>916.67</td>
</tr>
<tr>
<td>Start read miss, 6 copies</td>
<td>151.24</td>
<td>197.84</td>
<td>400.64</td>
<td>902.50</td>
</tr>
<tr>
<td>Start read hit [e]</td>
<td>.19</td>
<td>.92</td>
<td>1.63</td>
<td>1.11</td>
</tr>
<tr>
<td>Start read hit [f]</td>
<td>1.82</td>
<td>2.04</td>
<td>2.79</td>
<td>3.05</td>
</tr>
<tr>
<td>End read [g]</td>
<td>.29</td>
<td>.83</td>
<td>.53</td>
<td>2.36</td>
</tr>
<tr>
<td>End read [h]</td>
<td>3.83</td>
<td>4.12</td>
<td>3.20</td>
<td>4.19</td>
</tr>
<tr>
<td>Start write miss 0 inv</td>
<td>149.45</td>
<td>200.16</td>
<td>406.59</td>
<td>888.66</td>
</tr>
<tr>
<td>Start write miss 1 inv</td>
<td>220.97</td>
<td>271.84</td>
<td>485.07</td>
<td>932.01</td>
</tr>
<tr>
<td>Start write miss 2 inv</td>
<td>243.49</td>
<td>286.90</td>
<td>477.84</td>
<td>898.10</td>
</tr>
<tr>
<td>Start write miss 3 inv</td>
<td>236.13</td>
<td>287.62</td>
<td>476.09</td>
<td>900.17</td>
</tr>
<tr>
<td>Start write miss 4 inv</td>
<td>240.14</td>
<td>287.86</td>
<td>476.50</td>
<td>895.04</td>
</tr>
<tr>
<td>Start write miss 5 inv</td>
<td>239.80</td>
<td>287.47</td>
<td>477.79</td>
<td>†</td>
</tr>
<tr>
<td>Start write miss 6 inv</td>
<td>242.43</td>
<td>284.73</td>
<td>477.03</td>
<td>†</td>
</tr>
<tr>
<td>Start write modify</td>
<td>135.11</td>
<td>135.35</td>
<td>133.64</td>
<td>†</td>
</tr>
<tr>
<td>Start write hit</td>
<td>0.81</td>
<td>0.05</td>
<td>1.56</td>
<td>†</td>
</tr>
<tr>
<td>End write</td>
<td>3.18</td>
<td>1.52</td>
<td>1.45</td>
<td>†</td>
</tr>
</tbody>
</table>

Table 4.4: Latencies (in microseconds) on XOK
The most interesting comparison is between the TCP and XOK times for operations requiring communication. (These are the Map miss, Start read miss with 0 – 6 copies, and Start write misses with 0 – 6 invalidations required.) The read misses and write miss with no invalidations each require only a single message-acknowledgment pair to be exchanged. For 64 byte payloads, these operations take about 500 microseconds with TCP on OpenBSD compared to 150 microseconds on XOK, a difference of a factor of three. Clearly, the lighter weight protocol, lighter weight interrupt mechanisms, and lighter weight system calls offer some advantage versus the straightforward TCP implementation.

Bigger improvements are seen in the cases where multiple invalidations need to be made. The case with 2 invalidates requires communication with all four nodes. In this case, we see that for a 1024 byte packet, TCP requires 1100 microseconds to complete the CRL operation. However, the XOK implementation accomplishes the same thing in under 500 microseconds. This exchange requires sending at least 6 protocol messages, and thus 12 Ethernet packets (see diagram for the message order). Both operating systems send the same messages, but the Exokernel clearly wins in processing time.

The XOK version of CRL is still faster for large message sizes. Unfortunately, the factor of three performance goes away; for a 1024 byte payload, TCP requires approximately 725 microseconds round-trip, compared to 400 for XOK. But, the difference of 325 microseconds is the same as for much smaller packets; clearly the Exokernel advantage is lower overhead. The 4096-byte packet measurement requires not one but three underlying Ethernet packets to transmit. In this case, the gap narrows slightly, but XOK is still faster. One reason why the UNIX version does slightly better is because packet fragmentation is done within the kernel, so fewer system calls are required with TCP/IP. However, the XOK measurement is still competitive; this indicates that for larger packet sizes, the network bandwidth is often the bottleneck, while for small packet sizes, it is the operating system latency.

### 4.1.3 Synthetic Benchmarks

In tables 4.6 and 4.5 we see the results of the synthetic benchmark for the TCP/IP version of CRL, and in 4.8 and 4.7 we see the results of the synthetic benchmark of XOK. The synthetic benchmark mixes random calls on CRL regions with a synthetic workload which simply reads the contents of a region between messages. These synthetic benchmarks are the best too to analyze what types of applications will benefit the most from being run with CRL/XOK, and also show the efficiency improvements achieved in the XOK version. The CRL message rate is a more useful performance measure than the CRL bandwidth, because it takes into account both the network bandwidth and the network latency, and combines these in a way which can be easily applied to analyze applications. These measurements also allow us to gauge what granularity problems will run efficiently on this
Table 4.5: CRL message rates with TCP/IP and indexing by Region ID

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Fraction of Read Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Regions</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>

The benchmark program randomly executes and times 1024 CRL operations on each node and divides the number of operations performed by the running time to give a CRL operation rate. The synthetic benchmark numbers are given as rates; in the number of CRL operations per second. Thus, bigger numbers are better. We provide two sets of numbers for each platform. The first set specifies region operations by using region identifiers, and includes the rates for mapping and unmapping regions. The second set of region operations keeps the regions mapped during the benchmark. We give data for four different region sizes, and four different numbers of regions. Additionally, the benchmark program randomly mixes read and write operations. The horizontal axis on each chart corresponds to these differences.

The synthetic benchmark data helps us characterize what types of applications would benefit the most from the Exokernel implementation. In figures 4-1, 4-2, and 4-3, we have plotted the benchmark results for two different region sizes and two different numbers of regions. There are some general trends worth noting. First, read operations are cheaper than write operations, because read operations do not require invalidations, and regions can be shared during read operations. The horizontal axis on each chart corresponds to these differences.

Secondly, rates are faster for smaller region sizes. This follows from the lower transmission times required for smaller packets. Finally, the most curious trend is that for write operations, smaller numbers of regions give higher rates, but for read operation, larger numbers of regions give better rates. The reason for this effect is because a node wishing to start a read operation needs to obtain an exclusive copy of the region. If there are fewer regions, then the probability that a given node has an exclusive copy when it begins the operation is much greater. Thus, it takes less time to begin read operations because the probability that network communications are required decreases.
### Problem Size

<table>
<thead>
<tr>
<th>Size</th>
<th>Regions</th>
<th>Fraction of Read Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>1359.2 1340.0 1305.5 1322.4 1590.0 3050.7 3568.6 6338.2 12153.2 22407.2</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>1281.3 1245.5 1224.9 1253.4 1478.2 2857.6 3305.5 6184.2 12835.2 24111.4</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>1242.9 1258.5 1191.7 1216.5 1455.5 3782.8 4556.8 9034.4 17954.5 33428.7</td>
</tr>
<tr>
<td>16</td>
<td>1024</td>
<td>997.8 1159.9 1534.5 1954.0 3372.4 9897.7 11542.3 21607.4 39282.2 65507.7</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>1295.7 1283.2 1321.8 1542.7 2940.8 3531.8 5988.0 11688.6 21921.2</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>1297.3 1267.5 1252.4 1292.4 1496.5 2975.2 3456.3 6383.0 13077.1 24357.3</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>1245.4 1250.7 1184.6 1216.0 1439.9 3714.2 4605.6 8881.7 17759.8 33042.4</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>975.5 1156.7 1517.7 1934.9 3309.6 9276.2 11430.5 21334.2 38296.0 63129.9</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>1267.4 1267.5 1221.0 1293.5 1494.2 2860.4 3425.1 6074.6 11341.5 20504.4</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>1250.1 1238.5 1226.3 1271.8 1480.3 2833.7 3331.0 6157.3 12578.0 22995.6</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>1206.3 1218.4 1159.6 1195.4 1409.8 3620.9 4523.9 8542.0 16987.0 31086.6</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>958.3 1118.7 1478.1 1901.8 3236.4 8830.1 10866.1 19763.5 34940.5 54459.9</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
<td>1030.8 1035.8 1025.3 1087.9 1291.6 2466.9 2942.6 5049.6 9360.7 16516.4</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>1037.0 1051.0 1037.2 1081.6 1275.7 2476.7 2968.6 5190.7 10269.0 18589.6</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
<td>998.4 1017.5 986.9 1007.2 1220.6 3116.6 3797.9 6976.1 13082.3 22335.0</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>805.5 951.2 1233.4 1590.2 2719.5 7238.7 8718.9 15255.3 24685.8 35278.8</td>
</tr>
</tbody>
</table>

Table 4.6: CRL message rates with TCP/IP and indexing by Region Pointer

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Regions</th>
<th>Fraction of Read Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>5012.0 4967.8 4026.2 4660.2 5453.0 9922.4 12510.4 20810.3 36791.8 67445.2</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>4762.5 693.5 4709.8 4883.4 5016.9 10566.3 11795.4 20222.1 32762.7 60847.9</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>4107.8 4137.4 4163.8 4340.7 4671.0 12017.1 15553.7 24075.9 47027.7 81530.89</td>
</tr>
<tr>
<td>16</td>
<td>1024</td>
<td>2580.7 2632.6 3114.3 3941.2 5923.7 15768.5 18527.5 32629.0 48159.1 81553.2</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>4790.7 4701.6 3776.0 3942.3 5217.1 10176.8 12707.4 20338.2 36680.0 65244.3</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>4614.1 4871.3 4540.3 4700.0 5124.9 10338.9 10984.9 20312.3 35271.4 62970.5</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>3951.4 3955.5 3988.9 4215.0 5000.0 12618.8 13989.5 27549.5 49248.5 78094.1</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>2496.4 2549.3 2971.4 3682.1 5714.7 14903.0 17285.3 26319.7 48735.2 72996.4</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>3751.7 3687.6 3538.6 3537.2 4577.0 8136.7 9681.4 16859.1 29536.8 49512.2</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>3865.2 3586.4 3609.5 3537.2 4617.2 7059.6 10187.2 18029.8 30507.4 55681.3</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>3200.5 3243.0 3284.9 3431.9 4009.1 12199.7 13214.6 22445.7 37180.1 57546.3</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>2101.4 2150.2 2433.7 3179.2 4720.8 12388.9 147839.9 25697.4 42060.0 62919.4</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
<td>1848.4 1853.7 1877.2 1970.0 2347.4 4702.7 5448.0 9552.2 16971.9 †</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>1840.4 1819.3 1875.9 1987.2 2451.9 4654.1 5494.3 9515.9 16975.9 28000</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
<td>1660.7 1691.6 1719.2 1798.5 2220.9 4731.7 6733.1 12427.5 21367.3 33735.7</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>12014.4 1223.0 1367.1 1757.8 2444.7 6968.7 8388.4 14721.3 24154.0 35332.1</td>
</tr>
</tbody>
</table>

Table 4.7: CRL message rates with XOK and indexing by Region ID
Table 4.8: CRL message rates with XOK and indexing by Region pointer

Problem Size | Regions | Fraction of Read Operations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Regions</td>
<td>0.00</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>5166.3</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>5028.4</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>4852.7</td>
</tr>
<tr>
<td>16</td>
<td>1024</td>
<td>3819.4</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>4891.2</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>4823.4</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>4629.4</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>3623.6</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>3839.5</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>3816.3</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>3668.7</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>2843.8</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
<td>1834.0</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>1865.5</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
<td>1754.6</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1396.5</td>
</tr>
</tbody>
</table>

Figure 4-1: TCP and XOK synthetic benchmark results for sixty-four 64-byte regions
Figure 4-2: TCP and XOK synthetic benchmark results for two hundred fifty six 64-byte regions

Figure 4-3: TCP and XOK synthetic benchmark results for two hundred fifty six 1024-byte regions
Table 4.9: Times to complete the traveling salesman problem application, measured in seconds

<table>
<thead>
<tr>
<th></th>
<th>One Node</th>
<th>Two Nodes</th>
<th>Four Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>1.794</td>
<td>1.870</td>
<td>3.210</td>
</tr>
<tr>
<td>XOK</td>
<td>1.295</td>
<td>.660</td>
<td>.281</td>
</tr>
</tbody>
</table>

Comparing the data between the TCP and XOK versions, we notice that the XOK version manages to obtain significantly higher message rates across the entire set of tests. This is what we expect, given the much lower message sending overhead we saw in the micro-benchmarks. However, the gap between the two systems is much greater for small region sizes, where the XOK system achieves three times the data rate, than for large region sizes. This implies that CRL should perform much better on the XOK version because the majority of CRL messages are small (about 20 bytes). Additionally, it implies that many applications like Barnes-Hut, which uses a large number of small regions to communicate, should perform much better as well.

To understand what is going on, we will use a simple model of the effective transmission latency between two nodes. The latency is given by the following equation:

\[ t_L = t_o + \frac{s}{r} \]

Here, we let \( t_L \) be the latency, \( t_o \) be the overhead cost associated with each system, \( s \) be the message size in bits, and \( r \) be the number of bits which can be transferred every second (the bandwidth). For small messages, the overhead of the system dominates this latency. For large messages, the message size dominates the latency relation.

Thus, we expect that the XOK version of CRL will most help applications with small data sets requiring frequent communication—latency-bound applications. For applications requiring large blocks of data to be moved around—bandwidth bound applications—we expect that the XOK version of CRL will provide much smaller benefits.

4.1.4 TSP Application

In table 4.9, we see the results of running the TSP application with 12 cities on two and four machines. The application allocates one node to work distribution, and the other nodes to doing work, so we would expect that the four node version would run approximately three times as fast as the two node version.

The XOK application clearly takes much less time than the OpenBSD version of this application. In fact, the XOK application appears to achieve a super-linear speedup, taking less than 1/4 the time to run on four nodes than it took on one node. Unfortunately, it is not clear why this occurs; it is possible that the timers used for either application did not accurately time the application (it’s
In tables 4.11, 4.10, and 4.12, we see the results of running the LU decomposition kernel on each platform. We ran the LU application on three different matrix sizes and three different block sizes, and present the data for each of these. What we notice is that communications costs are a large problem for the UDP version of these applications; the performance on two nodes is essentially the same as for a single node. The TCP application performs better, but it still runs more slowly in parallel than it does on a single machine.

The XOK results are very interesting. We achieve what look like good speed-ups as we increase the number of nodes. But, the LU program runs much more slowly on the Exokernel than on OpenBSD. Even on a single node, the program took about four times as long to run. To figure out why this occurred, we compiled a special version of the XOK kernel which allowed user processes to manipulate the Pentium performance counters and measure several important pieces of data on each version. We used a version of the LU program which made no library or system calls during the timed portion of the code (and hence ran on a single node).
Table 4.12: LU Decomposition on various problem sizes on XOK

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Block Size</th>
<th>1 Node</th>
<th>2 Nodes</th>
<th>4 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>16</td>
<td>.794</td>
<td>.510</td>
<td>.357</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
<td>.734</td>
<td>.475</td>
<td>.334</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>.736</td>
<td>.543</td>
<td>.402</td>
</tr>
<tr>
<td>512</td>
<td>16</td>
<td>5.776</td>
<td>3.440</td>
<td>2.068</td>
</tr>
<tr>
<td>512</td>
<td>32</td>
<td>5.367</td>
<td>3.173</td>
<td>1.855</td>
</tr>
<tr>
<td>512</td>
<td>64</td>
<td>5.819</td>
<td>3.596</td>
<td>2.242</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
<td>46.531</td>
<td>25.373</td>
<td>14.231</td>
</tr>
<tr>
<td>1024</td>
<td>32</td>
<td>43.476</td>
<td>23.622</td>
<td>13.425</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>46.217</td>
<td>25.851</td>
<td>15.604</td>
</tr>
</tbody>
</table>

It looked like some sort of memory problem, so we measured resource stalls, instruction length decoder stalls, instruction unit cache misses, memory stalls, tlb stalls, and misaligned memory references. We ran the lu program with a matrix size of 1024 and a block size of 32 for the measurements.

We found that on OpenBSD about 194 misaligned memory references were made, while on XOK 943126854 misaligned references occurred. Inspection of the XOK malloc procedure revealed the problem: the malloc consumed the first four bytes of every memory region allocated. Hence, when the user's program referenced a large region, it was given a pointer \( x \) to memory where \( x \equiv 4 \pmod{8} \), while doublewords are given by \( x \equiv 0 \pmod{8} \). Modifying the malloc procedure allowed us to achieve an even better running time than the OpenBSD version (9.402 seconds in one test). Unfortunately, time restrictions precluded running the lu applications again to get more accurate measurements.

4.1.6 Barnes-Hut Application

The Barnes-Hut program performed poorly in parallel on the UNIX platforms. The program appears to run better in parallel on XOK (for all problem sizes), but the sequential times on the platforms are very different. Because of the anomalies with sequential running times, the results for Barnes-Hut are omitted from this thesis. The problems were probably due to the floating point errors described above; we simply masked floating point exceptions for the XOK run, but when we masked them on OpenBSD, they were caught by UNIX signal mechanisms. It would be interesting to see how these programs perform once the floating-point exception issues are resolved.

4.2 Discussion

The results presented here indicate that the performance of CRL on XOK is much better than the performance of CRL on a UNIX system. The ability to send and receive messages with very low overhead make CRL operations much faster. The advantages of the XOK version are not as great
for large messages as for small messages. However our goal was to build a system for fine-grained computation, so small messages are likely to be the more important case.

The synthetic benchmarks show us that more communication intensive applications will benefit greatly from the XOK version. The Exokernel implementation allows CRL events to occur at a rate often two or three times as great as on UNIX, and the latency of each event is much lower. This indicates that finer-grained applications requiring more frequent communication can be run efficiently on the XOK version of CRL. Moreover, it is interesting that some sequential applications appear to run more quickly on the Exokernel. When the Exokernel has better support for floating point exceptions, and better memory allocation facilities are implemented, it will be interesting to see how some other test applications run.
Chapter 5

Conclusions

Although we had enormous trouble getting applications to work correctly or efficiently on this environment, the preliminary results that we have obtained look very promising. Our straightforward implementation of Active Messages allows us to send small messages in as little as 65 microseconds. This is much better than the performance of OpenBSD on small messages, and is a good indicator of the performance increases that we would expect in a more robust and polished DSM system built on an extensible operating system. Our measurements of CRL on basic benchmarks have shown that we can achieve a much higher message rate with XOK, and that each CRL event can be completed more quickly.

Moreover, we have shown that there can be advantages in tightly integrating a user level protocol like CRL with a message passing protocol. Although we could give user processes control of the networking hardware on a normal operating system, or integrate some of the functionality we implemented into a normal operating system, the level of integration of our system could only be implemented securely with an extensible operating system. By giving the application secure control of system resources, we achieve the security and safety benefits of in-kernel abstractions with the performance of user-level primitives.

5.1 Design Mistakes

In evaluating the implementation of CRL for the XOK Exokernel, it became clear that there were several mistakes in the design of this system. The most important mistakes were the lack of flow control, the use of course-grain locking, the implementation of protocols optimized for Ethernet, and the lack of integration between the active message and CRL layers. Many of these mistakes would have been difficult to foresee before building a simpler implementation, but making these changes is likely to improve the performance of CRL on XOK.
5.1.1 Flow Control

The first important mistake was the lack of a flow-control implementation. This became a problem with some applications, such as the Fast Fourier Transform application, which require large numbers of communications with a single node. We discovered that CRL can fairly easily fill one node's buffers with incoming data messages (on flush operations), at a much faster rate than the node can take. At present, there is no mechanism for an application to figure out the source of dropped packets through packet rings. We could have solved this problem in one of two ways. First, we could add several protocol messages to CRL to warn the receiving node that it is about to receive a lot of data, and give the destination node the opportunity to delay the transfer. Second, we could have implemented the flow control at the ASH level. This would have been slightly more expensive than packet rings, but might have solved the problem.

5.1.2 Interrupt Masking

Both the CRL and XOK code need to mask interrupts at certain times to maintain the consistency of data structures. Additionally, the code also turns off interrupts and explicitly polls for messages at certain points to save on the cost of interrupts\(^1\). The performance of CRL depends heavily on the implementation of efficient upcalls for unexpected interrupts, but performs slightly better if it polls for expected interrupts.

The CRL and Active Message code both mask interrupts at different times to maintain data integrity. But, they don't directly share data structures; both libraries maintain some logically separate state information. Thus, it seems like it might be beneficial to mask different types of interrupts: upcalls and active messages. When CRL masks interrupts, it usually wants polling to be cheap. But sometimes, it just wants to make a state change atomic. The same is true for the active message layer. We could probably get higher network throughput (and cut down on the number of unacknowledged packets) by allowing upcalls to occur at some points when we don't want active messages.

5.1.3 Poor CRL, Active Message Integration

The current implementation does not fully exploit some redundancy in the CRL and Active Message protocols. For example, most CRL messages require an explicit acknowledgment; the sending node needs to know that the receiver has not just received, but has processed a message. The Active Message protocol acknowledges each message in its current implementation. This reduces the number of network messages needed for many operations.

\(^1\)This is part of CRL's CM-5 legacy. On the CM-5, interrupts could be very expensive, so polling could result in big performance gains [33]. On the XOK implementation, this saves us a context switch, and thus several hundred cycles. This isn't a big improvement with Fast Ethernet, but might be advantageous for much higher speed networks.
For example, consider what happens when a remote node sends a MsgSharedRequest packet (to ask for read permission) to a home node with the only copy of the region. In the normal case, the requesting node encapsulates the message in an active message packet and sent to the home node. The home node receives the active message packet, and explicitly acknowledges the message. The home node then calls a handler to process the message. The handler procedure then sends a copy of the region data to the remote node as an active message. Finally, the remote node receives the packet (or packets) containing the data, and calls a handler procedure to accept the data. This exchange requires four separate messages.

We can easily eliminate a packet in this exchange: the explicit acknowledgment of the MsgSharedRequest message by the active message layer is superfluous. Implementing this change would be straightforward; we add an optional flag to the am_send procedure to specify that the receiving active message layer should not acknowledge the packet. (There is an exception to this rule: the receiver should acknowledge the packet if it can not be processed right away. This happens in two cases: first, if their are queued active messages, then the receiver should send an acknowledgment. Second, if the message is blocked (because a remote node has an exclusive copy, for example), then an acknowledgment also needs to be sent. The first acknowledgment is easily accomplished from the active message layer, but the second requires the CRL code to ask the active message layer to acknowledge the packet.)

A second optimization that could be made involves where incoming messages are placed. When a node expects a large message (containing a region's contents, for example), it should be possible for the active message layer to copy these contents directly, without copying the data and storing it in a queue.

5.1.4 Bad Broadcast and Reduction Protocols

The broadcast and reduction protocols I implemented were designed to work well over a shared Ethernet. But, our current implementation uses a network switch and a small cluster of machines. A more standard protocol would probably result in fewer messages and more reliable and consistent performance on such a network topology. More important than cutting the number of messages is cutting the number of processor interrupts. The Exokernel communication architecture is cheap, but it isn't so cheap that getting rid of unnecessary interrupts would not help performance.

Broadcast based reduction are not an efficient way to implement reductions. In the best case, this scheme requires only $n$ packets to be sent out for $n$ nodes. However, our experiments have indicated that most programs send out more than twice this number. The reasons for this is probably poor synchronization. A better way to implement global reductions is to nominate one node as a record-keeper. Using the reliable active message protocol, each node sends a message to the record keeping node, and the record keeping node keeps a count of the number of messages it has received. When
the record-keeper has received a message from every node, it broadcasts a message telling each node that it is OK to pass the barrier.

This scheme has two apparent disadvantages over the scheme we implemented. First, a minimum of \(2n - 1\) packets need to be sent. However, a major advantage of this scheme is that \(2n - 2\) of these packets interrupt only the designated record node, or the node which has reached the reduction; each node receives an average of 3 interrupts per reduction operation. A second problem with this scheme is that it places a large burden on a single node. However, we could easily circumvent this problem by rotating through the nodes, and choosing a different node as the record-keeper on each iteration. (A simple way to do this would be to choose node \(e \mod n\) as the record-keeper, where \(e\) is the current epoch, and \(n\) is the number of nodes.) Using this scheme, each node receives an average of 3 interrupts per reduction operation amortized over the running time of the program.

5.1.5 Overly General Reliable Data Protocol

Go-back-n ARQ is an easy to implement and robust protocol. But, it isn't the best choice for our environment. On a LAN, packet losses rarely occur because of errors. In our test implementations, they usually only occurred when sending or receiving buffers became filled with packets. Moreover, even though message sending is not expensive, it still takes about 65 microseconds (which is about 1100 clock cycles) to send a message. Given that packet losses are infrequent, and that processor interrupts and message transmission are expensive, a more optimistic protocol would probably result in better network utilization, and better CRL performance.

For example, our implementation uses a ring of pointers that point to specific messages that were sent. It might be better to simply write all messages that we sent to a (fixed size) log. It would be harder to find the beginning of a specific message in a log, but if retransmissions were rare then this might not be a problem. (And, it would be straightforward to implement optimizations, like a second log that pointed to specific messages.) It would also be necessary to periodically reclaim pieces of the log corresponding to acknowledged packets, but this too could be easily and inexpensively accomplished.

Another improvement might be Selective Repeat ARQ. With Selective Repeat ARQ, the sender only retransmits negatively acknowledged packets. This optimization only helps if we lose packets frequently, or use a large retransmission delay of window size. Because a large window size and retransmission delay are desirable (they allow us to cut down on the number of acknowledgments), Selective Repeat ARQ would be a worthwhile change to add.
5.2 Directions for Future Work

We have managed to build a reasonably fast and usable system for parallel programming on a network of workstations. But, we have also learned a good deal about why parallel programming is hard, and why building parallel software systems is hard.

5.2.1 CRL Efficiency

The current implementation is somewhat unpolished, and a number of data structure improvements could be made to reduce the space and time requirements of the active message layer. Specifically, the rebroadcast queue is currently overly simplistic; there is a link in the queue corresponding to every unacknowledged packet. We could probably improve the efficiency of the program by eliminating this queue, and instead keeping the retransmission times within each link. Instead of traversing a list of unacknowledged packets, we could either iterate through each link info structure (if we knew that there were a very small number of nodes), or simply keep a short list of links which need to be checked.

5.2.2 Additional Operations

There are several additional operations which we believe would help improve application performance at little cost in additional complexity. The first desirable change would be a technique for merging requests into single packets. A programmer often wants to obtain several different regions at the same time, and these regions are often on the same node. Rather than sending 5 pairs of messages to lock five regions, it would be straightforward to merge these 5 pairs of messages into a single pair! (This would also give the CRL software an opportunity to make sure that a locking order is enforced, to avoid deadlocks). To do this, we would modify the rgn_start_op and rgn_end_op routines so that they were presented with a list of operations which we wished to perform, rather than a single operation. This simple technique could cut down on the number of packets transmitted by a large amount.

Another technique that could be helpful would be a technique to split CRL computations into more distinct threads. Rather than looping aimlessly while waiting for a request to be answered by another node, it would be useful if we could simply switch to a different thread and continue to compute. Without a significant gain in complexity, and using only a small number of threads on each node, we may be able to increase our processor utilization to almost 100%.

The measurements that we have taken indicate that, despite the fact that we have cut latency by a factor of five compared to UNIX systems, the latencies inherent in a Fast Ethernet network are still too high for many of our purposes. It might be worthwhile to further streamline our message sending code (possibly be caching more information in outgoing buffers), but the problem is mostly
a hardware problem. It is nice that processors are getting faster (the CM-5 used 33 Mhz Sparc processors, while we used 200 Mhz Pentium pros), but processor performance improvements simply make it more essential to get higher performance. While a 170 microsecond round trip latency on the CM-5 meant losing about 5000 cycles on a 33 Mhz processor, it means losing 30000 cycles on a 200 Mhz processor! To get really good performance, we should probably explore the benefits of switching to a better network topology, like Myranet for our DSM research.

5.2.3 CRL Maintainability

A multithreaded model would also be an improvement over the current interrupt-driven model for CRL. The CRL code would be clearer and more easily understandable if we were to structure it as a set of work threads, and a single memory-management thread. As CRL is currently implemented, an executing thread of computation is preempted by an interrupt handler. Because we can perform this context switch cheaply on network arrivals anyway, implementing cheap user level threads would neither be difficult nor overly detrimental to performance.

5.2.4 CRL Programming Model

There are several refinements which would be useful to make CRL more usable. First, it would be nice if we had a good system to manage launching and aborting processes on remote machines. PVM provides a good model for the interface we would like, but there is a lot of overhead associated with their interface, and it looks like porting PVM to XOK would be a great deal more work than would be worthwhile.

Secondly, a better system for work division would be helpful. The task of dividing work among many nodes is extremely complex and difficult. It would be interesting to see what would happen if the distributed shared memory capabilities of CRL were merged with the scheduling capabilities of CILK [12]. The CILK language, developed by Leiserson, Blumofe, and many others, is a language designed for efficiency multithreaded programming. It can also be proven that the run-time system that is used with CILK can be run within a factor of two of the optimal time! The CILK programming model is somewhat limited in the types of algorithms it can express, and is probably not the best solution for all programming problems. However, for many if not most applications, the combination of a good language for describing parallel programs, and a good system for managing shared memory would make parallel programming almost as easy as sequential programming, and would give the programmer an extremely elegant way to program an ugly set of machines. The programmer would be released from writing complex protocols to manage the operation of her program, and could instead concentrate on algorithmic issues.
 Appendix A

Protocol Details

In this section, we will describe the active message protocol more carefully using a precon- 

dition-effect style, as in [40]. We also argue very briefly as to the correctness of these algorithms (though 

not their implementations). This formal description of the algorithms is designed to supplement the 

informal description in the design section of this document.

A.1 Protocols

Global Information

States:

- `my_id`, a unique identifier for each node
- `num_nodes`, the total number of nodes
- `N`, the set of all nodes

A.1.1 Broadcasts

States:

- `eb`, the broadcast epoch,
- `Rb`, the set of nodes who received the last broadcast,
- `broadcasting`, a boolean variable telling whether this node is broadcasting
- `Qb`, the queue of waiting broadcast messages

User calls

- `Broadcast-Send(m)`
  - While (`Rb` \( \neq \) `N`)
    - send `Broadcast(sender, m, e, R)`
  - \( e_b \leftarrow e_b + 1 \)
  - `Rb` \( \leftarrow \emptyset \)

Protocol Messages

- `Broadcast(sender, m, e, R)`
  - `Precondition:`
\( \text{sender} \neq \text{my\_id} \)
\( e = e_b \)

Effect:
- put \( m \) in \( Q_b \)
- \( e_b \leftarrow e_b + 1 \)
- send \text{BroadcastAck} to \text{sender}

\text{Broadcast}(\text{sender}, m, e, R)

Precondition:
- \( \text{sender} \neq \text{my\_id} \)
- \( e = e_b - 1 \)
- \( \text{my\_id} \notin R \)

Effect:
- send \text{BroadcastAck} to \text{sender}

\text{BroadcastAck}(\text{sender}, e)

Precondition:
- \( \text{broadcasting} = \text{true} \)
- \( e = e_b \)

Effect:
- If \( \text{sender} \notin R_b \)
  - Then \( R_b \leftarrow R_b \cup \{\text{sender}\} \)
  - If \( R_b = N \)
    - Then \( e_b \leftarrow e_b + 1 \)
    - \( \text{broadcasting} \leftarrow \text{false} \)

\text{SendBroadcast}(\text{broadcasting})

Precondition:
- \( \text{broadcasting} = \text{true} \)

Effect:
- send \text{Broadcast} message to all nodes

A.1.2 Reductions

States:
- \( e_r \), the reduction epoch
- \( R_r \), the set of reduction messages received
- \( \text{value} \), the computed value
- \( f : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R} \), a commutative function on the real numbers
- \( \text{old\_value} \), the last computed value
- \( \text{waiting} \), a boolean variable indicating whether this node is waiting for messages

User calls

Protocol Messages

\text{Reduction}(\text{sender}, \text{number}, v, v_o, \text{sender}, e, R)

Precondition:
- \( e = e_r \)

Effect:
If $sender \not\in R$
Then $R_e \leftarrow R_e \cup \{sender\}$
value $\leftarrow f(value, number)$
If $my\_id \not\in R$
Then broadcast Reduction message
If $R = R_e$
Then $e_r \leftarrow e_r + 1$

Reduction($sender, v, v_o, sender, e, R$)
Precondition:
\[
e = e_r - 1
\]
Effect:
If $sender + 1 \equiv my\_id \mod n$
Then send Reduction message to sender

Reduction($sender, v, v_o, sender, e, R$)
Precondition:
\[
e = e_r + 1
\]
Effect:
\[
value \leftarrow v_o
\]
\[
e_r \leftarrow e_r + 1
\]

A.1.3 Active Message
The states defined here are the set of states for each link; we eliminate additional indexing to simplify this description. States:
\[
r_{in}, \text{ the next packet we expect to receive}
\]
\[
r_{out}, \text{ the last packet acknowledged by the other side}
\]
\[
s_{out}, \text{ the sequence number of the next packet to be sent}
\]
\[
\text{window} - \text{size}, \text{ the maximum number of unacknowledged packets allowed}
\]
\[
Q_{am}, \text{ a queue of received active messages}
\]

User calls

Protocol Messages

ShortMsg($sender, m, h, size, sn, rn$)
Precondition:
\[
sn = r_{in}
\]
Effect:
\[
\text{append } (m, size, h, size) \text{ to the tail of } Q_{am}
\]
mark the message ready to read
If $rn > r_{in}$
Then $r_{in} \leftarrow rn$

ShortMsg($sender, m, h, size, sn, rn$)
Precondition:
\[
sn \neq r_{in}
\]
Effect:
If $rn > r_{in}$
Then $r_{in} \leftarrow rn$
send a nack to $sender$
LongMsgStart(sender, m, h, size, sn, rn)
Precondition:

\[ sn = r_{\text{ini}} \]

Effect:

append \((m, size, h, size, sn)\) to the tail of \(Q_{am}\)
mark the message ready not ready to read If \(rn > r_{\text{ini}}\)
Then \(r_{\text{ini}} \leftarrow rn\)

LongMsgStart(sender, m, h, size, sn, rn)
Precondition:

\[ sn \neq r_{\text{ini}} \]

Effect:

If \(rn > r_{\text{ini}}\)
Then \(r_{\text{ini}} \leftarrow rn\)
send a nack to \(sender\)

LongMsgContinue(sender, m, h, size, sn, rn)
Precondition:

\[ sn = r_{\text{ini}} \]

Effect:

add \(m\) to the \(Q_{am}[sn]\)
If \(rn > r_{\text{ini}}\)
Then \(r_{\text{ini}} \leftarrow rn\)

LongMsgContinue(sender, m, h, size, sn, rn)
Precondition:

\[ sn \neq r_{\text{ini}} \]

Effect:

If \(rn > r_{\text{ini}}\)
Then \(r_{\text{ini}} \leftarrow rn\)
send a nack to \(sender\)

LongMsgFinish(sender, m, h, size, sn, rn)
Precondition:

\[ sn = r_{\text{ini}} \]

Effect:

add \(m\) to the \(Q_{am}[sn]\)
mark the message ready to read
If \(rn > r_{\text{ini}}\)
Then \(r_{\text{ini}} \leftarrow rn\)

LongMsgFinish(sender, m, h, size, sn, rn)
Precondition:

\[ sn \neq r_{\text{ini}} \]

Effect:

If \(rn > r_{\text{ini}}\)
Then \(r_{\text{ini}} \leftarrow rn\)
send a nack to \(sender\)
MsgNack(sender, sn, rn)
Precondition:

Effect:
If rn > rn_in
   Then rn_in ← rn
Else resend packet rn

A.2 Correctness Arguments

For formal proofs of the correctness of the ARQ protocol, see either [9] or [40]. Here, we do not try to present complete proofs, but instead offer intuitive arguments as to why these protocols work. The arguments are separated into two portions; a liveness argument and a safety argument. The safety argument shows that if a message is received by a user, then it is always correct. The liveness argument shows that any message submitted will eventually be delivered.

We assume here that the underlying protocol may drop packets or deliver packets out of order, but that if a packet is received, then it is error-free. For real networks, this is almost never the case, but it can be true with very high-probability, so we assume this here.

A.2.1 Active Message Protocol

Our active message protocol uses an ARQ system to assure that messages are eventually delivered exactly once and in order. The former property (ARQ safety) is maintained by the sequence numbers. After each sequence number is used, the receiver increments the value it is expecting. Thus, it will never accept the same packet twice. The sequence numbers are ordered, so as long as the sender transmits messages in order, the receiver will receive them in order. ARQ liveness is guaranteed because the probability of packet delivery is strictly greater than zero. This implies that, if neither node crashes, and if the sender continues to send messages, the probability of the receiver never receiving a packet is given by $\lim_{i \to \infty} (1 - p^i)$, where $p$ is the probability of a packet being delivered. Thus, the protocol guarantees that every packet is eventually delivered.

To show that this specific protocol is correct, we argue that this protocol is an implementation of the go-back ARQ specification, because there is an abstraction relation relating go-back ARQ to the active message protocol. The mechanisms to add packets to the protocols and take them out are identical. Each side sends and receives in this case, but this does not affect the correctness as we are essentially running two go-back-n ARQ protocols at the same time. What we need to do is to show that the sequence numbers of both protocols are related, and we are essentially done. In doing so, we consider only the sn value and data for each packet, and ignore the rn values because of our argument that this is two protocols in parallel.
For short packets, there is a one to one correspondence; each active message $sn$ and $rn$ value corresponds to exactly one $sn$ or $rn$ value in the go-back-$n$ ARQ protocol, and the message associated with each message corresponds exactly as well. For long packets, there is a slightly more complicated correspondence. We suppose that a long packet in the active message protocol is given by a sequence of packets $\{(sn_1, m_1), (sn_2, m_2), \ldots, (sn_n, m_n)\}$, and that in the go-back-$n$ ARQ protocol, it is given by $(sn, m)$. We define a correspondence as follows: we map the set of sequence number $\{sn_1, sn_2, \ldots, sn_n\}$ to $sn$ and the concatenated sequence of messages $m_1m_2 \ldots m_n$ to $m$. Hence we have shown a correspondence between the messages in each protocol. Now, we simply have to argue that the active message protocol delivers these messages correctly and in order. First, we argue that the results are safe. The argument for this is the same as for go-back-$n$ ARQ. Each active active message is received in order because the receiver only accepts the next packet in the sequence. Each piece of a message is received exactly once for this reason. For liveness, we note that any packet sent by the network will eventually be delivered with probability strictly greater than zero. Thus, over a very large number of tries, and packet will eventually be delivered to its destination and acknowledged.

A.2.2 Broadcast Protocol

We want to show that the broadcast protocol given here exhibits three properties:

1. No broadcast packet is received more than once by any node.
2. Each node receives the broadcast packets in the same order.
3. Every broadcast packet sent is eventually received by every node.

We will assume that there is only one sending node per epoch, and that every other node is a receiving node. We also do not guarantee that every node is trying to receive broadcasts from the same epoch; it is possible that there are two sending nodes at some point. (However, the protocol will still work because a sending node from the earlier epoch will not acknowledge a packet from a later epoch until all nodes have acknowledged the earlier epoch’s broadcast.)

To show that no broadcast packet is received more than once, we notice that when each node receives a broadcast packet from the expected period (or epoch), it increments the epoch which it is willing to accept, and only accepts packets from the next period. Hence, each packet is delivered only once. Because each broadcast message has an epoch attached to it, and each receiving node is tuned to the same epoch, we know that each node receives all the broadcast packets in the order of the epochs, and hence in the same order. Finally, as above, we assume that the probability of a packet being delivered is given by $p$, where $0 < p \leq 1$. Hence, as the number of rounds approaches infinity, the probability of packet delivery approaches one.
A.2.3 Reduction Protocol

Now, we want to show that the reduction protocol will eventually terminate, and will terminate with the correct value.

1. No node will reach reduction $i + 1$ before every node has reached

2. Each node will compute the same value during a reduction operation.

3. Each node will eventually reach the end of a reduction operation. reduction $i$.

First, we notice that every node will eventually receive a message from every other node during each epoch. No node will move beyond epoch $i$ to $i+1$ until it has received a message from every other node during epoch $i$. Once a node has received a message from every other node during epoch $i$, it knows that every other node has reached epoch $i$, and hence it is safe to move on to the next level.

The reduction function we use is a commutative function, so if every node receives the same set of values, every node can compute the same reduction function. We know that every node will eventually receive a value from every other node during an epoch $i$, because it will receive a message from every other node. Hence, every node will eventually compute an identical reduction value.

It follows that every node will eventually reach the end of a reduction operation if every node eventually enters the pool of nodes computing the reduction. We know that each packet sent will eventually be received with probability one. Hence, we know that eventually every node will receive a message from every other node with probability one. If a node does not receive a reduction message, then it periodically rebroadcasts a message to other nodes asking for the current epoch. Because this node will eventually receive a response with probability one, and because we know that some node eventually receives the reduction message, each node can make progress.
Appendix B

Preliminary Results

Below are the initial performance measurements made of the NOW version of CRL running on a cluster of 12 166 Mhz Pentium PCs running Linux 1.3.84. Absolute running times in seconds and speed-ups have been provided for a variety of problem sizes and numbers of nodes. A “speed-up” is an intuitive way to measure the performance of a parallel system. Speed-up is defined as the amount of time taken by a set of machines operating in parallel divided by the time taken by a single processor machine on the same program. Ideally, a program running on \( n \) processors would execute in \( \frac{1}{n} \) of the time required by a single processor; the speed-up would be \( n \).

This system has been tested on a small pool of PCs connected together by a switched Ethernet. The preliminary results obtained from this small test bed have been disappointing. Here is a summary of the performance results:

B.1 Barnes-Hut

The results of the Barnes-Hut simulation are illustrated in figures B-1 and B-2. The application runs significantly more slowly in parallel than it does on a single node. When run on more than 2 nodes,

![Figure B-1: Barnes-Hut time and speed-up on 16384 bodies](image)

82
the execution time decreases with the number of nodes. There are some benefits to parallelization, but the communication costs are currently too high to run a computation-intensive program like this on the system.

B.2 FFT

For the Fast Fourier Transform benchmark, the results were also disappointing. (See figures B-3 and B-4.) There is a large increase in computational time when switching from single processor to
multi-processor applications. Again, it appears that the communication costs in the current system are too high to run this program efficiently on the system.

### B.3 Blocked LU

Table B.1, lists the communication latencies for a number of different region sizes. Looking at different region sizes, we find the following average communication latencies. (These results come from the average values for a large number of transfers, taken by calling the start_read operation from a node with an invalid copy.)

<table>
<thead>
<tr>
<th>Region Size (in bytes)</th>
<th>bandwidth</th>
<th>latency (in microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>.031</td>
<td>2064</td>
</tr>
<tr>
<td>256</td>
<td>.106</td>
<td>2415</td>
</tr>
<tr>
<td>1024</td>
<td>.256</td>
<td>4000</td>
</tr>
<tr>
<td>4096</td>
<td>.537</td>
<td>7627</td>
</tr>
<tr>
<td>16384</td>
<td>.684</td>
<td>23953</td>
</tr>
<tr>
<td>65336</td>
<td>.769</td>
<td>85222</td>
</tr>
<tr>
<td>262144</td>
<td>.833</td>
<td>314698</td>
</tr>
</tbody>
</table>

Table B.1: Experimental bandwidth and latency results for different region sizes, using two nodes.
Using a 166 MHZ Pentium at full utilization, and using the estimate of 11,000 cycles of useful work between communication calls, this implies that for every $\frac{11000 \text{ cycles}}{1 \text{ second}} = 66.2$ microseconds of real work, a communication needs to be done. Notice that each communication takes a minimum of about 2000 microseconds! Clearly, the applications that we're using for testing will have a very difficult time making efficient use of processor power with such slow communication primitives.

Over Ethernet, there is a difference of 341 microseconds between the time for a 64 byte and 256 byte region. The time to transmit the extra information is as follows:

$$\frac{1536 \text{ bits}}{650000 \text{ bps}} = \text{a latency of 236 microseconds}$$

(The throughput of 6.5 M-bps was derived from a simple experiment that timed the transmission of a 1MB message.)

The transmission difference accounts for part of the problem, but not for all of it. The next step shows a similarly large gap between the expected and actual times. Considering the gaps for larger message sizes, the throughput comes very close to the theoretical maximum throughput for Ethernet: the difference is expected to be 241979 microseconds. In actuality, the gap is 229476 microseconds.

In addition to the problems inherent in the slow network, it appears that there is a large amount of overhead in the networking software. The overhead of the TCP and active message layers of the initial implementation make it difficult to achieve low latencies with the current system. These initial poor measurements led us to try to improve the performance of CRL by the means described in this paper.
Bibliography


