EXTENSIBLE MESSAGE LAYERS FOR RESOURCE-RICH CLUSTER COMPUTERS

A Dissertation

Presented to

The Academic Faculty

By

Craig D. Ulmer

In Partial Fulfillment
Of the Requirements for the Degree

Doctor of Philosophy in Electrical and Computer Engineering

Georgia Institute of Technology

November 2002

Extensible Message Layers for Resource-Rich Cluster Computers

Approved:

Dr. Sudhakar Yalamanchili, Advisor

Dr. Kenneth Mackenzie

Dr. David Schimmel

Date Approved: 11/19/2002

For my parents, Steve and Judy Ulmer,

and especially my patient and loving wife, Amy.

FOREWORD

Approximately five years ago, Dr. Yalamanchili asked me if I would be interested in working with NASA to develop the computational systems for the next generation of spaceborne vehicles. The Remote Exploration and Engineering (REE) project at NASA had just finished a report concluding that future unmanned exploration vehicles would have to operate autonomously in order to be successful. These systems would have to capture large amounts of scientific data, process it locally, and transmit the most significant information back to Earth. In order to reduce development costs, NASA was interested in employing cluster computers in the spaceborne vehicles to perform these computational evaluations. My contribution to this effort would be to construct communication software that allows data to flow in a reliable and efficient manner between the hardware components of the system. Being that cluster computers are normally tucked away in the unwanted closets of a research building, it was appealing to think that we would be setting a few free, to be clusters in the sky. Aware of how this sounded similar to a Star Trek movie, I jumped at the opportunity.

After three years of work, we had constructed a functional communication library that achieved the goals of the research project. In addition to implementing the functionality commonly found in other communication libraries, our software allowed an intelligent server adaptor card to interact directly with the network interface. Plus, it had a cool name: GRIM (to which my advisor is still rolls his eyes). After the initial release of GRIM, we began investigating how the software could be improved to support other peripheral devices. As this work evolved, we realized that we were really providing a new form of cluster architecture. We refer to these clusters as resource-rich cluster computers, which are the focus of this dissertation.

As fate would have it, a number of researchers in industry were working on similar problems for commercial network servers. Their effort resulted in the InfiniBand I/O standard. Being that multimillion-dollar companies backed InfiniBand, we were initially concerned that we would be swept away by this monumental effort. However, we continued on with our work, following in the do-it-yourself style that has been the basis of the cluster computing movement. Our effort was rewarded earlier this year, when an InfiniBand evangelist stated in a keynote speech that the true threat to InfiniBand was from grassroots efforts taking place in commodity networks such as Gigabit Ethernet. Being that GRIM is designed to be applicable to any network substrate with intelligent network interface cards (including Gigabit Ethernet), we recognize this as a small victory and an indication that our effort has been worthwhile.

There are many organizations and people that have had a significant and positive impact on this research. This work was financially supported through a fellowship from NASA's Jet Propulsion Laboratory as well as through grants from the National Science Foundation. The cluster computer hardware utilized in this research was funded through large donations by the Intel Corporation. This work certainly would not have been feasible without these contributions, and we gratefully acknowledge the financial support of these organizations.

There are a number of professors that have had a significant influence on this work. First and foremost, this work would not have been possible without the assistance of my advisor, Dr. Sudhakar Yalamanchili. Sudha has been a constant source of encouragement and guidance over the years. Our weekly meetings always motivated me to push a little harder, and to construct new functionality that was beyond our original goals. Sudha helped transform a large amount of my text into actual (and concise) English. He did this in a kind way, often suggesting that he only had to make "a few minor changes" to something that I knew was poorly written. I am grateful for all the help and counseling Sudha has given me over the years. I will truly miss our whiteboard-centered meetings, where everything needed a block diagram no matter how irrelevant it was.

Dr. Kenneth Mackenzie also provided a great deal of assistance in this work. Ken's keen eye for detail motivated me to take a closer look into the low-level performance characteristics of the hardware. Ken continually surprised me by answering my questions with real performance measurements from programs he cooked up on the spot. Without his obsession for optimality, GRIM would never have reached its current level of performance. Ken was kind enough to allow me to use his cluster, even when he knew that I have a tendency to disorganize or break things. Most importantly, Ken instantly saw the soul of my work and enlisted me in the crusade to turn modern, boring computer architecture on its head. I can only hope that Ken will continue the good fight, and not get tied down by the bureaucracy.

Other professors had a significant impact on this work. Dr. Leon Alkalai provided encouragement and summer internships at NASA JPL. Leon supplied me with valuable view of the internals of JPL's work, and helped open my eyes to solving larger problems than what my degree had prepared me for. Dr. José Duato also assisted me in this work over the years. While only briefly mentioned in this dissertation, the discussions of deadlock freedom in irregular network topologies that took place with José, Sudha, and myself are part of the work that I enjoyed the most in graduate school.

I am fortunate to have been surrounded by many high-quality researchers and co-workers as a graduate student at Georgia Tech. Early on, Darrell Stogner, Santiago Abraham, and Phivu Nguyen provided me with a drive to investigate new technical material. Emily Crawford performed the initial backbreaking work with the Myrinet hardware that served as a starting point for my work. Ivan Ganev meticulously answered my kernel questions, no matter how silly they were. His shaved head also took Ken's attention away from my infrequent, self-induced buzz cuts. William Norton, Damon Love, and John Lockhart supplied me with a constant stream of desirable distractions, and served as a reality check for my work. I frequently harassed these people with painful implementation problems ("I just shifted all of a host's physical memory by 1024 bytes"). Thanks for frequently coming out to the Original Pancake House to listen and

provide useful suggestions ("Try not to do that"). Finally, Amer Abufadel has been a constant source of knowledge and help. In addition to answering my DSP questions, Amer and I went through many of the graduate student anxieties at the same time.

This work would not have been possible without the support of my family. My brother, Dr. Todd Ulmer, constantly pressed me to push on and finish. He also wrote a wonderful "forward" in his own dissertation that has made it impossible for me to write a halfway decent one myself. In any case, when I was young, Todd teased me by throwing my favorite toy, a little red plastic hammer, out the widow of our dad's moving car. While I will hold that over his head forever, he should know he is a good older brother and I thank him for all the help he has given me in school. I would also like to thank my parents for their love and support over the years. Even though they never quite understood what exactly it was I was working on, they always encouraged me to try harder. Hopefully, this dissertation unravels the mystery of my work a little bit.

Finally, I must thank my wife, Amy Pomerance, for her love and support. Amy patiently waited for me to finish, never letting on that she did not believe me when I kept telling her that it would "only be about a year now". While all of this work seemed to take forever, being with you during this time made it all fine. Thanks for everything.

CRAIG ULMER

NOVEMBER 2002

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

3GIO:	Third generation I/O
AGP:	Accelerated graphics port
AM:	Active messages
API:	Application programming interface
ASAN:	Active system area network
ASIC:	Application-specific integrated circuit
BIP:	Built-in Parallelism message layer
BT8x8:	Brooktree video capture card chipset
COTS:	Commercial off-the-shelf
CPLD:	Complex Programmable Logic Device
CPU:	Central processing unit
DMA:	Direct memory access
DSM:	Distributed shared memory
DSP:	Discrete signal processor/processing
EISA:	Enhanced industry standard architecture
FC:	Flow control
FFT:	Fast Fourier transform
FM:	Fast messages
FPGA:	Field-programmable gate array
GM:	Glenn's Messages
GNU:	GNU's not Unix
GRIM:	General-purpose Reliable In-order Messages
IB:	InfiniBand
IP:	Internet Protocol
ISA:	Industry standard architecture
LC:	Logical channel
LFC:	Link-level flow control
MCA:	Microchannel adaptor
MFLOPS:	Millions of floating-point operations per second
MP:	Multiprocessor
MPI:	Message Passing Interface
MPP:	Massively parallel processor
MTU:	Maximum transfer unit
NI:	Network interface
NIC:	Network interface card
PC:	Personal computer
PCI:	Peripheral component interconnect
RAID:	Redundant array of independent disks
RPC:	Remote procedure call
SAN:	System area network (also storage area network)
SMP:	Symmetric multiprocessor
SRAM:	Synchronous RAM
TCP:	Transmission control protocol
TPIL:	Tunable PCI injection library

UDP:	User datagram protocol
VM:	Virtual memory
VMMC:	Virtual memory mapped communication
VNN:	Virtual node number

SUMMARY

Cluster computing is an alternative approach to supercomputing where a large number of commodity workstations are utilized as the processing elements in a multiprocessor system. These workstations are interconnected by high-performance system area network hardware and specially designed "message layer" communication software. In the current generation of cluster computers, researchers have optimized message layers for communication between the host CPUs in the cluster in order to provide scalable computing performance. However, the recent development of a number of high-performance peripheral devices challenges the notion that message layers should be designed in such a CPU-centric manner. Modern peripheral devices feature powerful embedded processing and storage capabilities that can be leveraged to boost the performance of distributed applications. These peripherals function as sources and sinks of application data, and in some cases, as computational accelerators for offloading host-CPU tasks.

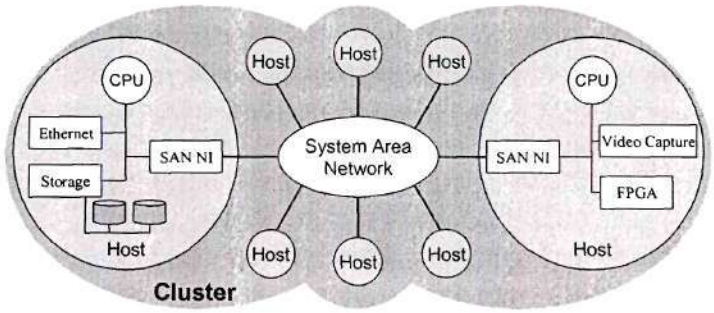


Figure 1: The architecture of emerging resource-rich cluster computers.

As Moore's Law continues its relentless trend, there will continue to be a migration of computing power to peripheral devices. Future clusters will not appear anything like the clusters of today. They will be rich in connectivity and computing power that is deeply embedded in the distributed components of the cluster. We refer to this new generation of systems as *resource-rich cluster computers* (Figure 1). These systems differ from traditional clusters in that application

processing takes place in both the host CPUs *and* the peripheral devices. While the semiconductor industry continues to alter the economies of scale, the system software that productively enables resource-rich clusters is sorely lagging. Specifically, current generation message layers are ill equipped to service the needs of resource-rich clusters, as they are not designed to utilize peripheral devices as globally accessible resources in a cluster.

This thesis focuses on the challenge of designing extensible message layers for this new generation of resource-rich clusters. We are specifically concerned with making peripheral devices available as globally accessible resources in the context of a programming model that permits applications to effectively and efficiently exploit the capabilities afforded by resource-rich clusters. The key contributions of this thesis fall into two categories. The first includes design concepts and programming abstractions for structuring messages layers to integrate powerful peripheral devices into a globally accessible pool of resources. The second class of contributions is engineering solutions to the challenging problems of effectively and efficiently realizing these design concepts in a manner that tracks the evolution of technology, that is, the continued migration of computing power to distributed resources.

CHAPTER I

INTRODUCTION

1.1 Background

After years of escalating supercomputer costs, researchers in the early 1990's began investing alternative means by which parallel-processing systems for scientific and military applications could be constructed in a more economical fashion. This work resulted in the field of cluster computing, where a large number of commodity workstations are utilized as the processing elements in a multiprocessor system. These workstations are interconnected by a high-performance communication network and function as part of a single, parallel-processing machine. While cluster computers typically lack the peak performance levels of traditional supercomputers, they provide excellent cost-to-performance ratios that have attracted the attention of many users.

The enabling technology for cluster computers is communication software that is referred to as the cluster's message layer. This software provides a set of message-passing programming abstractions that are utilized to transport data between communication endpoints in the cluster. Early research in cluster computers revealed that end application performance is often sensitive to the latency and bandwidth characteristics of a message layer's implementation. Therefore, a significant amount of research in the late 1990's focused on improving the host-to-host communication performance of a cluster's message layer. This effort has resulted in message layers that are highly optimized for efficiently transferring data between a cluster's host CPUs.

1.2 Resource-Rich Cluster Computers

While current generation message layers have been suitable for a number of parallel applications, the recent development of a number of high-performance peripheral devices challenges the notion that message layers should be designed in such a *CPU-centric* manner. Modern peripheral devices feature powerful embedded processing and storage capabilities that can be leverage to boost the performance of distributed applications. These peripherals function as sources and sinks of application data, and in some cases, as computational accelerators for offloading host-CPU tasks.

As Moore's Law continues its relentless trend, there will continue to be a migration of computing power to peripheral devices. Future clusters will not appear anything like the clusters of today. They will be rich in connectivity and computing power that is deeply embedded in the distributed components of the cluster. We refer to this new generation of systems as *resource-rich cluster computers*. These systems differ from traditional clusters in that application processing takes place in both the host CPUs and the peripheral devices. While the semiconductor industry continues to alter the economies of scale, the system software that productively enables resource-rich clusters is sorely lagging. Specifically, current generation message layers are ill equipped to service the needs of resource-rich clusters, as they are not designed to utilize peripheral devices as globally accessible resources in a cluster.

This thesis focuses on the challenge of designing extensible message layers for this new generation of resource-rich clusters. We are specifically concerned with making peripheral devices available as globally accessible resources in the context of a programming model that permits applications to effectively and efficiently exploit the capabilities afforded by resource-rich clusters.

1.3 Contributions

The key contributions of this thesis fall into two categories. The first includes design concepts and programming abstractions for structuring message layers to integrate powerful peripheral devices into a globally accessible pool of resources. The second class of contributions is engineering solutions to the challenging problems of effectively realizing these design concepts in a manner that tracks the evolution of technology, that is, the continued migration of computing power to distributed resources.

The specific contributions of the thesis are as follows.

- We define a general framework for high-performance, extensible message layers in resource-rich cluster computers. This framework is structured around the following high-level components.
    - A device *independent* communication core that executes within the network interface and employs i) reliable message delivery mechanisms, ii) a virtual network interface abstraction, iii) an active message programming interface, and iv) a memory transfer programming interface. This communication core is implemented in a message layer for commodity clusters called the General-purpose Reliable In-order Message layer (GRIM).
    - Device *dependent* functionality is captured in the form of active message function handlers that are implemented within target peripheral devices. Services are provided for the global registration and management of device handlers. Thus, new devices can be integrated and made available for use throughout the cluster in a relatively seamless manner.

- The ability of the core message layer functionality to effectively and efficiently serve as the host for high-level communication functionality is demonstrated via the implementation of the following.
    - Multicast: A system for replicating multicast messages in the network interface is implemented as an extension to the core communication framework of the message layer. These extensions improve application performance by significantly reducing the workload of communication endpoints during multicast operations.
    - Sockets API Emulation: An implementation of a sockets API for the message layer allows legacy applications to leverage system area network hardware for significant improvements in communication performance.
    - Streaming Computations Framework: An API and services for pooling peripheral FPGA accelerators across multiple hosts into a unified computing resource allows operations to be performed on high-volume data streams. A unique aspect of this implementation is that device-specific handlers in this case are constructed in hardware, underscoring the generality and flexibility of the GRIM-based services and API.
- The ability of the core message layer functionality to be effectively, efficiently, and easily integrated with a variety of peripheral devices is illustrated via the integration of four peripheral devices listed below in increasing degrees of functionality and power.
    - AGP Video Display Card: Extensions to the communication library allow distributed endpoints to graphically update remote video display cards in the cluster.
    - Brooktree Video Capture Card: The BT8x8 video capture card allows video data streams to be generated and distributed to endpoints in the cluster.

- I_2O Server Adaptor: This server adaptor card provides intelligent management of both network and storage resources.
- Celoxica RC-1000 FPGA Card: This field-programmable gate array (FPGA) card functions as a computational accelerator through the emulation of application-specific circuits in reconfigurable hardware.

This thesis has focused considerable effort on the engineering challenges of harnessing emerging and powerful peripherals and proposing solutions that are comparable with existing message layers in terms of performance. This thesis demonstrates that message layer design, driven by a system-level view and supported by engineering trade-offs that carefully distribute functionality, can provide effective solutions to harnessing the potential of resource-rich clusters.

1.4 Organization of the Thesis

The work presented in this thesis is organized as follows.

- **Chapter 2**: A brief background of cluster computers is provided to summarize how clusters have emerged and evolved over the last decade. A fundamental description of traditional cluster hardware is presented, as well as brief descriptions of existing message layers for cluster computers.
- **Chapter 3**: This chapter provides information about the environmental characteristics of resource-rich clusters. Based on these characteristics, fundamental message layer properties for these clusters are discussed.
- **Chapter 4**: The guidelines for designing a resource-rich cluster message layer are then applied to implement a real system. This chapter discusses the core functionality of the GRIM communication library.
- **Chapter 5**: The performance characteristics of GRIM for traditional transactions between host CPUs is examined and compared with existing work.

- **Chapter 6**: This chapter provides a description of how commercial peripheral devices can be attached to the GRIM communication library. In order to illustrate the extensible nature of GRIM for supporting hardware, four commercial peripheral devices with different operating characteristics are integrated into the GRIM library. Implementation and performance details are provided for each card.
- **Chapter 7**: Integrating distributed, specialized computing resources into a unified infrastructure for an application is the topic of this chapter. Specifically, this chapter provides insight as to how peripheral devices can be utilized to construct distributed, computational pipelines.
- **Chapter 8**: To demonstrate the extensibility of GRIM for application software, this chapter provides implementation details of a multicast system that performs message replication in the NI, general-purpose fragmentation and reassembly mechanisms, and an emulation of a sockets API.
- **Chapter 9**: The thesis concludes with some summary remarks and directions for future work.

CHAPTER II

BACKGROUND

By the end of the 1980's, the need for high-performance computing platforms in scientific and military applications had resulted in the emergence of a small number of supercomputer companies. These companies constructed large-scale systems that utilized massive amounts of custom hardware to improve application performance. Unfortunately, because these systems were extremely expensive, supercomputers were not a practical option for a large number of end users. Therefore, researchers in the 1990's began exploring alternative high-performance computational platforms that could be constructed in a more cost-effective manner. One of the results of this effort is the field of *cluster computing*. In cluster computing a large number of commercial workstations are collectively utilized to function as a single, multiprocessor system. Since system hardware is comprised of widely available commercial components, cluster computers can be constructed at a fraction of the cost of traditional supercomputers. As such, a considerable amount of high-performance computing research in recent years has focused on improving cluster computer performance.

A key challenge in improving cluster computer performance is adapting commodity hardware and software to function as part of a high-performance, multiprocessor system. Early cluster computing efforts revealed that application performance is highly dependent on the performance of a cluster's communication facilities. From a hardware perspective, several companies have addressed this issue by constructing system area networks (SANs) that provide an order of magnitude improvement over traditional local area networks (LANs). From a software perspective, researchers have constructed specialized communication libraries, or message layers, that are designed to deliver native SAN performance to end applications. In addition to

facilitating low-latency, high-bandwidth communication, these message layers provide a programming abstraction where the cluster is viewed as a pool of host CPUs in a large *virtual parallel-processing machine*. This abstraction has sufficed for numerous researchers to effectively utilize a cluster computer's hardware as a distributed multiprocessor system.

2.1 Evolution of High-Performance Computing Platforms

Supercomputers are the computational systems that deliver the highest peak performance of all computer systems available at a given point in time. These systems typically employ large amounts of custom hardware to accelerate computational performance and often feature specialized computer architectures. Supercomputers have been primarily designed to process complex scientific applications that frequently exhibit large amounts of data parallelism. A number of commercial supercomputer systems have been produced since early groundbreaking work performed by the industry in the 1970's. The evolution of this technology provides both insight into high-performance computing and a motivation to continue the work in related research areas.

2.1.1 A Brief History of Commercial Supercomputers

While numerous people have contributed to the field of supercomputing over the years, perhaps the most influential individual in this effort is pioneer Seymour Cray. After leaving the Control Data Corporation in 1972 to form Cray Research, Cray began work on a new computer architecture that would provide significant gains in peak performance levels. In addition to advances in high-speed circuitry, Cray investigated the use of sophisticated vector processing units that allow computations to be applied to a stream of data to achieve high throughput. In 1976 the Cray-1 [1] was brought to market with a retail value of approximately nine million dollars and a record-breaking performance of 133 million floating-point operations per second (megaflops). In addition to being a technological marvel, the Cray-1 demonstrated that there was

a definite market for expensive high-performance computing systems. Cray continued his work with vector processor systems, producing the 2 gigaflops Cray-2 in 1985 and the 5 gigaflops Cray-3 in 1989. A number of other computers followed the trend of vector processor systems, including the Meiko CS-2 [2], the NEC SX series supercomputer [3], the Fujitsu VP series supercomputer [4], and IBM's vector extensions to the System/370 [5]. Currently, the fastest system in the world [6] is the NEC SX-6, used in the Earth Simulator Center [7] in Japan. This system provides up to 8 teraflops of performance and employs multiple single-chip vector processing units.

The supercomputing industry also explored other architectural techniques for increasing the computational performance of a system. A key effort in this work is the use of a large number of processors to perform computations in parallel. In the SIMD (single instruction stream, multiple data streams) approach, a large number of identical processors perform the same series of operations on different data sets. Multiple SIMD systems were constructed in the early 1990's, including the MasPar Computer Corporation's MP-1 [8] and the Thinking Machines Corporation's CM-2 [9]. Both of these systems housed up to 16,384 SIMD processing elements, and could be used for parallel applications such as image processing. Due to the programming complexity of SIMD, researchers began constructing MIMD (multiple instruction streams, multiple data streams) systems that employed a large number of general-purpose CPUs. This work resulted in massively parallel processing (MPP) systems such as the Intel Paragon [10] (up to 4,000 Intel 80860 processors), the TMC CM-5 [11] (up to 16,000 SUN SPARC processors), and the Cray Research Cray-T3E [12] (up to 2,048 DEC Alpha 21164 processors).

2.1.2 Motivation for Alternate Computing Platforms

While the supercomputer companies of the 1980's provided significant advances in the field of high-performance computing, a large number of these companies withdrew from the supercomputer business in the 1990's. In hindsight it can be said that a common vulnerability for

these companies was the large amount of custom design that was required to build a supercomputer. Several of these companies operated with a vertical design methodology, constructing all components of the system from the individual processors to the interconnection network. While having complete control over the design space gave engineers freedom to innovate performance enhancements, product design times were increased and complicated by the volume of custom hardware design that was required. Therefore, new supercomputers were expensive, brought to market infrequently, and often could not be designed in time to utilize the latest developments in state-of-the-art technology.

Additional issues make traditional supercomputers less appealing to researchers that need high-performance computing platforms. First, supercomputers generally are not scalable and therefore offer a limited lifetime of leading-edge use. An investment in a state-of-the-art supercomputer depreciates rapidly in value due to Moore's Law, thereby making current systems obsolete within 18 months. Second, supercomputers require specialized hardware and software maintenance that adds to the expense of ownership. These components can be expensive to replace and there are generally few people that are trained to perform such maintenance. Finally, it must be noted that a risk in purchasing a traditional supercomputer is that the manufacturer might go out of business or otherwise abandon support for a particular product. Maintaining and utilizing orphaned hardware is time consuming and ultimately impedes end users.

Given the problems associated with using traditional supercomputers, a number of researchers in the early 1990's began exploring alternative methods by which high-performance computational platforms could be constructed. This effort made several observations about commercial technological advances and the global marketplace that would influence the construction of future parallel-processing systems. These observations include the following:

- **Commercial Off-the-Shelf (COTS) Parts**: In industry there are numerous corporations producing state-of-the-art hardware and software components. By using COTS parts,

designers leverage other people's work and reduce the design time for a system. COTS parts are also beneficial because components can easily be replaced or upgraded from third-party products.

- **Growth in the Workstation and Network Markets**: Consumer demand for personal computers has resulted in high-performance workstations that are available at a low cost. Processor design in this market remains competitive, resulting in frequent updates to peak performance levels. Likewise, consumer interest in the Internet has resulted in advances in network hardware. The need for faster networks has resulted in low-cost local area networks (LANs) that economically offer high-bandwidth communication.
- **A Rich Software Environment**: An important aspect of commodity workstations is the wide availability of software. Operating systems such as GNU/Linux provide a UNIX-like environment with built-in network features. The open source nature of Linux allows researchers to easily incorporate custom functionality into the operating system kernel.

In summary, researchers observed that advances made in consumer markets in the 1980's and 1990's had resulted in hardware that was widely available, economical, and offered respectable levels of computational performance. These systems could be utilized to provide impressive price-to-performance ratios and have benefited from considerable efforts to improve the PC's software environment.

2.1.3 Emergence of Cluster Computers

In the mid-1990's, researchers began investigating the use of multiple commodity workstations to construct a new form of high-performance system. This work resulted in the notion of a *cluster computer,* where a number of workstations are collectively utilized to function as a single parallel-processing system. Through commodity network hardware and specialized communication software, a cluster computer can effectively appear as a large pool of host

processors to the end user. Since workstations in the cluster are commercially available products, cluster computing can leverage the performance gains achieved by the workstation industry. The high-level architecture for a cluster computer is depicted in Figure 2.1.

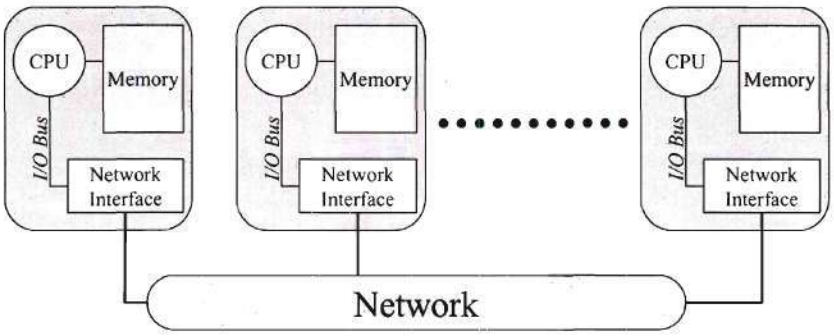


Figure 2.1: A cluster computer constructed with commodity workstations and network hardware.

One of the first cluster computing projects to receive serious attention from the scientific community was NASA's Beowulf Project [13]. In this work, researchers demonstrated that a small number of dedicated workstations could collectively operate to perform computations that were beneficial to scientific computing [14]. Utilizing commodity PCs equipped with multiple Ethernet adaptors, the 16-node demonstration cluster achieved 60 megaflops in 1994. Later clusters in this project would expand the number of workstations to 199 nodes and accomplish 10 gigaflops of performance for under $50,000. While researchers stated that Beowulf clusters were a far step from true supercomputing, the price-to-performance ratio was a significant motivator for building such clusters. After this work numerous research institutes constructed Beowulf-style clusters out of Ethernet-connected PCs.

An observation made about the early Beowulf-style of cluster was that while some applications performed well, others did not. An examination of this problem revealed that these clusters were severely limited in terms of communication performance. Grossly parallel applications that did not require significant amounts of communication between host CPUs

performed well because each task in the cluster could operate independently. However, applications that required frequent exchanges of data between CPUs performed poorly due to the low performance of the network. The conclusion to be drawn from this observation is that ultimately, the communication performance of the cluster determines the granularity at which parallel-processing applications can productively use a cluster.

Realizing that the poor communication performance limited the types of applications a Beowulf cluster could run, researchers in the mid to late 1990's began examining ways in which the cluster's communication performance could be enhanced. Several academic projects focused on adding hardware to facilitate specific types of communication. In the SHRIMP project at Princeton [15], workstations were extended with hardware that allowed hosts to operate in a distributed shared memory (DSM) environment. At Purdue, the PAPERS project [16] utilized custom hardware to rapidly distribute barrier synchronization information to host computers. However, the most significant advance for cluster computers came with the advent of commercially available system area networks (SANs). SANs provide communication performance that is over an order of magnitude better than traditional LANs. This allows for significant improvements in fine-grain parallel processing performance. Current work in high-performance cluster computers involves delivering as much native performance from a SAN as possible to end applications.

2.2 Using Workstations as a Cluster Computer's Processing Elements

Multiprocessor systems are generally comprised of two types of hardware components: processing elements that are used to perform computations, and a communication network to distribute data in the system. In cluster computers, individual workstations function as processing elements, while commodity SAN hardware performs communication tasks.

2.2.1 Workstation CPUs

A number of vendors have constructed different workstations that can be utilized in a cluster computer. Historically, companies such as Sun Microsystems, Hewlett-Packard, SGI, and Compaq/DEC have dominated the workstation industry with CPU architectures that offer high performance at a relatively high cost. However, the workstation market for these companies has eroded over the last decade as x86-based PCs and PowerPC-based Apple Macintosh computers have steadily improved in performance and popularity. Because of its impressive price-to-performance ratio, the x86 PC has become the workstation of choice for the majority of cluster computing efforts. Therefore, the work presented in this thesis specifically deals with clusters constructed from x86-based PCs.

While affordable, x86-based systems have some of the most limiting architectural characteristics of any workstation when used for high-performance computing. First, the x86 is based on a 32-bit architecture that may not be sufficient for the processing needs of scientific applications that require 64-bit computations. Second, an x86 processor can only support 4 GB of physical memory. This trait limits the amount of state information an application can have loaded at a workstation, and is becoming more of an issue as memory prices continue to decline. Finally, in order to obtain peak performance levels in x86-based hosts, it is often necessary to utilize architectural extensions such as the MMX and streaming SIMD (SSE) units. The performance of these units can vary greatly between different generations of x86 processor.

2.2.2 Evolution of Workstation I/O Systems

A second key factor that affects the performance of a workstation as a processing element is the architecture of its I/O system. In ideal multiprocessor systems, processing elements are placed in close proximity to the NIs in order to allow fine-grained interactions between applications and the network. Unfortunately, in most workstations the CPU and NI are separated by a complex general-purpose I/O system. Transactions involving the I/O system can be up to an

order of magnitude slower than similar transactions with host memory. While the computer industry makes improvements to x86 CPU performance multiple times a year, PC I/O performance is improved on average once every *three years*.

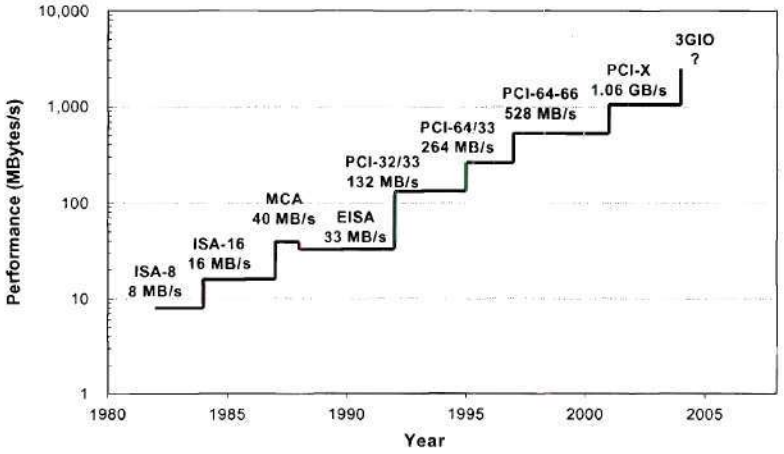


Figure 2.2: A history of PC I/O systems.

Figure 2.2 highlights the history of I/O systems utilized in PCs. The peripheral component interconnect (PCI) standard [17] provides reasonable performance and has become the de facto standard for peripheral devices in current workstations. While this thesis targets PCI-based systems, the implementation can be tuned to platforms with higher bandwidth I/O systems.

2.2.3 Peripheral Component Interconnect (PCI)

The peripheral component interconnect (PCI) standard was introduced in 1992 as a means of allowing high-speed peripheral devices to be incorporated into the x86 PC architecture. The architecture of modern host systems employing PCI is depicted in Figure 2.3. In this architecture the system's memory controller is responsible for routing data between the host CPU(s), main memory, and peripheral devices on the PCI bus. At boot time the memory controller assigns regions of the host's 32-bit physical address space to both main memory and individual peripheral devices. When a device driver for a PCI card is loaded into the kernel, the driver can establish a memory translation that allows the card's memory to appear in the kernel's

virtual address space. The driver can then share this mapping with user-space applications through the implementation of a memory map system call, handled by the device driver. Doing so allows user-space applications to directly read and write the on-card memory of a peripheral device.

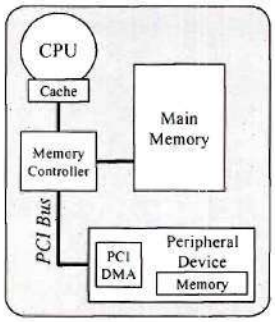


Figure 2.3: Architecture of a modern host utilizing PCI.

In addition to memory mapped reads and writes from the host CPU, communication involving peripheral devices can be facilitated by on-card DMA engines that are available with *bus-mastering* PCI devices. These DMA engines adhere to the low-level PCI bus standard and can be used to transfer blocks of data between a peripheral device and host memory or other peripheral devices in the system. All memory references on the PCI bus are in terms of the host's 32-bit physical address space in the x86 architecture. Each PCI device also controls an interrupt request (IRQ) line, which can be used to transmit an interrupt to the host CPU. Due to a limited number of IRQs in a host, multiple peripheral devices may share the same interrupt, requiring each card's device driver to determine which card initiated an interrupt.

A number of modern PCI devices support sophisticated DMA transfers through the use of chained DMA operations. With chained DMA, a peripheral device is capable of performing a series of DMA operations as specified by a linked-list of DMA descriptors. Each descriptor in a linked list specifies the length, direction, and addresses of a transfer. Some implementations allow users to specify whether an interrupt should be generated for the host at the completion of a transfer for a given DMA descriptor. The DMA engine processes each descriptor linearly until an

end-of-chain marker is specified in a descriptor. While most cards employ a similar API for controlling chained DMA operations, it should be noted that there is no standard and that each peripheral device driver must be outfitted with custom functionality.

2.2.4 Architecture Tradeoffs

There are a number of architectural tradeoffs designers must face when considering how cluster computer workstations can be used as processing elements in a multiprocessor system. From the previous three subsections it is clear that host CPUs in a cluster computer incur significant overheads when interacting with other CPUs in the cluster. This trait is a serious obstacle for application designers, especially when clusters are compared to traditional MPP supercomputers that allow fine-grained network interactions. However, a workstation by itself is a complete, self-contained system that features processing, memory, and storage resources, as well as a sophisticated operating system for managing these resources. Therefore, a processing element in a cluster computer is more likely to be better equipped to perform diverse tasks than a processing element in a traditional MPP supercomputer. The architectural tradeoffs of using workstations as processing elements therefore suggests that cluster computers are better utilized for computations where operations can localized to individual processing elements.

2.3 Cluster Computer Network Hardware

In addition to processing elements, multiprocessor systems must be equipped with communication infrastructure that allows distributed processor elements to interact. In cluster computers, this infrastructure is built from commercial network hardware. Several network substrates have been used in cluster computers over the years. One of the most popular approaches is to employ traditional LAN hardware such as Ethernet. While economical, the drawback of Ethernet is that it only offers limited host-to-host communication performance in a cluster environment. Therefore, a number of companies have constructed system area network

(SAN) products that are better suited for cluster computers. These SANs feature multi-gigabit bandwidths and host-to-host transmission latencies that are less than 50 μs. SANs generally offer high levels of reliability and commonly utilize intelligent NI cards to manage network interactions. Examples of SANs include Myricom's Myrinet [18], Compaq's Servernet [19], Dolphin Interconnect Solutions' implementation of the scalable coherent interconnect (SCI) [20], and Quadrics' QsNet [21].

2.3.1 Ethernet

The Ethernet network standard first created at Xerox Parc labs in 1976 [22] has grown to become the most popular network ever utilized. The Ethernet standard has been periodically updated over the years, and now features link speeds of up to 10 Gbps in the most recent standard [23]. Ethernet NI cards traditionally have employed a simple hardware architecture, where the NI only manages a pair of message queues for incoming and outgoing transmissions. In this approach, the host CPU formats and processes all messages transferred to and from the network. In order to reduce the workload of the host CPU, some high-end Ethernet NI cards feature more sophisticated processing engines that are capable of managing network interactions on behalf of the host. These intelligent NI cards are especially beneficial for Gigabit Ethernet networks where high-bandwidth transactions are necessary.

While widely available, Ethernet is not the ideal communication substrate for cluster computers. The primary issue is that Ethernet was designed for use in LANs. Since data in LANs is transmitted over long distances, Ethernet is largely optimized for bandwidth but not latency. Another consequence of Ethernet being designed for LANs is that the hardware is not designed to facilitate reliable transmissions. Instead, workstations in the cluster must implement reliable transmission protocols that can tolerate dropped messages. Finally, Ethernet hardware can be criticized because currently there is a lack of high-performance NI adaptors. In [24], researchers compare several commercial Gigabit Ethernet adaptors. Tests using a host with 32b/33MHz PCI

found that the maximum obtained bandwidth was only 436 Mb/s, while most of the cards provided less than 200 Mb/s. In tests using a host with 64b/66MHz PCI, general performance rose to 650 Mb/s, with one card obtaining 928 Mb/s. While industry is steadily improving Gigabit Ethernet product performance, these tests demonstrate that it is still challenging to obtain the peak performance levels of the network.

2.3.2 Scalable Coherent Interconnect (SCI)

The scalable coherent interconnect (SCI) standard is a SAN for clusters that has gained widespread use in Europe. SCI evolved out of the Futurebus+ project [25] in 1988 as a means of developing a next-generation I/O infrastructure for high-performance workstations. SCI is designed to allow a large number of hosts to function as part of a distributed shared memory machine. In the programming model for this system, each host is allocated a region of memory in SCI's global address space. When a host reads or writes a region of the address space that is not available at the local node, the SCI NI card forwards the transaction to the memory system of the host that owns the memory. Distributed memory interactions take place efficiently in SCI because shared memory protocols are implemented in hardware in the SCI NI cards. Initial versions of SCI interconnected hosts in ring topologies that were similar to token ring LANs. As the standard evolved SCI hardware was adapted to operate in point-to-point network topologies using dedicated switches.

An advantage of SCI's approach to communication is that it provides a specific set of actions that the network hardware must perform. These actions can be implemented with custom hardware that benefits from circuit-level optimizations. While this prevents the NI from being extended with functionality by the user, it allows NI hardware to be simplified and produced more economically. One of the largest vendors of SCI hardware is Dolphin Interconnect [26]. This company's implementation of SCI has an application-to-application performance of up to 2.6 Gb/s in bandwidth and 1.4 μs in latency [27] (using IA64 Itanium hosts).

2.3.3 Myrinet

Myricom's Myrinet is one of the most commonly utilized SANs for cluster computers, due to its high levels of performance and programmability. Myrinet is a descendent of the Mosaic [28] and ATOMIC [29] research projects. In these projects, researchers developed a high-performance network for multiprocessor systems that employed source-routed, wormhole [30] messages to reduce switch latencies. These networks provided high levels of data reliability and would only drop messages if deadlock was suspected. Myricom converted Mosaic into a commercial product known as Myrinet for use with commodity workstations. Current generation Myrinet hardware is comprised of network switches, 1.28 to 2.0 Gb/s full-duplex links, and programmable NI cards. Network hardware is connected in a point-to-point fashion, allowing the construction of both regular and irregular network topologies. In minimizing switch latency, Myrinet designers have pushed network tasks out of switch hardware and into the NI cards. A beneficial side effect of this design choice is that network functionality (e.g., multicast or added fault tolerance) can be implemented by users in the form of NI firmware. Myricom has released several generations of NI hardware as summarized in Table 2.1.

Table 2.1: A history of Myrinet network interface cards.

NI Processor	Year	Link Speed	NI Speed	NI Memory	Host I/O
LANai 3	1994	640 Mb/s	25 MHz	128 KB	20 MHz SBUS
LANai 4	1996	1.28 Gb/s	33 MHz	1 MB	32b/33MHz PCI
LANai 9	2000	1.28 / 2.0 Gb/s	100-200 MHz	2-8 MB	64b/66MHz PCI

The organization of the Myrinet NI is depicted in Figure 2.4. In this architecture, the NI is situated between an interface to the host I/O system and an interface to the network wire. High-speed SRAM is utilized to house both the executable firmware and data for the NI. Firmware typically occupies less than 256KB of SRAM memory, allowing the remaining memory to be used as needed by communication library designers. The SRAM is shared between the LANai and DMA engines through a priority-based memory controller.

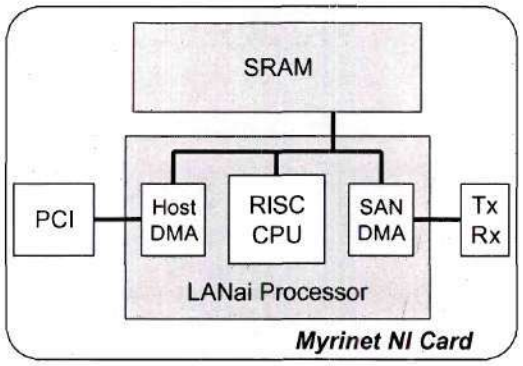


Figure 2.4: Architecture of the Myrinet NI card.

While the architecture of the Myrinet NI is relatively simple, the NI is a powerful device because it can support multiple data transfers at the same time. The NI can be configured to simultaneously send data to the network, receive data from the network, and issue a DMA transfer to or from host memory. In more recent versions of the Myrinet NI, the NI is also capable of supporting multiple DMA transfers between the NI and host using four PCI DMA engines. The programmable nature of the NI has allowed firmware designers to construct efficient communication pipelines with the NI, where data is transferred in a cut-through manner without buffering delays in the NI. Basic performance measurements of the LANai 4 and 9 NI cards are provided in Appendix A.

2.3.4 Quadrics QsNet

The QsNet [21] interconnection network is a relatively new SAN product created by Quadrics in Europe. QsNet is currently being utilized in high-end cluster computers such as the Terascale Computing System [31] at the Pittsburgh supercomputer center (currently the third-fastest supercomputer in the world [6]). Similar to Myrinet, QsNet uses wormhole routing to efficiently transfer data between NI cards through a point-to-point network. However, QsNet differs in that communication resembles a virtual circuit approach, as wormhole transmission paths are not released until the receiver transmits an acknowledgement token. Similar to SCI, QsNet provides a means of allowing hosts in the network to share a global address space. In order

to accelerate remote memory operations, NI cards are equipped with hardware engines that can dynamically translate virtual addresses into physical addresses. Initial reports of QsNet indicate that it is capable of providing over 2.4 Gb/s of bandwidth and approximately 2 μs of latency between user-space applications.

2.4 Cluster Computer Network Software

After early work in Beowulf-style clusters, researchers observed that the high latency of traditional communication libraries for LANs precluded fine-grained cluster applications. Given the raw performance available in SAN hardware, a considerable amount of cluster computing research in the late 1990's focused on techniques for harnessing this communication performance. This work resulted in the development of a number of custom "message layer" communication libraries that offered mechanisms for reducing communication latency and increasing bandwidth between host-level applications. The most commonly utilized SAN in this effort is Myrinet, due to its open source software and well-documented hardware. A large number of message layer packages have been implemented for Myrinet, including Active Messages (AM, AM II) [32,33], Fast Messages (FM) [34], PM [35], Link-level Flow Control (LFC) [36], Trapeze [37], Virtual Memory Mapped Communication (VMMC) [38], GM [39], and BIP [40].

2.4.1 Limitations of LAN Protocols

Early Beowulf-style cluster computers utilized traditional LAN hardware and software to provide reliable communication between workstations in the cluster. These clusters typically employed Ethernet network hardware and communication software based on the transmission control protocol (TCP). While leveraging existing LAN equipment allowed large clusters to be constructed easily in a cost-effective manner, researchers observed that these Beowulf-style clusters offered limited performance in some parallel processing applications. The fundamental issue observed with using LAN equipment is that it is primarily designed to transmit data over

long distances using an error-prone medium. Therefore, LAN software such as TCP must perform a number of complex transmission management operations in order to guarantee that messages are reliably delivered in the proper order to a destination.

Cluster computers have different operating conditions than LANs. In cluster computers, workstations are separated by small distances and utilize dedicated network switches for local communication. Under these conditions, messages are dropped or reordered by the network infrequently. Therefore, TCP's reliable transfer mechanisms are not optimal for cluster computers and are in general too heavy weight for high-performance applications. Transmissions using Ethernet and TCP can suffer communication latencies greater than 100 μs for host-to-host deliveries. By comparison, inter-processor communication in a symmetric multiprocessor (SMP) node takes place in only a few microseconds. This difference in communication performance is enough to significantly limit the effectiveness of cluster computers in the case of fine-grained, communication intensive parallel programs [41]. Therefore, researchers in the late 1990's began investigating custom message layers that were better suited for cluster computers.

2.4.2 Message Layer Characteristics

Message layers for cluster computers serve as a means of transferring application data between *communication endpoints* in the cluster. Naturally there are many ways in which message layers can be designed. Therefore, a first step in understanding how message layers function is to consider a few of their key characteristics. These characteristics include the following.

- **Programming Interface**: One of the most defining characteristics of a message layer is the programming interface that is provided to end users. Message layers generally employ one of three types of programming interface. First, active message [42] systems utilize an interface similar to remote procedure calls (RPCs) [43], where an application

can invoke an operation at a remote endpoint simply by transmitting a message. Second, in rendezvous approaches, sending and receiving endpoints are tightly synchronized and require the receiving endpoint to post requests to extract certain messages from the network. Finally, systems using a shared memory programming interface use remote memory operations to manipulate data located at different hosts in the cluster.

- **Buffer Management and Flow Control**: NI cards have a limited amount of buffer space for housing in-flight messages. Therefore, an important aspect of a message layer is the means by which it manages the reliable transfer of messages from one endpoint to another. In some message layers, flow-control schemes are applied at either the host or NI levels in order to prevent messages from being dropped due to insufficient buffer space at the receiver. Other message layers do not implement such mechanisms, either for performance reasons or because they are not necessary. For example, in a shared memory system, the receiving NI always accepts and processes an incoming message, so buffer management is not necessary.

- **Delivery Order**: In a strictly ordered system, messages are processed by a receiver in the same order they were injected into the network by the sender. When messages are dropped in the network, a message layer with ordered delivery performs retransmission and reordering to maintain consistent data flow. In systems where network messages can carry priorities, some message layers allow higher priority messages to bypass lower priority messages by relaxing ordered delivery constraints.

- **Receiver Notification**: Another characteristic of a communication library is the manner in which the receiving application is notified that a new message has arrived. In message layers that notify the receiving endpoint, either an interrupt mechanism or polling is

utilized. While interrupts allow applications to interact with the message layer only when new data has arrived, the interrupts can take place at any time and are therefore challenging to manage. Polling techniques require an application to periodically examine the message layer for new data, but can generally provide better performance than interrupts. Shared memory systems do not necessarily need to utilize any explicit form of notification, as this task is implicitly performed by the receiver application.

2.4.3 Common Message Layer Optimizations

Researchers often utilize a number of common techniques for improving the performance of a message layer. One of the earliest and most widely used techniques is to construct a message layer in user-space. This technique is beneficial because it allows an application to interact with the NI card without invoking expensive system calls. Another common technique used in many message layers is to make use of the reliable nature of SAN hardware. Since SAN hardware can operate for months at a time without a single bit error, researchers often simplify message layer protocols by assuming the common case of error-free transmissions. Finally, with the observation that the host CPU is much more powerful than the NI processor, a number of researchers have minimized the amount of work NI processors perform in the message layer. While these optimizations have boosted performance in message layers, such approaches have resulted in *CPU-centric* message layers. As will be discussed in the following chapter, these message layers are inappropriate for resource-rich cluster computers, which require both host CPUs and peripheral devices to function as communication endpoints.

2.4.4 Myrinet Message Layers

A number of message layers have been constructed for Myrinet over the years. An excellent survey of several of these message layers is provided in [44]. The more influential of these message layers are summarized as follows.

- **Active Messages (AM, AM-II) [33]**: The active messages project was one of the first academic message layer packages for Myrinet hardware. In addition to demonstrating the active message programming paradigm, AM software illustrated that communication libraries implemented in user space could provide significant performance improvements. AM utilizes host-based flow-control mechanisms to manage buffer space in the library.

- **Fast Messages (FM) [34]**: The FM library was released shortly after AM and extends AM concepts by providing mechanisms for increased performance, stability, and usability. FM utilizes an active message programming interface and includes mechanisms for registering and managing application function handlers. Another feature of FM is its ability to efficiently fragment and pipeline large message transmissions, which allowed for significant gains in communication performance. While FM originally employed NI-based flow-control mechanisms, these mechanisms were later deferred to the host due to poor NI performance.

- **BIP [40]**: The BIP message layer is an effort to construct a lean message layer that can provide high performance for higher-level programming interfaces such as MPI [45]. BIP uses a rendezvous communication model where a receiver must provide the NI with information that specifies where the NI should store a particular incoming message. BIP

provides no reliability guarantees, and has been reported to have the best communication performance of any Myrinet message layer.

- **Virtual Memory Mapped Communication (VMMC) [38]**: The VMMC layer is designed to support shared memory operations on cluster computers. In this software, the library provides efficient means of transferring blocks of data from the virtual memory of one application to the virtual memory of another application located on a different host. These operations take place with remote DMA operations and require no flow-control mechanisms. VMMC NI firmware is equipped with mechanisms to perform virtual-to-physical address translation, as well as facilities to cache translation results.

- **GM [39]**: GM is an industrial strength message layer from Myricom that provides good performance and is supported on a wide variety of cluster platforms. Like BIP, GM utilizes a rendezvous programming interface that works well with MPI. GM provides rich functionality at both receiving and sending endpoints and uses callback functions to notify applications that message layer operations have been completed. GM requires that all data transferred with the message library be loaded in a block memory that is registered with the library. Registered memory allows the NI to efficiently DMA data between the host and card, with virtual memory translation performed through a simple table lookup.

2.5 The Virtual Parallel-Processing Machine

In addition to providing low-level mechanisms for transferring data between cluster workstations, SAN message layers provide a programming abstraction that allows end users to control a cluster's computational resources. This abstraction presents the cluster as a *virtual*

parallel-processing machine that is capable of running parallel and distributed applications. An example of such a virtual machine is presented in Figure 2.5. In this example, the message layer maintains information about the workstations in a cluster and provides an interface where applications can globally reference any host CPU in the cluster. Therefore, an application running at one host CPU transmits data to another CPU by providing the message layer software with a message that is labeled with the reference identifier for the destination. In current generation message layers, host CPUs are the only resource included in the virtual machine architecture.

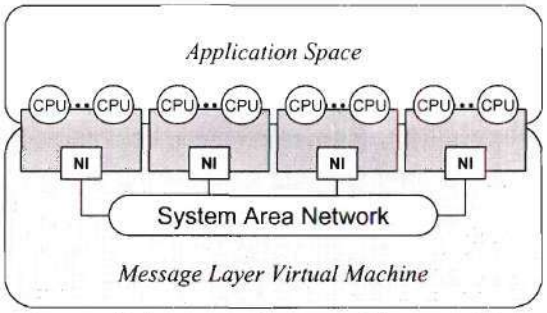


Figure 2.5: The virtual parallel-processing machine architecture provided to end users in current message layers.

While there are many ways in which a virtual machine can be realized for a cluster, a common approach is to load each host in the cluster with an executable program that is part of an overall parallel-processing application. Each executable program contains user-defined functionality for the local host as well as message layer library functions and information about the cluster's global resources. After all hosts in the cluster have executed message layer initialization functions, the virtual parallel-processing machine becomes operational and each host begins processing the application code defined in its local executable program. Maintaining the appearance of a virtual machine is a straightforward process in the message layer after this point, as the message layer must simply service application queries regarding cluster resources and route transmissions to the appropriate cluster resources.

CHAPTER III

MESSAGE LAYERS FOR RESOURCE-RICH CLUSTER COMPUTERS

As discussed in the previous section, cluster computers provide a cost-effective platform for processing distributed applications. If a cluster computer's communication library is visualized as a means of providing a virtual parallel-processing machine for distributed applications, there is one component of the virtual architecture that current generation communication libraries omit: peripheral devices. Traditional cluster communication libraries are designed to transfer data only between host CPUs, not peripheral devices. For these libraries it is assumed that cluster interactions with a peripheral device are performed by a host-level application that resides in the same host as the device. Therefore, in order to interact with a remote peripheral device, an application must communicate with the remote host's CPU and request that an operation be performed on behalf of the application. This method of controlling a peripheral device through a proxy incurs costly overheads that limit the dynamic use of peripheral devices in distributed applications.

The fact that peripheral devices can strongly influence a cluster computer's performance challenges the notion that communication libraries should be designed in such a CPU-centric manner. Peripheral devices are becoming increasingly more powerful, and therefore represent a valuable opportunity for accelerating cluster computer applications. The inclusion of powerful peripheral devices in the cluster architecture results in a new category of cluster computer that we refer to as *resource-rich cluster computers*. Since these clusters exhibit different communication requirements than traditional clusters, it is beneficial to examine the design of new communication libraries that are well suited to these clusters. These libraries specifically allow both host CPUs and peripheral devices to be efficiently utilized as computational resources by

distributed applications. The resulting virtual parallel-processing machine provided by the communication library is depicted in Figure 3.1.

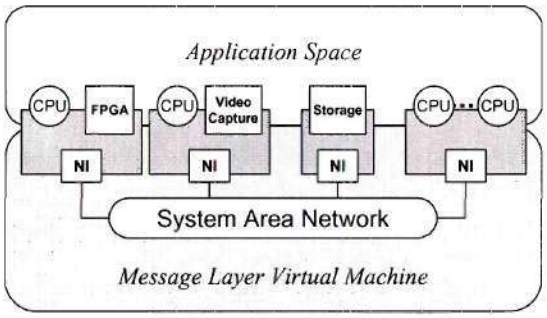


Figure 3.1: Including peripheral devices in the virtual parallel-processing machine architecture provided by the communication library.

This chapter provides the groundwork for designing message layers that are well suited for resource-rich cluster computers. As a first step in this effort, definitions of the hardware environment are provided as well as motivating examples of how these clusters can be beneficial to end users. This work is followed by a discussion of fundamental concepts that influence the construction of the communication library. These concepts are then individually elaborated. Finally, this chapter concludes with a listing of research projects that are related to resource-rich clusters.

3.1 Emergence of Resource-Rich Cluster Computers

Resource-rich cluster computers are clusters in which individual workstations are supplemented with powerful peripheral devices. It is important to examine the characteristics of these clusters in order to determine how message layers should be designed for these systems.

3.1.1 Availability of Powerful Peripheral Devices

In recent years, commercial hardware vendors have constructed a number of powerful peripheral devices that are designed to perform a variety of application-specific operations. One

of the key motivations for this effort has been the need to produce high-performance network servers for the Internet. In answer to market demand, developers have constructed a number of intelligent I/O cards for both LAN and storage operations. For LANs, developers have constructed high-performance network cards that feature embedded processors and multiple physical links to the network. Some of these cards are equipped with firmware that allows common network operations such as TCP connection management to be performed on the network card as opposed to the host. Storage controller cards are also becoming increasingly more powerful due to the active disk [46] and network-attached storage (NAS) [47] efforts. Modern intelligent storage adaptor cards are capable of managing a disk's file system at the controller in a self-contained manner. These storage cards provide a file-level interface to end applications, and do not require the assistance of the host's operating system.

Another area where peripheral devices are becoming more powerful is in multimedia applications. Driven by consumer interest in high-quality video and audio editing, developers have made significant improvements to multimedia peripheral devices. Modern audio and video capture devices can be configured to automatically push high-resolution data samples directly into host memory, allowing data streams to be captured in real time. Some of these cards feature hardware to perform desirable operations such as compression, clipping, and filtering. Other multimedia cards are available for rendering high-quality output for people to observe. Audio playback and video display cards generate output signals from large on-card buffers that can be written to by applications. Some of these output cards feature processing devices that are capable of performing significant computations in real time.

A third area where peripheral devices have become more powerful is in the field of computational accelerator cards. These cards are designed to utilize custom hardware to improve the performance of certain types of computations. Typically, these cards employ digital signal processors (DSPs), field-programmable gate arrays (FPGAs), or even dedicated application-specific integrated circuits (ASICs). Often these cards feature large amounts of high-speed

memory for storing large data sets at the card, and function as a form of co-processor for the host. The common procedure for utilizing a hardware accelerator card is for the host application to pass data to the card, have the peripheral device process the data, and then have the results transferred back to the host. Custom hardware accelerator cards are often useful for processing large streams of data such as multimedia traffic.

3.1.2 Categorizing Peripheral Devices

Based on the previous examples, it is possible to broadly categorize peripheral devices by the manner in which they are utilized. Three common categories include the following.

- **Data Sources and Sinks**: Peripheral devices are often utilized to produce data for the host (i.e., a data source) or store data from the host (i.e., a data sink). Some peripheral devices, such as storage adaptor cards, are capable of performing both of these operations. Typically, these devices do not perform elaborate computations on incoming or outgoing data.

- **Intermediate Processing Elements**: Peripheral devices such as the custom hardware accelerators are primarily designed to process data for the host. Data is typically injected into these cards, processed, and then ejected back to the host system. Incoming and outgoing data rates for these cards do not have to be equal and are dependent on the application.

- **System Bridges**: A peripheral device can also be designed to serve as a form of bridge between two separate systems. The bridge device therefore manages communication between the two systems, performing protocol translations when needed. One example of a bridge is a LAN adaptor card that is utilized to connect the cluster to an application

running at a host that is not part of the cluster. The cluster side of the bridge communicates with a SAN protocol while the external side utilizes a LAN protocol (such as TCP).

3.1.3 Characteristics of Resource-Rich Cluster Computer Hardware

A resource-rich cluster's hardware architecture is similar to traditional cluster computers, with the exception that workstations are equipped with powerful peripheral devices. Physically adding these devices to the cluster is relatively simple, as cards are placed in the available PCI slots of a cluster's workstations. Figure 3.2 depicts the physical architecture of a resource-rich cluster computer. Each workstation features various peripheral devices and is connected to the cluster through a high-performance SAN. This SAN functions as a backbone for communication in the cluster and can be accessed by both host CPUs and peripheral devices.

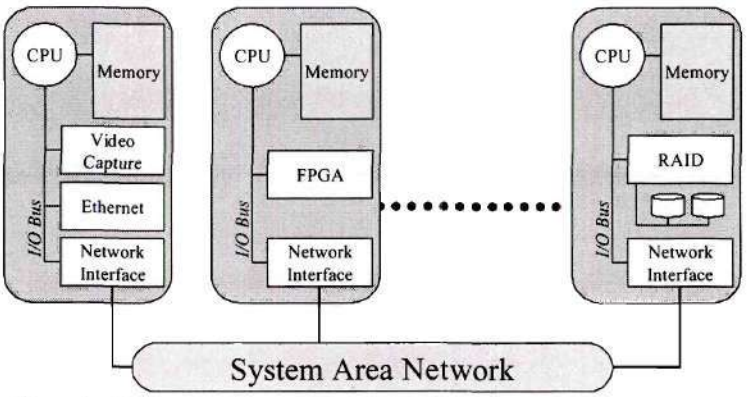


Figure 3.2: The inclusion of peripheral devices in the resource-rich cluster architecture.

While resource-rich clusters are not a radical departure from traditional cluster architectures, there are several unique characteristics that communication library designers must be aware of. The more significant characteristics include the following.

- **Two-levels of Communication Infrastructure**: Communication within a resource-rich cluster takes place in two distinct levels: in the local host context (intra-host) and between hosts using the SAN (inter-host). Intra-host communication can be facilitated with software that intelligently utilizes the host's local I/O system. Inter-host communication requires software that transfers data through both the local I/O system and backbone SAN substrate.

- **Globally-Shared Peripheral Devices**: Resource-rich clusters feature a number of peripheral devices that can be utilized by end applications. While each device in the cluster is owned and managed by the operating system of the workstation in which it resides, it is beneficial for devices to be accessible in the global context of the cluster. The ultimate goal is for any resource to be able to efficiently utilize any other resource in the cluster.

- **Differences in Peripheral Device Capabilities**: Peripheral devices are generally designed to perform specific functions using minimal amounts of hardware resources. While some devices feature programmable embedded processors and large amounts of on-card memory, others may only be equipped with low-speed ASICs configured with simple state machines. Therefore, different peripheral devices have different capabilities. These differences influence the extent to which a device can be integrated into the resource-rich cluster environment and made available as a global resource

- **Limited Local I/O Capacity**: Workstations have a fixed capacity for local I/O operations. In addition to being limited, local I/O bandwidth is generally shared among all peripheral devices in a host. Therefore, it is important that data transfers involving the

local I/O system be orchestrated in an efficient manner. For example, if data is being moved from one resource to another in a host, it should be transferred directly with a single copy as opposed to a two-copy approach where the data is first transferred into an intermediate host buffer.

3.1.4 Resource-Rich Cluster Computer Applications

There are a number of applications that can benefit from the use of resource-rich cluster computers. One motivating example can be found in the field of high-performance network servers. As depicted in Figure 3.3, a resource-rich cluster could be used to implement a tightly synchronized web server that is capable of sustaining high network loads. In this example, each host in the cluster utilizes an intelligent LAN adaptor card to service incoming requests from external clients and an intelligent storage adaptor to house portions of a large database. In order to service incoming requests, the LAN adaptors communicate directly with the appropriate storage controller card using the SAN and the communication library. This form of large-scale server is particularly useful for applications such as digital libraries, where the database is enormous and cannot simply be replicated at each host.

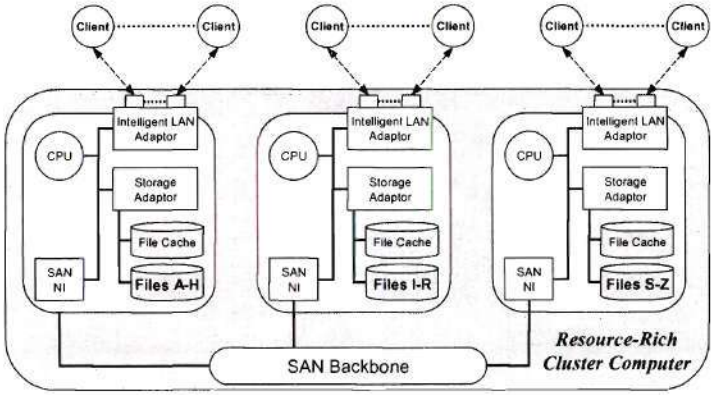


Figure 3.3: An example of a resource-rich cluster functioning as a network server.

Resource-rich cluster computers can also be utilized for applications that process large multimedia data streams. In a full-scale multimedia task, audio and video data is acquired by multimedia capture devices, streamed through various computational resources in the cluster, and then ejected to either storage or output devices. In this application, the communication library must efficiently transfer data between cluster resources in order to meet real-time requirements. A variant of this task is to utilize host CPUs to generate the data streams instead of capture devices. An example of this task is illustrated in the WireGL project [48], where multiple hosts in a cluster generate objects that are combined and rendered to a grid of output displays. These types of operations can be beneficial in scientific applications where a small cluster is utilized to graphically render the computational results of a larger cluster [49,50].

3.2 Design of Message Layers for Resource-Rich Clusters

Physically constructing a resource-rich cluster is a relatively straightforward task: individual components of the architecture can be purchased and assembled from commodity parts that are widely available. A more challenging task is constructing software that allows the hardware to function as part of a single system. Utilizing commodity software such as the open source GNU/Linux operating system is a significant first step in this effort. Linux provides well-defined APIs and built-in device drivers for managing many different hardware devices. However, current generation commodity operating systems are only designed to control a local host, not a cluster of hosts. What is needed is a communication library that is located in or directly above the operating system to provide an application with a means of utilizing the resources that are distributed throughout the cluster. As discussed in the previous chapter, this communication library serves as a means of presenting end users with a form of virtual parallel-processing machine for distributed applications.

Existing communication libraries are inappropriate for resource-rich clusters because they do not provide mechanisms for accessing peripheral devices in the global context. Extending

these communication libraries to provide such access is nontrivial or impossible because the libraries are optimized under the assumption that the NI is controlled exclusively by the host CPU. Therefore, it is necessary to consider how a communication library can be designed with fundamentals that support the needs of resource-rich cluster computers. In this effort, it is beneficial to examine both system level issues as well as practical features that assist end users. These factors influence the design of the communication library and must be addressed in order for a resource-rich cluster to function efficiently.

3.2.1 Definition of a Communication Endpoint

Central to the design of a message layer is the functionality of an endpoint. A communication endpoint is a set of programming abstractions for interacting with a resource. These abstractions define operations for sending/receiving messages to/from other resources in the cluster. For resource-rich clusters, both host CPUs and peripheral devices function as communication endpoints.

In general, the endpoint abstraction has three components. The first is a block of local memory for queuing incoming messages. These queues allow other resources in the local host (e.g., the NI and other local endpoints) to pass messages directly into the endpoint's address space. The second is a set of operations for interpreting and processing messages from the incoming message queues. The third is a set of mechanisms for ejecting an incoming message to another resource in the local host (e.g., the NI or a local endpoint). These mechanisms allow the endpoint to interact with other resources in the cluster.

Simplified endpoints may implement only a subset of the preceding components. For example a low-end peripheral device that functions as a data source only requires mechanisms for sending messages. Similarly, a data sink needs only to accept and process incoming messages. The advantage of implementing all three components of a generic communication endpoint is that the customization of services and functions becomes feasible.

3.2.2 Architectural Design Issues

The architectural characteristics of a resource-rich cluster have a strong influence on the way that communication library software should be designed for these clusters. Key issues that must be addressed include the following.

- **End-to-End Flow Control**: Flow control is utilized as a means of preserving buffer space in the communication library implementation. Resource-rich clusters typically employ a large number of communication endpoints, many of which have limited computational facilities. Therefore, it is infeasible for each endpoint to manage end-to-end flow control for delivering messages. Instead, resource-rich cluster communication libraries should utilize per-hop flow control schemes that simplify the workload of the endpoints.

- **Shared NI Access in a Host**: In resource-rich clusters a host may be equipped with multiple communication endpoints at both the host CPU and peripheral device levels. Each of these endpoints must access the NI to communicate with other endpoints in the cluster. Therefore, the communication library must provide efficient means of sharing the NI among multiple endpoints. These mechanisms must allow multiple endpoints to coherently inject data into the NI. For this task we propose the use of NI-based logical channels.

- **Flexible and Powerful Programming Model**: The communication library must provide a programming model that is flexible enough to serve the diverse needs of cluster users. This programming model must be able to support traditional host-to-host communication mechanisms as well as methods for interacting with peripheral devices in the cluster. The

model must also be extensible, allowing new functionality to be added by end users when necessary. We propose the use of two APIs in the communication library: one for active messages and the other for remote memory operations.

- **Simple Standardized Endpoint Interface**: A variety of diverse cluster resources must implement communication endpoint software. For robustness and portability it is useful if the endpoint interface adheres to a standard form that is universal for all endpoints. Since there is a large amount of diversity in the capabilities of peripheral devices, it is important that this interface be designed in a manner that allows it to be implemented on even the simplest of peripheral devices.

- **Optimizations**: Modern communication libraries are expected to deliver high levels of performance for traditional host-level transactions. While a communication library for a resource-rich cluster trades some performance for increased functionality, the library should still be able to provide reasonable amounts of host-level performance. Therefore, it is necessary to include optimizations in the library when possible for improving performance.

3.2.3 Design Overview

Designing a communication library for a resource-rich cluster requires the construction of appropriate mechanisms to address the preceding design issues. While there are certainly many possible solutions, we define a list of four key design characteristics that can be utilized to provide a suitable communication library. These characteristics are discussed in detail in the following sections and are summarized as follows. First, per-hop flow control can be utilized to address the need for dynamic buffer management in the communication library without complicating the communication endpoint software. Second, the use of multiple logical channels

in the NI allows communication endpoints in a host to share a NI without heavyweight synchronization protocols. Third, an active message style programming interface provides a uniform means by which end users can efficiently utilize peripheral devices. Finally, the programming interface can be supplemented with methods for interacting with remote endpoint memory in order to improve the flexibility of the library as well as its performance.

3.3 Per-hop Flow Control

Reliable communication libraries utilize flow control mechanisms to manage buffer space in the library. Without flow control an incoming message can erroneously overwrite an in-flight message that has not yet been processed. For simplicity, several reliable message layers implement flow control at the host level. This approach can be labeled as endpoint-managed flow control, and requires an endpoint to acquire a flow control credit for the intended destination before it injects a message into the NI. The credit represents a reservation of buffer space along the entire communication pathway in the library (i.e., the sending and receiving NIs and the receiving endpoint). Endpoints must maintain flow control state information and communicate with other endpoints when updating this information.

Endpoint-managed flow control is inappropriate for resource-rich clusters because it complicates endpoint responsibilities. A more appropriate mechanism recommended for resource-rich clusters is to manage flow control on a per-hop basis. In this approach, a message can progress along its communication pathway when enough buffer space is available to receive the message in the next communication stage. While this adds complexity to the design of the communication library, it simplifies the work a communication endpoint must perform in order to interact with the library. A key element of this design is managing flow control between NI pairs. An optimistic approach is suggested for this effort in order to reduce communication latency.

3.3.1 Disadvantages of Endpoint-Managed Flow Control

In endpoint-managed flow control schemes, an endpoint must secure a reservation of buffer space for a message from all of the network elements that will be used to transfer the message before the message can be injected into the network. Rather than perform reservations on-demand, most endpoint-managed flow control schemes use a credit-based reservation system, where network buffers are allocated in advance and assigned to the endpoints in the system. An endpoint has a limited number of credits to communicate with each endpoint in the system and must spend a credit before the communication can begin. After receiving a message, an endpoint must transmit a credit-replenishing reply to the sender. An example of this scheme for a single transaction is depicted in Figure 3.4. The shaded regions in the message queues represent buffer space that is allocated for a transmission during the time between when the message is first transmitted and the reply is received.

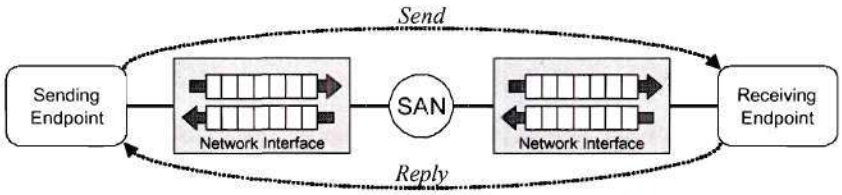


Figure 3.4: Endpoint-managed flow control schemes typically require send/reply messages to be transferred between endpoints to manage flow control credits. The dark buffers in the NIs represent buffer space that is reserved until the send/reply transaction completes.

There are several negative aspects of endpoint-managed flow control for both traditional and resource-rich clusters. First, endpoint-managed flow control schemes perform injection policing at a coarse granularity. It is possible that an endpoint will delay injecting a message into a NI that has buffer space for the message, simply because buffer space has not yet been reserved for the entire communication path. Second, endpoint-managed flow control schemes require credit information to flow between endpoints. This information adds to the network traffic and

may be redundant. Finally, in endpoint-managed flow control schemes, each endpoint is responsible for dynamically managing its own flow control credits. This requirement adds to the work that individual endpoints must perform in order to communicate. As the number of nodes increases in the system, this management becomes a substantial effort that requires larger memory and compute resources. These resources may exceed the capabilities of some peripheral devices, thereby preventing their use in the cluster.

3.3.2 Per-hop Flow Control

An alternative approach to endpoint-managed flow control is for the communication library to perform buffer management on a per-hop basis. In this approach, a message is transmitted to the next stage in the communication path as soon as buffer space is available to receive the message. As illustrated in Figure 3.5, the communication library moves data in three phases: sending-endpoint to sending-NI, sending-NI to receiving-NI, and receiving-NI to receiving-endpoint. Each of these phases employs flow-control mechanisms to guarantee that data is transferred reliably from one stage to the next. This approach is commonly referred to as NI-based flow control because the most challenging aspect of the implementation is the transfer of data between NI pairs.

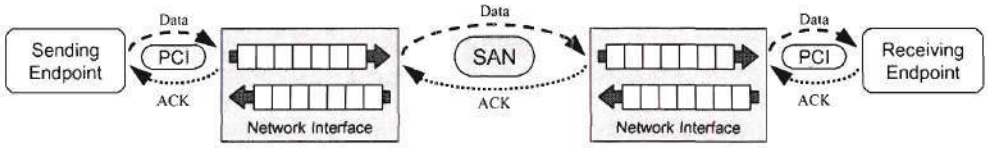


Figure 3.5: Per-hop flow control utilizes synchronization in the communication path to allow messages to progress when buffer space is available.

For resource-rich cluster computers, the primary advantage of per-hop flow control is that it can greatly simplify the software for communication endpoints. In this scheme, an endpoint simply injects a message into its local NI as soon as buffer space becomes available in the NI. From the endpoint's perspective the communication process completes after the injection because the individual network elements in the communication path are guaranteed to reliably transport

the message to its destination endpoint. Unlike endpoint-managed flow-control schemes, the per-hop approach does not require endpoints to maintain state information for in-flight messages. This property simplifies the amount of work an endpoint must perform to communicate in a reliable fashion, and is particularly valuable in cases where peripheral devices with limited capabilities are being used as endpoints.

Another benefit of a per-hop flow control scheme is that buffer space can be managed dynamically. In this approach, a communication element such as a NI makes a decision to accept or reject an incoming message based on whether the element currently has enough buffer space to house the message. Therefore, the hardware devices that propagate a message allocate buffer space on demand as needed by applications. An example of how this trait can be beneficial can be found in a scenario where two endpoints are communicating exclusively with each other at a particular point in time and are not receiving data from other endpoints in the cluster. In this situation the NIs of the elements effectively allocate all of their buffer space for the communication between the two nodes. This buffer space allows more messages to be in-flight between the endpoints at the same time, which improves overlap in the communication pipeline. Endpoint-managed flow control schemes do not allow such dynamic use of resources because allocations are managed at a high level with coarse granularity.

3.3.3 Optimistic NI-NI Flow Control

NI-based flow control mechanisms can be implemented in a variety of manners. A popular approach is to employ a credit-based scheme where each NI has a limited number of credits for communicating with other NIs in the cluster. As observed in the endpoint-managed flow control case, this approach may result in a NI unnecessarily delaying a transmission because acknowledgements have not propagated back to the sender. Another approach is to utilize a scheme where the sending NI requests a reservation of buffer space from the receiving NI before a message is transmitted. This approach is useful in times of high network loads because data

messages are only transmitted when they can be received. However, this approach has poor performance for the common case where the network is not saturated, because a reservation must be acquired before a data message can be transmitted.

An alternative approach to credit-based flow control is to utilize an optimistic transmission scheme. In an optimistic approach, the sending NI transmits a message with the expectation that the receiving NI will be capable of accepting the message when it arrives. The receiving NI transmits a positive or negative acknowledgement to the sender depending on whether the message could be accepted or not. If the sending NI receives a positive acknowledgement, the buffer space allocated for housing the in-flight message is deallocated. If a negative acknowledgement is received, the sender performs a rollback on the outgoing message queue and retransmits the message and all of the following messages that are to the same destination.

An optimistic NI-based flow control protocol has several benefits. First, similar to a credit-based scheme, an optimistic protocol allows a newly detected message to be transmitted without delay. Second, the optimistic approach does not require any form of credit management. Instead messages must be identified and tracked by the NIs. However, this work is normally required by any NI-based flow control scheme. Third, the NIs naturally allocate buffer space in this approach to meet runtime needs. This trait takes place automatically without explicit signaling between NIs. Finally, the optimistic approach allows the network's delivery latency to be overlapped with useful work. The sending NI can begin transmitting a message at a time when the receiving NI cannot accept it. By the time the message arrives at the receiver it is possible that the receiver will be able to accept the message, thereby reducing the latency of delivery.

3.4 Logical Channels

An important characteristic of resource-rich clusters is that there are multiple communication endpoints in a host that need to interact with the SAN. Since a host generally has

more endpoints than NI cards, it is necessary to construct mechanisms that allow the endpoints to share the NI. In traditional approaches, this sharing is performed in the kernel by constructing multiple virtual network interfaces for end applications. Unfortunately, this approach is inefficient for resource-rich clusters because it is difficult to present these virtual interfaces to peripheral device endpoints in an efficient manner.

Without kernel-based NI management, it is necessary to implement synchronization mechanisms in the individual endpoints to guarantee that the NI is accessed in a mutually exclusive manner. Utilizing explicit signaling between endpoints is complex and impedes performance. Therefore, we propose moving the task of managing shared access to the network into the NI. In this approach, the NI employs multiple message queues that are referred to as logical channels. Each endpoint has exclusive ownership of a small number of the NI's logical channels. The endpoint utilizes these logical channels as virtual interfaces for communication with the network. The task of mapping the logical channels onto the physical network is dynamically performed by the NI. In addition to providing a sharable means of low-latency communication, logical channels can also be utilized by applications to provide isolation between different types of network data streams and allocate bandwidth among peripheral devices.

3.4.1 Sharing Network Access through Kernel Management

For traditional networks such as Ethernet, the kernel is utilized as a means of sharing a physical NI card with multiple applications. As depicted in Figure 3.6, the kernel has exclusive ownership of the NI and provides virtual communication interfaces for multiple application endpoints. The kernel therefore must merge the messages injected by endpoints into a single outbound NI queue and distribute incoming messages from the network to the proper endpoints. In addition to providing a scalable means of sharing the NI, this approach protects end applications from each other by insulating the applications from the low-level hardware.

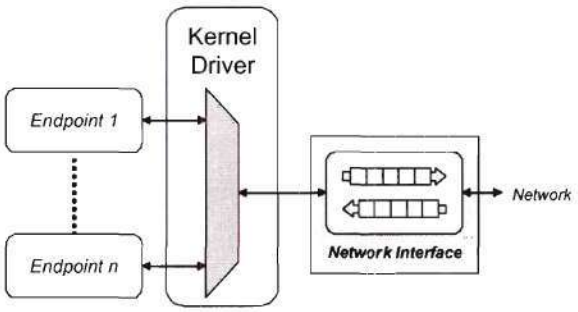


Figure 3.6: The traditional approach to providing shared access to a network device through the use of the kernel.

Utilizing the kernel as a means of sharing the NI is impractical for resource-rich clusters due to the types of network interactions that are utilized in these clusters. The primary problem with relying on the kernel to manage the NI is that the communication interfaces provided by the kernel are designed to operate with host-level endpoints, not peripheral device endpoints. Adapting a peripheral device to operate with these interfaces is difficult and inefficient. The peripheral device would have to route all of its network transactions through the kernel and utilize costly interrupts to invoke the necessary kernel operations. This process requires extra data copies and taxes the memory and I/O systems of the host. Another disadvantage of utilizing the kernel to manage the NI is that host-level endpoints must invoke kernel calls for network operations. Since kernel calls can be relatively expensive operations, it is beneficial if shared access to the NI can be accomplished without involving the kernel driver.

3.4.2 Sharing Network Access through Logical Channels

Another means of sharing the NI with multiple endpoints is simply to remove the dependencies that exist between endpoints that interact with the NI. One such approach is to implement a small number of independent message queues or logical channels in the NI. Each of these logical channels is assigned to an endpoint in the host when the system is initialized. Because an endpoint has exclusive ownership of its logical channel(s), it can send and receive messages without having to synchronize with other endpoints in the system. The NI in this

approach is responsible for mapping the logical channels onto the physical network at runtime through the use of a simple scheduling algorithm. An example of this approach is illustrated in Figure 3.7.

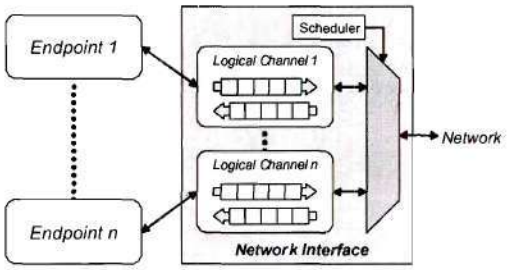


Figure 3.7: Utilizing multiple logical channels in the NI to provide shared access to the network.

There are several benefits to using logical channels as a means of providing shared access to the NI. First, this approach removes the need for any form of direct synchronization between endpoints that are interacting with the NI. An endpoint can begin injecting data into the NI as soon as buffer space is available in its logical channel. Second, endpoints interact directly with the NI. Unlike kernel-managed approaches, an endpoint transfers data directly into the NI without intermediate buffering. Finally, this approach provides a simple interface for communication that can be implemented for many peripheral devices without complex management mechanisms.

There are two primary disadvantages to utilizing logical channels in the NI. First, there is a finite amount of buffer space available in the NI for implementing logical channels. As the number of logical channels in the NI increases, the buffer capacity of each logical channel decreases. Therefore, it is expected that most resource-rich cluster users will allocate only a few logical channels in the NI (i.e., roughly one per endpoint). Second, the presence of multiple logical channels in the NI has a negative impact on the performance of the NI. Because the NI must spend time managing each logical channel, the NI's workload increases as more logical channels are added to the NI. Additionally, NI firmware becomes more complex when it is

switched from servicing a single queue to multiple queues. This complexity results in extra NI operations, which detract from performance.

3.4.3 Application-Level Use of Logical Channels

In addition to allowing multiple endpoints to share the same NI, logical channels can be utilized by end applications as a simple means of separating traffic streams. For this use an application requests two or more logical channels from the NI and assigns different traffic streams to each channel. Data streams on different logical channels are isolated from each other due to two properties of logical channels. First, each logical channel has a private allocation of buffer space in the NI. Therefore, if one traffic stream saturates its logical channel's buffer space, other logical channels are not affected. Second, in-order delivery in the communication library is guaranteed only for messages that belong to the same logical channel. This property is necessary in order to allow the NI to implement a fair scheduling algorithm in which each logical channel has equal access to the NI. The result is that a message injected into an empty logical channel does not have to be delayed until all of the messages in other queues are transmitted.

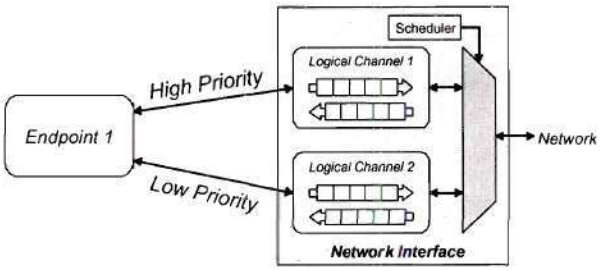


Figure 3.8: Utilizing multiple logical channels to prioritize messages.

An example of how the presence of multiple logical channels in the NI can be exploited by an end application is illustrated in Figure 3.8. In this example, an endpoint obtains two separate NI logical channels for two types of network traffic. The network bandwidth made available for each logical channel is controlled through a scheduler implemented in the NI.

3.5 Active Message Programming Interface

One of the defining characteristics of a communication library is the programming interface that is provided to the end user. Users of resource-rich clusters require a flexible programming interface that can easily be extended to support new functionality. As a means of addressing this need, we propose constructing the communication library with two types of programming interfaces: one that employs active message style processing (described in this section) and another that provides a means of interacting with remote memory (described in the following section). For the active message interface, each communication endpoint is equipped with various function handlers for processing incoming messages. Whenever an endpoint injects a message, it specifies the function handler the receiving endpoint should use to process the message when it arrives. In addition to providing a powerful means of controlling computations in a distributed processing environment, the active message programming interface is well suited to controlling peripheral devices in a resource-rich cluster. In this effort, peripheral device functionality is encapsulated as a set of active message function handlers that all endpoints in the cluster can utilize.

3.5.1 Active Message Operation

The fundamental concept of active messages is that a message contains both application data and information describing how the receiver should process the message. While active network research [51] has discussed encoding complex processing instructions into active messages, a more common approach is for endpoints to be equipped with predefined methods for processing messages. These methods are commonly referred to as function handlers. When the communication library is initialized, each endpoint publishes a list of its function handlers to other endpoints in the system. At runtime when an endpoint injects a message into the communication library, it must specify the function handler the receiver should use to process the

message. Endpoints are responsible for monitoring incoming message queues and processing new messages with the appropriate function handler.

The appeal of an active message interface is that it provides basic programming mechanisms that are both powerful and flexible. As opposed to simply transferring data between endpoints, active messages provide a means of invoking actions at remote endpoints. These actions can be utilized in an active manner to remotely control the behavior of an endpoint. For example, a message handler can be designed to spawn, modify, or terminate a computation in an endpoint. With these types of operations, a user can directly control the flow of computations in a distributed system. Active messages can also be utilized in a passive manner where a remote endpoint's state is not affected by the execution of a handler. For example, a handler can be designed to simply return the remote endpoint's current dataset to the sender of the message. From the remote endpoint's perspective, the processing of the function handler takes place in the background and does not affect the endpoint's main thread of execution.

The original active message specification [52] is not directly applicable for resource-rich clusters because it is only designed to operate with homogeneous endpoints. Therefore, it is necessary to construct a more robust specification that allows diverse endpoints to interact with the active message interface. Three issues must be addressed in this specification. First, function handlers must be managed in a dynamic fashion by the communication library. It is not practical to statically configure endpoints with a list of the cluster's handlers because endpoint software would have to be recompiled every time a new application defined new handlers. Second, handlers should be referenced with useful labels, such as string and integer identifiers. In addition to being portable, these identifiers help make programs more readable (e.g., referencing a handler by the string "handler_compute_PI" has more meaning than a pointer to the handler's virtual memory address). Finally, active messages should be formatted in a manner that is interpretable by endpoints with different byte orders and word alignments. Constructing a single message format that takes into account these characteristics provides standardization among endpoints and

allows an endpoint to transmit a message without having to know the processing characteristics of the destination endpoint.

3.5.2 Utilizing Active Messages with Peripheral Devices

The active message programming abstraction is particularly useful for resource-rich clusters because active messages can be used as a simple but powerful means of controlling peripheral devices. In this approach, active message function handlers are defined for all of the actions that a peripheral device can perform. Endpoints in the cluster can therefore trigger an operation at a peripheral device by transmitting an active message to the device containing a reference to the function handler that needs to be invoked. Figure 3.9 illustrates an example of how a host-level endpoint can interact with an intelligent storage controller at a remote host using the active message programming interface. In order to obtain data from a desired file, the host CPU transmits an active message that contains the name of the file and the function handler id `am_fetch_file()`. Upon receiving this message, the storage controller accesses the file and generates an active message with the handler `am_return_file_data()` and the requested data. The transaction completes when the host CPU endpoint receives this reply and stores the data accordingly.

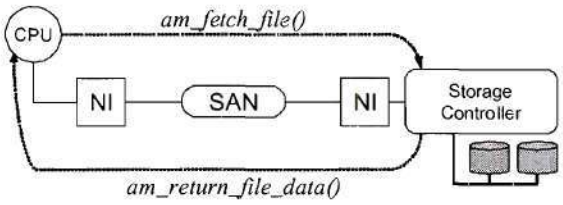


Figure 3.9: Active messages can be used to facilitate an API for a peripheral device.

Using an active message programming interface to control a cluster's peripheral devices is beneficial for a number of reasons. First, it is relatively easy to integrate new peripheral devices into the cluster using this interface. Designers simply construct a series of card-specific active message function handlers for a peripheral device and provide references for the handlers to

application designers. Second, the active message interface serves as a universal communication substrate upon which multiple APIs can be layered. In this system, each peripheral device has its own API that is comprised of card-specific active message handlers. Endpoints therefore invoke a peripheral device's API operations by transmitting the corresponding active messages using the communication library's message passing functions. Finally, the active message interface is beneficial for controlling peripheral devices because it allows an endpoint to utilize a peripheral device no matter where the resources are physically located. Since API operations are separated from communication mechanisms, users can issue API operations knowing that the communication library will automatically perform any routing in the cluster that is necessary.

3.6 Remote Memory Programming Interface

The second programming interface proposed for a resource-rich cluster's communication library is one that allows an endpoint to directly interact with the memory of a remote host. This remote memory interface is designed to provide an efficient means of transferring data from one endpoint to another. A remote memory programming interface can also be utilized as a means of performing custom interactions with a cluster's peripheral devices. This functionality is especially beneficial because it can be used to allow an endpoint to control a peripheral device for which it is impossible to construct endpoint software. Issues involved in implementing a remote memory interface include integrating the interface into a library that also supports active messages, and providing the functionality to translate an endpoint's virtual address space into a physical address space.

3.6.1 The Need for a Remote Memory Interface

While active messages provide a flexible communication interface for end users, there are certain operations for which active messages are not ideal. For example, consider the case where an application needs to transfer a large block of data from one endpoint to another. In the

active message approach, the data is encapsulated in an active message that is marked with a data transfer function handler. The receiver processes this message by copying the message's payload data to the memory location specified in the message's arguments. This process is inefficient because two transfers are involved in the receiving endpoint: one from the NI to the endpoint's incoming message queue and another from the message queue to the target address. With a remote memory programming interface, it is possible for the NI to transfer the data directly to the message's target memory address.

Remote memory operations are also valuable in resource-rich clusters because they can be used to support low-level interactions with remote peripheral devices. The architecture of several peripheral devices makes it impossible to construct endpoint software that would allow these devices to participate as intelligent resources in the cluster. For example, video display adaptors are generally designed as data sinks, and therefore it is unlikely that endpoint software can be constructed for such adaptors. However, it is still desirable for other resources in the cluster to be able to interact with the adaptor. With a remote memory interface, it is possible for an endpoint to transmit image data into the video adaptor's frame buffer. These forms of direct memory transactions can be useful in a number of resource-rich cluster applications where data must be deposited into distributed memory locations in an efficient manner.

3.6.2 Remote Memory Interface

From an end user's perspective, a remote memory interface is relatively straightforward. The user supplies the interface with the source and destination addresses, the direction and size of the transfer, and the identifier of the remote endpoint. The communication library is then responsible for transferring the block of data utilizing the most efficient means available. In the case of multiple transactions, remote memory transfers are completed in the order that they are issued. Remote memory interfaces generally allow both read and write operations. Write operations are simpler to implement, as data is simply streamed from the sender's address space

to the receiver's. Read operations are more complex, as the sender must issue a message that fetches data from the receiver's address space. Results are returned in a reply message and written into the sender's address space.

A host system operates with two related address spaces: virtual memory addresses and physical (or bus) memory addresses. The API for a remote memory interface must be designed to allow users to universally reference memory distributed throughout the cluster. As a means of simplifying the interface for end users, utilizing virtual memory references are preferred since a memory reference is the same in both the host where the memory resides and remote endpoints. As a consequence, it is necessary for the communication library to be capable of internally transforming virtual addresses to physical addresses that the NI can utilize. The library must also provide mechanisms to prevent a memory region from being moved by the kernel (e.g., a page fault) during a memory operation. Finally, it is beneficial for a remote memory interface to be able to operate with physical addresses, in order to provide efficient direct access to memory-mapped devices.

3.7 Related Work

Various aspects of this thesis are related to themes found in other research projects. A common goal of all these efforts is to enhance cluster computer performance by incorporating powerful peripheral devices within the hosts. Researchers have designed custom I/O architectures to support this functionality, as well as specialized software to integrate specific peripheral devices into the communication model. This thesis is distinguished from past work in that it provides a general framework for integrating all manner of peripherals into a low-latency message layer. Device-specific functionality is separated from network-specific functionality to produce an extensible design and significantly improve the productivity of the application designer with minimal sacrifices in performance. The following efforts represent state-of-the-art research being performed involving resource-rich cluster computers.

3.7.1 InfiniBand

Industry is currently developing a new generation of I/O fabric called InfiniBand (IB) [53] that can potentially serve as a means of constructing resource-rich cluster computers. IB is primarily designed as a turnkey solution for a number of high-end server issues. It provides a high-performance communication substrate that functions as a system area network, a storage area network, and a distributed I/O system. In addition to featuring expandable multi-gigabit links, IB defines a protocol for efficient communication between peripheral devices and host CPUs. This protocol could therefore be utilized by end users to allow peripheral devices to be integrated into the cluster's computational model. Therefore, IB represents a promising communication substrate for resource-rich cluster computers in the near future.

A fundamental difference between the work presented in this thesis and InfiniBand can be found in the hardware architectures used for these systems. In the work presented in this thesis, it is assumed that cluster computers will be constructed with commodity hardware that is currently available. This approach utilizes existing hardware and defines flexible mechanisms for addressing the performance obstacles of the hardware. In contrast, IB is a complete overhaul of the I/O architecture found in current generation clusters. With the freedom to redesign the low-level architecture of the cluster computer, IB designers constructed a new hardware environment that is conducive to high-performance communication. The difficulty in this approach is public acceptance: the success of IB as a communication substrate depends on the generation of new hardware products that provide better performance than current products. In comparison, the work in this thesis utilizes current generation hardware and can be adapted to exploit gains in faster network substrates as they become available.

3.7.2 Extensions to the GM Message Layer

In recent years Myricom's GM message layer has become the de facto standard for traditional clusters interconnected with Myrinet hardware. GM exhibits a number of basic

characteristics that make it a desirable starting point for constructing a message layer for resource-rich clusters. In addition to utilizing NI-based flow control mechanisms, GM supports multiple concurrent users of the NI through the use of multiple work queues. While GM does not specifically support active messages, it provides a generic programming interface that allows other APIs to be layered on top of it. GM also provides mechanisms for low-level interactions with the virtual memory system, which can be extended to provide remote memory operations.

While GM can be extended, we note that there are fundamental design issues that make the adaptation of this message layer to resource-rich clusters nontrivial. The primary difficulty is that the basic means for a communication endpoint to interact with the message layer is through a work queue. In this approach, an endpoint inserts a reference to a message that needs to be injected. When the NI is ready, it processes the work entry by pulling the message into the NI. It then inserts a notification message into the endpoint's completion queue that specifies that the host memory housing the message can be reused by the application. While suitable for host-level endpoints, this process may not be appropriate for some peripheral devices because it requires the peripheral device to maintain a block of data until the NI has retrieved it. Peripheral devices generally have limited memory and resources to manage such interactions.

3.7.3 OPIUM

GM has been extended in previous work to allow the NI to directly interact with multiple peripheral devices. In the OPIUM [54] project, researchers examined the extension of GM to support SAN interactions with a specific SCSI card. The goal of this work is to minimize the number of traversals that take place across the PCI bus for servicing network requests for file data. The researchers accomplished this task by modifying the storage card's device driver so that it could issue DMA operations to route file data directly to a buffer located in NI card memory. In later work OPIUM was modified to allow the NI to directly write data into a video display card's frame buffer [55].

While OPIUM provides the first steps in allowing peripheral device interactions with the SAN, the work is directed at providing an ad hoc solution for two specific devices. While the modifications that allow the host to control SCSI interactions with the network is certainly useful for network-attached storage efforts, the work is card-specific and may not be suitable for other peripheral devices that could be used in the cluster. Likewise, the work with integrating a video display card into the communication library does not demonstrate an interaction with an intelligent peripheral device, because a display card's frame buffer can trivially be written by any PCI device in a host. This work however, does provide a motivation to improve the flexibility of the communication library in order to allow peripheral devices to be utilized in an efficient manner by cluster applications.

3.7.4 Adaptive Computing Machines

Another area of work that is related to this thesis is the field of Adaptive Computing Machines (ACMs). In ACMs, a number of field-programmable gate arrays (FPGAs) are utilized as a means of processing an application with dedicated hardware [56]. In this approach, the FPGAs are configured to emulate application-specific circuitry that can rapidly perform an application's computations. Unlike ASICs, which cannot be reprogrammed, FPGAs can easily be configured to emulate different circuits as needed by the application. While ACMs are not particularly useful for general-purpose applications, they can be valuable for applications that require complex computations to be performed in real time [57,58].

Initial work in ACMs resulted in custom hardware that employed arrays of FPGAs [59]. Observing that these systems were expensive to construct, researchers in the late 1990's began investigating the use of multiple commercial FPGA cards to function as an ACM. In the Tower of Power project [60], sixteen x86 workstations were equipped with commercial FPGA cards and linked using a Myrinet SAN. The researchers investigated the use of existing Myrinet software to allow data to be transmitted between FPGA cards [61]. This effort resulted in the computational

environment where researchers could effectively utilize the distributed FPGA cards as part of an ACM.

One of the hardships that researchers had to face in the Tower of Power project is transporting data between FPGAs in the cluster. Rather than implement new communication software, the researchers layered their programming interface on top of a Myrinet implementation of MPI. The researcher's software therefore utilizes the host CPU to manage application interactions with an FPGA card. While simplifying the design effort, this approach delays communication and results in extra traversals of the host's I/O bus. Additionally, selecting MPI as the base programming interface makes it challenging to modify the system to support direct interactions between the FPGA card and the NI. MPI endpoint software is complex and therefore nontrivial to implement for an FPGA card. However, this work indicates that there is a definite interest in utilizing peripheral devices in a cluster to perform custom computations.

CHAPTER IV

MESSAGE LAYER IMPLEMENTATION: GRIM

The design considerations outlined in the previous chapter have driven the implementation of an extensible message layer for resource-rich clusters. This message layer is known as GRIM: the General-purpose Reliable In-order Message layer. GRIM has evolved considerably since its initial development in 1997, but has always provided three basic features: NI-directed flow-control, NI-based logical channels, and an active message style programming interface for interactions between cluster resources. This work was extended in later versions to provide an additional remote memory interface for efficiently transferring data between endpoints. GRIM has been used to incorporate multiple peripheral devices into the cluster-computing environment. This chapter examines the core functionality of the GRIM library, specifically focusing on low-level implementation details that shaped the library. The core's end-to-end performance for host-CPU interactions is presented in the next chapter, which is followed by details of the library's use in peripheral device interactions.

4.1 Overview of GRIM

GRIM is a message layer for resource-rich cluster computers. The current version utilizes a Myrinet SAN for interconnecting host systems, although GRIM could be adapted for use with other network substrates. The GRIM communication library is comprised of user-space software, kernel-space device drivers, and peripheral device firmware. The library currently utilizes the Linux 2.4 kernel, although previous kernels have been used during GRIM's evolution. In order to minimize the impact of an ever-changing Linux kernel, the majority of GRIM's functionality is constructed in user-space software and NI-based firmware. In recent versions of GRIM, the

LANai 4 NI firmware has been ported for use with the newer LANai 9 version of the Myrinet NI card. This adaptation has resulted in significant performance improvements due to advances in the LANai's architecture.

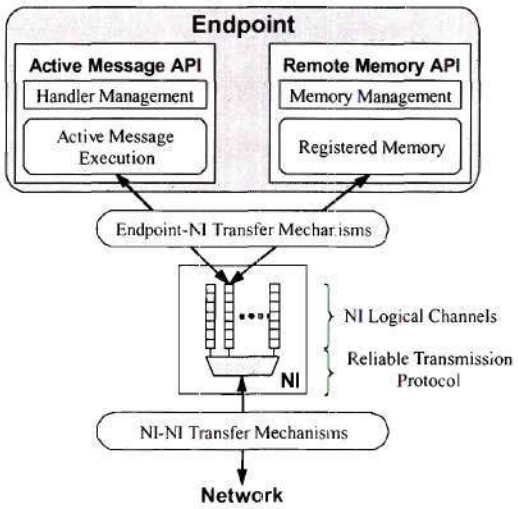


Figure 4.1: GRIM utilizes logical channels and a reliable transmission protocol at the NI and provides two different programming interfaces for end applications.

The organization of GRIM's core components is depicted in Figure 4.1. Starting at the lowest level of the software, GRIM utilizes a NI-based reliable transmission protocol for delivering messages in order between NIs. This protocol is optimistic in that messages are transmitted with the expectation that the receiver can accept the message. Messages are supplied to the reliable transmission protocol from a small collection of logical channels located in NI memory. Each logical channel provides a virtual communication interface for an endpoint in the host and serves as a place for buffering messages that are in transit. At the application level, an endpoint can utilize two programming interfaces for interacting with a logical channel. The active message interface allows the sender to label an outgoing message with the function handler the receiver should use to process the message when it arrives. This interface provides a queue in the endpoint for buffering incoming message that cannot be processed immediately by the endpoint. The second programming interface is for remote memory operations. Memory used with this interface must be registered with the communication library. The interface provides multiple

mechanisms for translating virtual addresses to physical addresses, and allows the NI to process incoming remote memory messages directly.

4.1.1 Message Structure

The fundamental unit of communication in the GRIM communication library is a data message. GRIM uses a single message format for all of its operations, and therefore it is instructive to examine the common message format. A data message in GRIM is comprised of two segments: a message header and payload data. The message header consists of eight 32-bit words that are formatted in network byte order (i.e., big endian). The first four words contain information that is used to deliver the message. These words are arranged in a manner that allows a receiver NI to begin processing an incoming message as soon as possible with consequent reductions in overall latency. The remaining portion of the header allows users to include up to four 32-bit data values in a message. These values serve as function handler arguments in active messages, and specify transfer instructions in remote memory messages. The second segment of a message is its payload data section. This region allows a large (approximately 64 KB) block of data to be associated with each message.

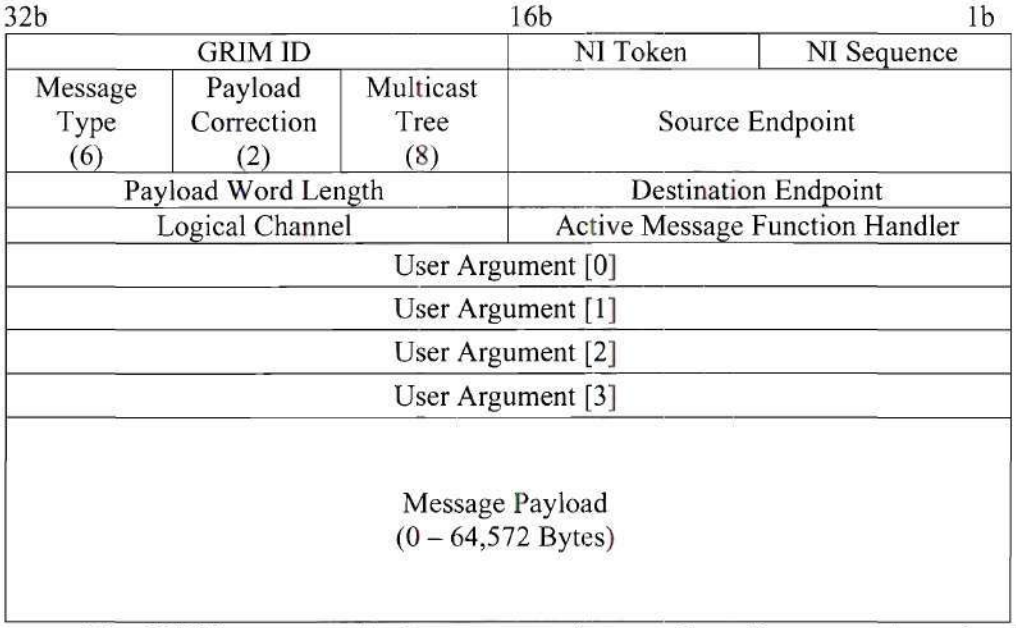


Figure 4.2: GRIM uses a single message format for all transactions in the communication library.

Figure 4.2 provides the format of a GRIM data message. The individual fields in a message are defined as follows.

- **GRIM ID**: This field identifies a message as belonging to the GRIM communication library. A NI in GRIM only examines messages that are labeled with this identifier. GRIM is registered with Myricom and has been assigned the identifier 0x0636.
- **NI Token**: The token ID is supplied by the NI and utilized to reference an in-transit message.
- **NI Sequence Number**: The sequence number is created by the sending NI and utilized by the receiving NI to verify delivery order in the reliable transmission protocol.
- **Message Type**: The type field is used to distinguish between different forms of messages used in the library. Message types include active messages, remote memory operations, NI control messages (e.g., ACK or NACK), and multicast operations.

- **Payload Correction**: Internally GRIM aligns payload data on 32-bit boundaries. The payload correction value is used to truncate the length of the payload to match the number of bytes specified by the sending endpoint.
- **Multicast Tree**: This value is used in multicast operations to identify the multicast tree a message belongs to. Multicast operations are discussed in Section 8.1.
- **Source Endpoint**: The source field identifies the endpoint that created the message.
- **Payload Word Length**: This field specifies the number of 32-bit words that are in the payload section of the message.
- **Destination Endpoint**: This value identifies which endpoint in the cluster the message should be delivered to.
- **Logical Channel**: An endpoint that transmits a message can assign a 16-bit logical channel identifier to the message. This identifier is used to group messages that must be delivered in order by the message layer.
- **Active Message Function Handler**: This value identifies which active message function handler should be used to process the message at the receiving endpoint.
- **User Arguments [0-3]**: Users can specify up to four 32-bit arguments to be included in an active message. These fields are utilized in remote memory operations to specify the addresses to be utilized in a data transfer.

Internally GRIM uses an abbreviated version of the message header for control messages (ACKs and NACKs). Control messages are only 8 bytes long and contain basic information to allow updates in the reliable transmission protocol.

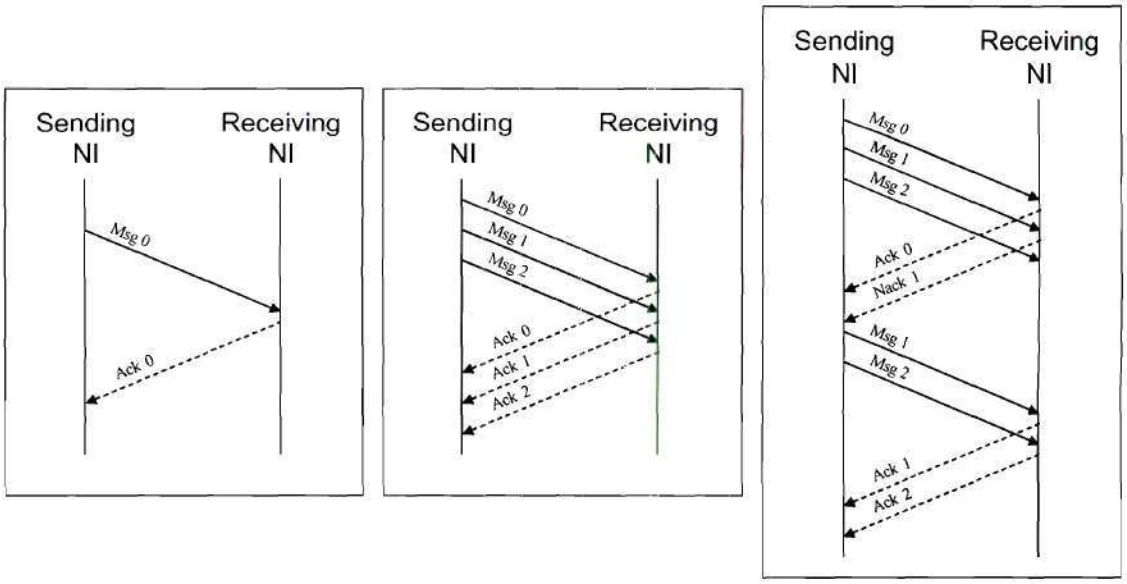
4.2 NI-Based Reliable Transmission Protocol

One of the key characteristics of GRIM is the use of a reliable transmission protocol for transferring messages between NIs in the cluster. GRIM utilizes a variation of the "go-back-n" protocol that optimistically transmits a message with the expectation that the receiver will be able to accept the message when it arrives. If the receiver cannot accept the message, the protocol automatically performs rollback on the sender's message queue and retransmits messages as needed. In order to facilitate this operation GRIM utilizes control messages and maintains state information for each message queue. Since control messages utilize the same network as data messages, it is possible for a poorly designed system to reach deadlock. GRIM avoids this condition by buffering outgoing control messages when a response to an incoming message cannot be transmitted due to a busy outgoing link. Performance measurements of GRIM suggest that the NI-based reliable delivery protocol is advantageous over other approaches. When compared to system employing host-based flow control, GRIM allows endpoints to inject a greater number of outstanding messages, thereby reducing the sending endpoint's injection overhead.

4.2.1 Protocol

The NI-based reliable transmission protocol implemented in GRIM is a variant of the traditional "go-back-N" protocol [62] for retransmissions and operates as follows. The successful transmission of a message is depicted in Figure 4.3(a) with three steps. (1) When a sending NI observes a new message to send, it marks the message with the next sequence ID for the destination NI and a token ID that can be used to reference the message. It then transmits the message to the destination and increases the sequence number register for the destination. (2) When the message reaches its destination NI, the receiver NI verifies that the message's sequence number matches the expected value for the sender. If it does and the receiver has enough buffer

space, the message is accepted and a positive acknowledgement (ACK) with the data message's token ID is transmitted to the sender. (3) When the ACK reaches the sender NI, the NI uses the token ID to mark the corresponding message in the message queue as acknowledged. If the message is the oldest outstanding message, the NI walks through the queue structure freeing buffer space for all acknowledged messages until it reaches an unacknowledged message, a message that has not been transmitted, or the back of the outgoing queue. With this protocol the sender is allowed to have multiple messages to a destination in-flight at the same time as illustrated in Figure 4.3(b).



(a) (b) (c)

Figure 4.3: (a) Acknowledged transmission of a single message between NI pairs. (b) The optimistic transmission and acknowledgement of three messages. (c) The optimistic transmission of three messages with retransmission due to the lack of buffer space.

In the case where the receiver cannot accept an incoming message due to a lack of buffer space, the protocol uses a negative acknowledgement (NACK) control message to force the sender to retransmit messages. Figure 4.3(c) illustrates such a case where a sender optimistically transmits three messages with sequence numbers and token IDs of 0, 1, and 2. After the first message is accepted the receiver is unable to accept data due to a lack of buffer space and must

transmit a NACK for message 1. The receiver at this point drops incoming messages from the sender until message 1 is received and buffer space is available. When the sender receives a NACK it must rollback the outgoing queue to the message referenced in the NACK. It then retransmits the message and all following messages in the outbound queue that are for the same destination. Messages for other destinations are not retransmitted. The procedure is repeated until all messages are reliably delivered.

The implementation of this protocol in GRIM takes advantage of Myrinet's reliability guarantees and the fact that source-routed messages do not get reordered in the network. While other network substrates do not exhibit these characteristics, the implementation can be extended to function under different operating conditions without significant changes to the protocol. For networks that can re-order messages (such as an Ethernet LAN with multiple routers), the sequencing portion of the protocol preserves in-order delivery. The arbitrary dropping of packets on the other hand requires the protocol to be modified with timeout mechanisms. These mechanisms automatically retransmit data and control messages after a specified amount of time under the assumption that the network has lost a message.

4.2.2 Managing In-flight Messages for Different Queue Mechanisms

An important part of implementing a NI-level reliable transmission scheme is constructing mechanisms that allow the NI to manage multiple in-flight messages. These mechanisms require the sending NI to maintain a database of in-flight messages that is populated with information that can be used to coherently perform rollback on a message queue when a message needs to be retransmitted. This work is highly dependent on the types of queuing mechanisms that are used to buffer messages in the sending NI. Over its evolution, three different styles of queuing have been used for GRIM, as illustrated in Figure 4.4(a-c). In a slotted approach (a), queue buffer space is evenly divided into fixed-sized slots for housing individual messages. An append-style approach (b) differs in that messages can be placed in the queue without any gap

between successive messages. Finally, in a hybrid-approach (c), a combination of the previous two mechanisms is used. In this approach, a message's header is stored in a slotted message queue, while its payload is stored in a separate append-style queue.

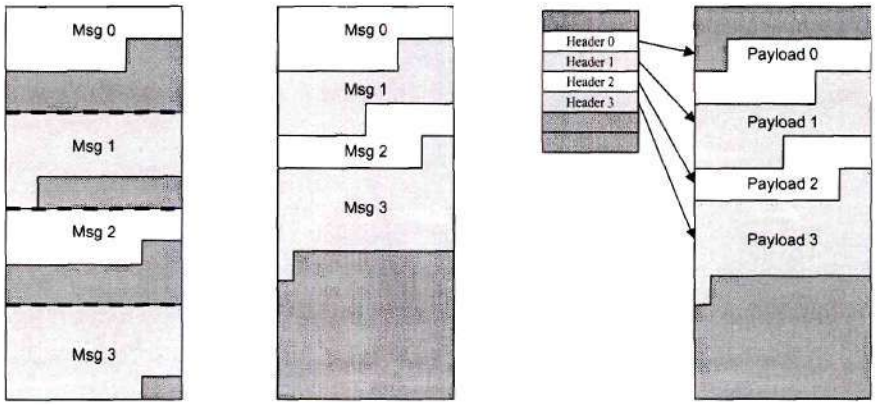


Figure 4.4: Three approaches to queue buffer management include (a) a fixed-sized slot queue buffer, (b) an append-style approach, and (c) a hybrid approach.

The advantage of a slotted approach to queuing messages is that messages always begin at specific locations in the queue. In addition to simplifying the management of the queue, the NI can easily store state information for a message in its queue slot. For example, the first word in a message slot can be reserved for housing the message's current acknowledgement status (e.g., not sent, sent, acknowledged, or unused). The sending NI can then easily walk through the queue structure when rollback is performed, and make necessary state updates by modifying specified values in each slot. The downside of this approach is that queue buffer space is not used efficiently when messages are not the maximum transfer size. Because of the limited amount of buffer space available in the NI, this approach is not utilized in GRIM.

The append-style message queue makes more efficient use of a queue's buffer space by allowing messages to be placed in the queue without wasting space between messages. The difficulty with this approach is that messages are not stored at fixed locations in the queue. Therefore, accessing a particular message in the queue is nontrivial because the NI must

sequentially walk through the queue, examining each message to determine the starting address of the next message. This operation is expensive and makes storing state information in the message queue impractical. Instead, state information for in-transit messages can be stored in a separate data structure by the sending NI. Recent versions of GRIM employ an append style of queuing and store message information in a structure called the scoreboard. When a NI detects a new message in the outgoing queue, it records information about the message (such as its memory location) in a new scoreboard entry. Because scoreboard entries are at fixed offsets, the NI can easily walk through the scoreboard when processing incoming control messages.

The last style of message queue used in GRIM is a hybrid-approach where a slotted queue is used to store a message's header and an append-style queue is used to store a message's payload. This approach is advantageous because (i) in-flight messages are easily managed because state information is stored in the slotted header queue and (ii) large messages are efficiently stored in the append-style payload queue. Although the hybrid approach was used in early versions of GRIM, it had to be abandoned due to a few shortcomings. The first issue is that endpoints in this approach must maintain two sets of queue pointers to interact with an outgoing message queue. This requirement complicates endpoint software and does not match the design goals of a resource-rich cluster. A second and more serious issue is that it is difficult to protect this approach from network deadlock. Since a message is housed in two separate memory regions, the NI must perform two separate DMAs when transferring a message to the network. If the NI firmware blocks the transmission of the second DMA until the first DMA completes, a cyclic dependency is formed and it is possible to reach deadlock. This form of deadlock was observed in the early versions of GRIM and therefore the hybrid approach was dropped in favor of the previous append-style approach. The append-style approach allows a message to be transferred to the network with a single DMA.

4.2.3 Avoiding Deadlock Caused by Control Messages

An important aspect of implementing a communication protocol is constructing it in a manner that does not lead to deadlock. While the network itself may be deadlock free, poorly designed reliable transmission protocols for the NI may result in cyclic dependencies between the NIs that prevent forward progress in the system. The primary hazard is that a NI must inject an ACK or NACK message back into the network upon receival of a data message. If the NI cannot accept a new message until the control message is dispatched, a dependency is formed between incoming and outgoing network links. An example of how this dependency can lead to deadlock is pictured in Figure 4.5(a). In this example, two NIs transmit data messages to each other at the same time in a congested network. Both NIs accept their incoming messages and must transmit reply messages for the receive process to complete. However, because one NI cannot proceed until the other completes the injection of the control message, neither can make progress and the result is deadlock. This phenomenon was infrequently observed in early versions of GRIM, even with small network configurations.

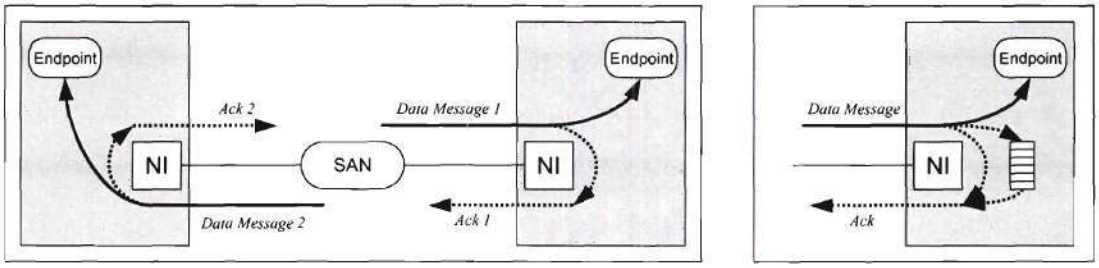


Figure 4.5: The use of control messages can result in deadlock. A cycle is formed in (a) when two nodes transmit data messages to each other at the same time. Deadlock can be prevented by buffering control messages (b) when the outgoing link is not available.

One means of deterring this form of deadlock is to provide buffering within the cycle. As illustrated in Figure 4.5(b), GRIM employs a special queue for buffering control messages that cannot be injected into the network due to a busy outgoing link. When an incoming data message

is processed by a NI, an ACK or NACK message is inserted into the control message queue if the outgoing link is busy or the control message queue is already populated. The NI's firmware is designed to transmit buffered control messages as soon as the outgoing link becomes available. This method of preventing deadlock relies on the consumption assumption [63] that is basis of most deadlock prevention schemes.

4.2.4 Observed Advantages to NI-based Flow Control

One of the arguments for employing flow control in the NIs is that buffer space can be used dynamically as needed by applications. In this scheme a receiving NI rejects a message only if the intended endpoint lacks buffer space for accepting the message. In comparison, endpoint-based flow-control schemes generally reserve buffer space across the entire communication path before a message can be transmitted. This reservation results in injection policing that can limit performance. As a means of investigating the effects of injection policing, a benchmark program was constructed for GRIM to simulate an endpoint-based flow-control scheme. In this test, an endpoint transmits a large number of messages to another endpoint, which in turn transmits all the messages back to the sender as soon as they are received. In order to observe policing effects, the sending NI is limited to having no more than a fixed number of outstanding messages in-flight to the destination at any time. The benchmark measures the amount of time required for the sender to inject a burst of null-length messages. This value is divided by the burst size to determine the average injection overhead for a single message in the burst transfer. The benchmark is run multiple times, varying the burst size and maximum number of outstanding messages the sending endpoint is allowed to have at any time.

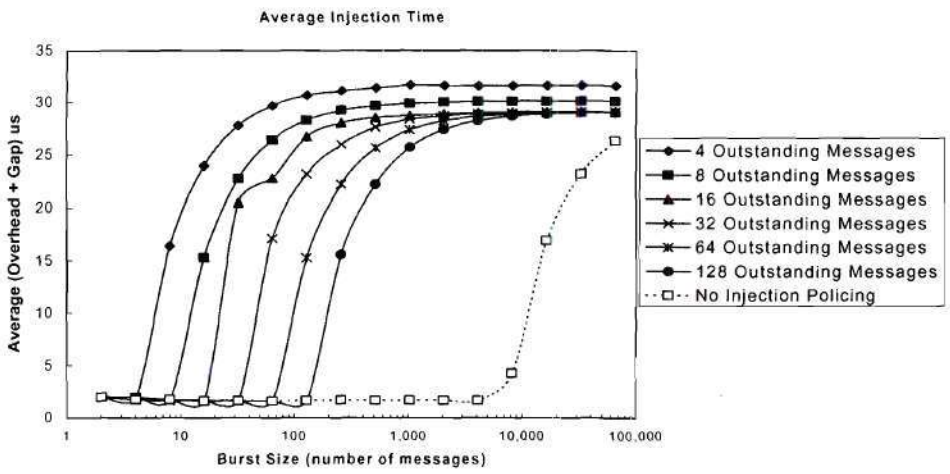


Figure 4.6: Injection policing effects of a credit-based flow control scheme implemented on top of the optimistic NI-based scheme used in GRIM. Performance is measured as the average message injection overhead time for null length messages.

The results of this experiment are presented in Figure 4.6 for hosts that allow from four to an unbounded number of outstanding messages. As expected the average message injection overhead for each test remains low until the injection burst size exceeds the number of outstanding messages the sender is allowed to have. After this point injection overhead rapidly increases to a steady-state value. While sharp, this increase is not instantaneous because the receiver injects credit-replenishing replies at the same time the sender injects outgoing messages. As these tests demonstrate, increasing an endpoint's maximum number of outstanding messages allows the endpoint to inject larger bursts without overhead penalties. For the case where no injection policing is performed, injection overhead does not increase until burst size is larger than 5,000 null-length messages. At this point the NI buffers become saturated and the host must wait for space to become available in the NIs. As these tests demonstrate, a NI-based flow control scheme allows buffer space in the system to be utilized in a dynamic manner. This trait allows endpoints to inject a large number of messages with a minimal amount of overhead for each message.

4.3 Logical Channels

In resource-rich cluster computers, the message layer must be designed to allow multiple communication endpoints in a host to interact with the network through a single NI. In GRIM this task is performed through the use of NI-based logical channels. A logical channel in this context refers to a set of data structures housed in NI memory for facilitating message transfers between an individual endpoint and the NI. The NI is equipped with a logical channel for each endpoint in the host, and therefore the NI must coherently transfer data between its collection of logical channels and the physical network at runtime. The advantage of this approach is that each endpoint is provided with its own virtual communication interface for the network.

The use of multiple logical channels in the NI has had a significant impact on the design of GRIM's NI firmware. One of the more challenging tasks in this effort has been adapting the reliable transmission protocol used by pairs of NIs to operate with multiple logical channels. The approach taken in GRIM is to structure the reliable delivery mechanisms to operate at the logical channel level as opposed to simply the NI level. This approach can help prevent head-of-line blocking that impedes communication performance. Another area of GRIM's firmware that was influenced by the use of multiple logical channels is the manner in which in-flight messages are buffered by NIs. Because there is a limited amount of memory available for implementing logical channels, modern versions of GRIM employ NI-level message buffering only at the sending NI. Finally, using multiple logical channels results in an increased workload for the NI. Therefore, the GRIM firmware was examined to determine which areas are affected the most by the use of logical channels. Performance tests were constructed to determine the maximum number of logical channels a NI could support under practical conditions.

4.3.1 Logical Channel Structure

A logical channel provides a virtual communication interface that an endpoint can use to interact with the network. Each logical channel is equipped with data structures in NI memory that are necessary for maintaining this interface. Figure 4.7 depicts the data structures employed for a logical channel in the GRIM communication library.

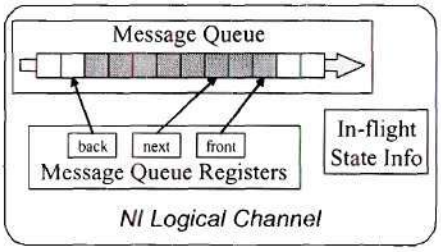


Figure 4.7: Each logical channel contains data structures necessary for providing a virtual communication interface.

In GRIM the three components of a NI-based logical channel are as follows.

- **Message Queue**: Each logical channel provides a dedicated amount of buffer space known as the message queue for housing in-transit messages. An endpoint supplies the NI with a new data message by appending the queue with the message and notifying the NI of the update. Multiple queue styles have been employed in GRIM and are discussed in Section 4.2.2.
- **Message Queue Registers**: A logical channel employs three registers for managing the capacity of its message queue. The first two of these registers are the front and back pointers, which indicate the region of the queue that is currently occupied by in-flight messages. A third register provides a next pointer for the NI. This pointer indicates the next message that is scheduled for transmission in the message queue.
- **In-flight State Information**: In addition to queuing data structures, the NI must also maintain state information for each logical channel. This state includes sequencing

information used by the NI's reliable transmission protocol, as well as information about the endpoint that owns the logical channel.

A NI's logical channels are configured by the host when the system is initialized. Endpoints are connected to logical channels based on configuration information supplied by users. In the current implementation the same logical channel provides both incoming and outgoing interfaces for an endpoint. While it is possible to use different logical channels to manage an endpoint's incoming and outgoing network interactions, doing so complicates the interface for the end user, and is therefore avoided.

4.3.2 Message Sequencing with Multiple Logical Channels

The use of multiple logical channels affects the manner in which a reliable transmission protocol is implemented in the NI. The primary issue involves the manner in which pairs of NIs are synchronized to provide in-order delivery of messages from different logical channels. In the simplest approach logical channel information is ignored in the reliable transmission process. In this approach, all logical channels are mapped on to a synchronous connection that exists between a pair of NIs. As described in Section 4.2.1, sequencing information is stored in two one-dimensional arrays: one for labeling outgoing messages and the other for verifying the order of incoming messages. The (outgoing/incoming) sequencing arrays are indexed by the value of the (destination/source) NI for the transmission. Sequence values in the arrays are incremented after every successful transmission. The downside of this approach is that congestion for one logical channel affects the performance of other logical channels. Since there is no way to distinguish which logical channel is congested at the destination, a request to retransmit a message forces the sending NI to perform rollback on all of its outgoing logical channels. This approach is undesirable, especially in resource-rich clusters where a host may have multiple endpoints that service incoming messages at different rates.

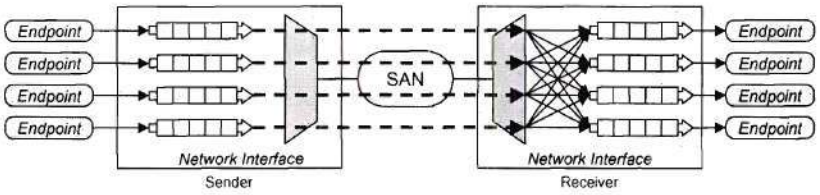


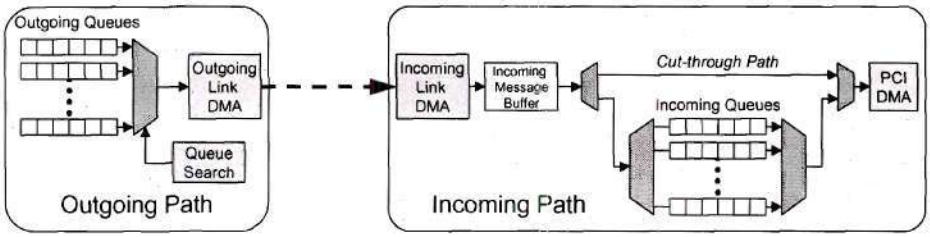
Figure 4.8: In the any-to-any approach, message sequencing is performed on messages based on the sending and receiving logical channels.

Another approach to implementing a reliable transmission protocol is to perform message sequencing on a *per logical channel* basis as opposed to a *per-NI* basis. In this approach, each NI manages two three-dimensional arrays of sequencing values. These (outgoing/incoming) sequence arrays are indexed by the values of the source logical channel, the (destination/source) NI, and the destination logical channel. As depicted in Figure 4.8 this approach allows an any-to-any form of communication between sender and receiver logical channels. In this approach, a message is retransmitted only if previous messages to the same logical channel were refused. The downside of this approach is that fetching sequence information is more time consuming and logical channels must maintain more state information. This approach is utilized in the current version of GRIM.

4.3.3 Distribution of NI Message Queues

Using multiple logical channels in the NI also affects the manner in which in-transit messages are buffered during the NI-NI communication process. In the ideal case it is desirable to provide buffering at both the sending and receiving NIs. Sending buffers allow the communication library to hide network congestion from the injecting endpoint. Likewise, a large receiving buffer for a NI can prevent the communication library from having to retransmit a message when a receiving endpoint is saturated. However, the issue with using multiple NI logical channels is that there is a finite amount of memory in the NI for housing the logical channels. Naturally, as the number of logical channels in the NI increases, the amount of buffer

space provided to each logical channel decreases. Therefore, it is necessary to consider how message buffering is performed in the NI in order to efficiently allocate the NI's buffer space.



(a) (b)

Figure 4.9: It is possible to buffer in-flight messages at both the (a) sending NI and the (b) receiving NI. A cut-though path at the receiver improves the performance of the receiving process.

In initial versions of GRIM, message buffering was provided at both the sending NI (outgoing queues) and receiving NI (incoming queues) as depicted in Figure 4.9. The intention of this approach is to split the NI's buffer space evenly between the sending and receiving tasks. Unfortunately, there are at least three drawbacks to this approach. First, it was observed that the incoming message queues were used infrequently in the communication path. The characteristic can be attributed to the fact that the endpoints in the system commonly feature enough buffer space and processing power to match the rate at which messages arrived from the network. Second, using incoming message queues adds to the workload of the NI. While a cut-through path allows messages to bypass an empty incoming message queue, the NI must still examine the incoming message queue when processing newly arrived messages in order to maintain ordered delivery. Finally, the allocation of incoming message queues decreases the amount of space available for outgoing message queues. This trait decreases the number of messages an endpoint can inject into the communication library at a time.

Based on these issues, GRIM was redesigned in a manner that only provides NI-level message buffering at the sending NI. In this approach, a message that cannot be accepted by an endpoint due to a lack of endpoint buffer space is simply dropped by the NI and negatively

acknowledged. Upon receipt of the NACK the sending NI automatically performs rollback on the appropriate outgoing message queue and retransmits the message at a later time. While this approach increases network load, Myrinet provides a considerable amount of bandwidth and retransmissions take pace only when a receiving endpoint is saturated. In addition to increasing the amount of buffer space available for housing outgoing messages, this approach reduces the overhead in receiving portion of the NI's firmware.

4.3.4 Number of NI Logical Channels

Utilizing multiple logical channels in the NI naturally results in an increased workload for the NI. Therefore, it is beneficial to examine how the use of logical channels impacts the performance of the NI. A first step in this process is determining which portions of NI firmware are most affected by the use of logical channels. In GRIM's firmware the use of multiple logical channels has more of an impact on sending tasks than receiving tasks. In the sending portion of the NI's firmware the NI must inspect each outgoing logical channel to locate newly injected messages. Increasing the number of logical channels therefore increases the amount of time that a NI must spend searching for new messages to send. In contrast, the receiving process is not significantly affected by the use of multiple logical channels. This is because an incoming message contains all the information necessary for the NI to determine which incoming logical channel should be used to accept the message.

A second step in examining the impact of logical channels on NI performance is determining the maximum number of logical channels a NI can support under practical conditions. Given that the sending tasks of the NI are the region of NI firmware that is most affected, an experiment was constructed to determine how increasing the number of logical channels in the NI impedes performance. Specifically, this experiment is designed to measure the amount of time required for a NI to scan all of its outgoing logical channels for new messages. This scanning time is important because it can add delay to the total transmission time of an

individual message. For example, a NI with 16 logical channels may have to scan 15 empty logical channels before it detects a message that was available all along in the last logical channel. Scanning time is also important because it consumes NI CPU cycles that could have been used to perform other NI tasks.

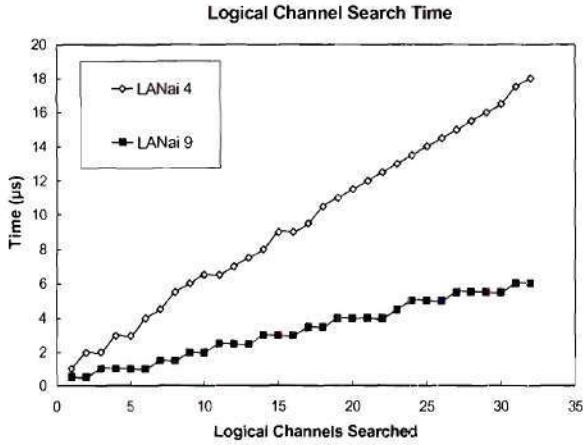


Figure 4.10: The amount of time required by the NI to search a fixed number of message queues for new messages.

The results of the outgoing logical channel search experiment are presented in Figure 4.10. The test was performed using the LANai 4 and 9 versions of the Myrinet NI card. As expected the LANai 9's performance is roughly three times better than the LANai 4 due to architectural enhancements of the card. When using a single logical channel, the tests revealed that the LANai 4 and 9 cards require 1 µs and 0.5 µs respectively for the NI to examine the logical channel. These times increase to 5.5 µs and 1.5 µs for 8 logical channels, 9 µs and 3 µs for 16 logical channels, and 18 µs and 6 µs for 32 logical channels. These results can be used to set a practical limit on the number of logical channels employed in a NI. Given that most Myrinet message layers provide end-to-end latencies of approximately 10-20 µs, a conservative approach would dictate that in the worst case, a NI would spend no more than half of the potential delivery time searching for messages in the outgoing logical channels. Therefore, it is suggested that the LANai 4 and 9 NI cards use no more than 8 and 24 logical channels respectively.

4.4 Active Message Interface

The first of two APIs used in GRIM provides end users with an active message style programming interface. This interface is designed to be more robust than the original AM specification [52] due to the heterogeneity of communication endpoints used in resource-rich clusters. In GRIM, active message function handlers must be registered with a global server before they can be used by applications. In this process endpoints submit a string identifier for each function handler and are returned a unique integer identifier that any endpoint can use to reference the function. An active message contains all arguments necessary for an endpoint to process a message. Therefore, GRIM endpoints employ a polling interface for detecting and processing incoming messages. The active message interface was extended with an overflow buffer to break the dependency between outgoing messages and incoming messages, thus avoiding deadlock.

4.4.1 Active Message Handler Management

GRIM provides an infrastructure for dynamically managing active message function handlers used in the cluster. The challenge in this task is providing a means for endpoints to publish a list of available function handlers to the global context and have these handlers universally identified in a coherent manner. Since handler registration generally only takes place during system initialization, GRIM dedicates a single node in the cluster for providing global handler registration. In this approach, endpoints submit a list of string identifiers for available function handlers to the server. The server correlates submissions and assigns unique integer identifiers for each string identifier. The list of mappings between string and integer identifiers is then published to all nodes in the cluster.

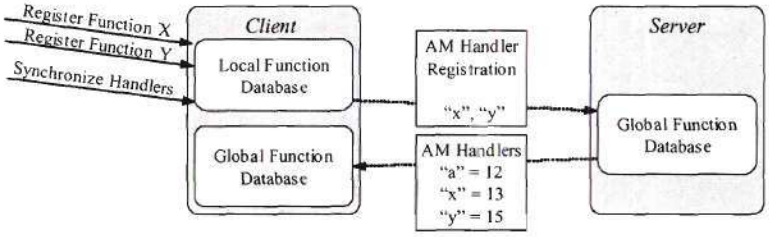


Figure 4.11: The active message API requires endpoints to register function handlers locally and then publish the information to a global database.

Figure 4.11 depicts an example of the active message handler registration process used in GRIM. The first step in this process is for an endpoint to locally identify its available function handlers using a local registration function. When the endpoint is prepared for other endpoints in the cluster to use the handlers, it executes a synchronization function. This function transmits the endpoint's table of available function handlers to the server in the cluster that manages the global database of active message handlers. The server processes the message by assigning new global identifiers for function handlers that have not yet been registered. The entire list of global function handlers is then transmitted back to the sender, where the data is stored in a database of global function handlers. At run time, the endpoint consults this database to determine an integer identifier for a named function handler. If a node cannot perform the translation locally, it issues a request to the server to determine the proper integer identifier.

4.4.2 Polling Interface

Message-passing programming interfaces specify the means by which applications consume incoming messages. In active messages, the incoming message is labeled with all of the information necessary for the receiver to process the message, regardless of the state of the application. Therefore, active message libraries typically do not implement explicit receive functions to extract specific messages from the network. Instead, these libraries usually provide a poll function. This function extracts incoming active messages that are queued at an endpoint, and

executes the appropriate active message function handlers. GRIM provides a polling function to perform these operations, which must be invoked periodically by applications to guarantee that incoming messages are processed. For threaded programs, GRIM provides a built-in thread that periodically invokes polling operations.

4.4.3 Deadlock Avoidance for Message Handlers

A common problem in implementing an active message based system is that it is possible for deadlock to occur at the application level if precautions are not taken. Deadlock can occur because an active message handler can inject a reply message back into the network. The buffer space housing the incoming message cannot be freed until the handler completes and a handler that issues a reply cannot complete until buffer space is available in the network to inject the reply. As illustrated in Figure 4.12(a-b), this can result in a cyclic dependency between two applications when the network is congested. One option for removing this dependency is to utilize separate buffer space or separate networks for send and reply messages, and specify that a reply message cannot generate additional replies. This option is costly in terms of buffer space and limits the functionality of end applications.

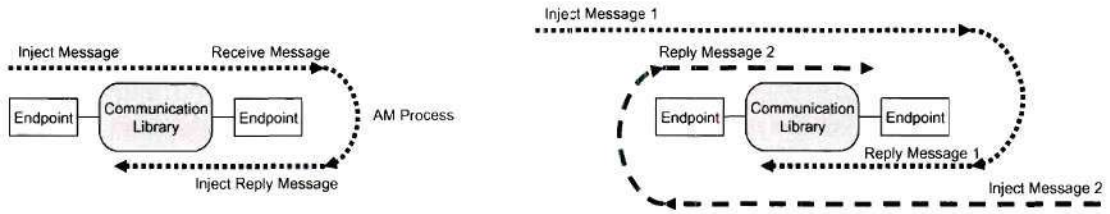


Figure 4.12: Example of deadlock at the application level. (a) The dataflow of messages for an active message handler that injects a reply message. (b) Application deadlock due to the simultaneous injection of two messages that require replies.

A second option that is utilized in GRIM is to provide an overflow message queue at the host. If a handler must inject data back into the network and there is no room available for the

outgoing message in the NI, buffer space is allocated in host memory to house the message. This memory serves as an overflow message queue, with additional injections appended to the queue until all overflow messages can be injected into the network. Since the host has a finite amount of memory, this approach does not guarantee that application deadlock due to message recycling will never occur. However with the large amount of virtual memory available to a host, this approach makes deadlock extremely unlikely and comes at little penalty to the common case.

4.5 Remote Memory Interface

The second programming interface provided by GRIM is for remote memory operations. This interface is designed to provide low-level mechanisms for manipulating and observing memory distributed throughout the cluster. For management purposes, GRIM requires that all remote memory operations utilize *registered memory*. Registered memory is memory allocated by GRIM that is guaranteed to always be available in physical memory. The remote memory interface utilizes virtual memory addresses to reference registered memory, and therefore requires mechanisms to translate a virtual address into a physical address that the NI can use to complete a remote memory operation. GRIM allows both remote memory reads and writes, and provides a simple notification mechanism to indicate that a transaction has completed. As a means of improving performance, GRIM also provides a special remote memory write operation that uses a physical address to reference registered memory. While there are basic rules that a user must follow when using the remote memory interface, the API provides a powerful means of managing distributed data in the cluster.

4.5.1 Registered Memory

The first step in using the remote memory API is obtaining an allocation of registered memory. Registered memory refers to a block of memory that has been allocated by the communication library and pinned so that the host's operating system does not attempt to relocate

the block's physical pages. GRIM provides two interfaces for obtaining registered memory. The first interface obtains a block of memory in user space and then utilizes a system call to pin all of the pages of the allocation. While this approach can acquire large blocks of memory, its primary drawback is that the allocated memory is non-contiguous in the physical address space. This characteristic can result in reduced performance for NI interactions, because when the NI accesses the memory, it must fragment its DMA operations into a series of page-sized transfers. Additionally, applications using this option must be given sufficient access privileges for invoking the system call that pins the memory.

GRIM provides a second interface for obtaining registered memory that uses a specially designed pinned memory management unit. This unit is a combination of both user- and kernel-level software, and provides mechanisms for allocating large blocks of pinned memory that are contiguous in the physical address space. While the operating system has a limited amount of contiguous memory, it is possible for this library to obtain multiple 4 MB regions from a host with 256 MB of system memory. There are two advantages to using this interface for obtaining registered memory. First, since this memory is contiguous, the NI can execute remote memory operations with a single DMA transfer. This feature decreases overhead and improves performance. Second, applications do not have to be given system privileges in this approach because a dedicated device driver performs the privileged task of interacting with the kernel's memory system.

4.5.2 Virtual Memory Translation in the NI

Remote memory operations are designed to use virtual memory addresses to reference a block of registered memory. Because the NI operates with physical memory addresses it is necessary for the NI to be equipped with mechanisms for translating a virtual address to a physical address. In GRIM the Myrinet device driver is designed to perform address translation for the NI on demand. In this process the NI DMAs an address translation request to a known

location in the kernel's memory space and then triggers an interrupt signal to obtain the host's attention. The Myrinet driver handles the interrupt with an interrupt service routine that examines the request, performs the translation, and writes the resulting information back to the NI. Once equipped with a translation, the NI can process a remote memory operation.

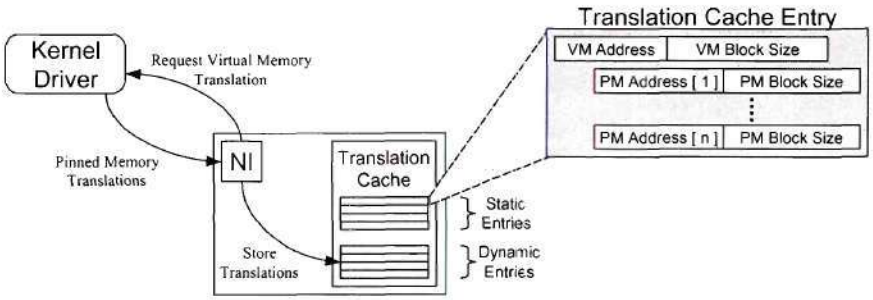


Figure 4.13: If a virtual memory translation is not available in the NI's translation cache, the Kernel must be consulted. An entry in the translation cache contains the size of a virtual memory block and a list of its physical memory regions.

Interrupt service routines are expensive operations for both the host CPU and the NI. Therefore, it is beneficial if the NI is equipped with a means of caching address translations. As Figure 4.13 illustrates, GRIM's NI firmware is designed with a translation cache that is divided into two regions. The first region houses static translation entries that are programmed by the communication library when an application acquires a large block of contiguous, registered memory. The second region of cache entries is for storing translations performed at runtime by the NI that were not satisfied by the first set of cache entries. This part of the cache is beneficial in situations where the first set of cache entries is full, or when an application frequently accesses the same virtual address. Cache entries contain the virtual memory address for the registered block of memory, the size of the block, and the physical addresses and sizes of the pages that are used for housing the block of memory.

4.5.3 Remote Memory Writes (RM-V, RM-P)

GRIM provides two functions for performing remote memory writes. The first of these functions utilizes a virtual memory address to reference a remote node's registered memory and is referred to as an RM-V operation. At runtime the receiving NI must examine the arguments of the remote memory write operation, translate the virtual memory address, and perform the necessary DMAs to store the message's payload in the physical pages of the registered memory. Referencing the block of remote memory with a virtual address simplifies the interface for end users because local and remote endpoints can use the same memory pointer to reference a block of memory.

GRIM provides a second form of remote memory write referred to as an RM-P operation. An RM-P operation utilizes a physical address to reference a block of registered memory instead of a virtual address. RM-P operations are designed for experienced users that need to perform custom data transfers that must take place efficiently. Since RM-P operations reference the destination's memory with a physical address, the receiving NI does not have to perform virtual memory translation to execute the message. Therefore, RM-P messages are expected to have better performance than RM-V messages. GRIM provides a set of mechanisms for an end application to translate the virtual address of a local block of registered memory into a physical address. RM-P operations can also be utilized as an efficient means of updating the memory of a remote peripheral device.

4.5.4 Remote Memory Reads (RM-RV)

The second type of remote memory operation allows an endpoint to fetch data from another endpoint's address space. Remote memory reads operate using virtual addresses to reference registered memory at the sending and receiving endpoints, and are referred to as RM-RV operations. RM-RV messages are utilized in GRIM as follows. First the sending endpoint injects an RM-RV message that contains references to (i) the receiver's memory that is to be

fetched, (ii) the sender's memory where the results are to be stored, and (iii) the length of the transfer. When the message arrives at the receiving NI, address translation is performed and the requested data is fetched into a NI buffer. This data is then transmitted back to the sending NI in the form of an RM-RV reply message. Upon receiving this reply message, the original NI translates the virtual address specified in the message, DMAs the message's payload to the address, and marks the original RM-RV message as acknowledged in its outgoing message queue.

4.5.5 Endpoint Notification for Remote Memory Operations

When utilizing a remote memory interface, a common operation is to transfer a block of data and then update a memory location in either the sending or receiving endpoint's address space to notify the endpoint that the transfer has completed. This notification operation can easily be performed using two remote memory operations, the first performing the transfer and the second performing the update. However, processing two instructions increases the workload of the communication library, which can degrade performance. Therefore, remote memory operations in GRIM are equipped with signaling mechanisms that allow an endpoint to be notified when a remote memory operation completes. In these mechanisms, users can specify the virtual address of a single 32-bit word in registered memory to update when the remote memory operation completes. For remote memory writes, the user can specify both the location of the variable to update in the receiver's address space, as well as a 32-bit value to write to the variable. Remote memory reads allow users to specify the virtual address of a variable at the sender that is cleared when all data is fetched.

4.5.6 Mixing Active Message and Remote Memory Operations

GRIM is designed to allow users to work with the active message and remote memory programming interfaces at the same time. This feature is possible because both interfaces are

implemented as independent units that are layered on top of a system that reliably transfers messages between endpoints in the cluster. Because the delivery system forces each programming interface to adhere to a common message format, it is possible to mix traffic from different interfaces in the delivery system. The outgoing messages of different programming interfaces are merged when the endpoint injects the messages into the NI. For messages arriving from the network, the NI delivers the messages to the proper programming interface based on the Type field of the messages. However, the active message and remote memory programming interfaces have different strategies for buffering and processing messages arriving at the NI. Therefore, it is necessary to specify the order in which incoming messages are processed when the two programming interfaces are used at the same time.

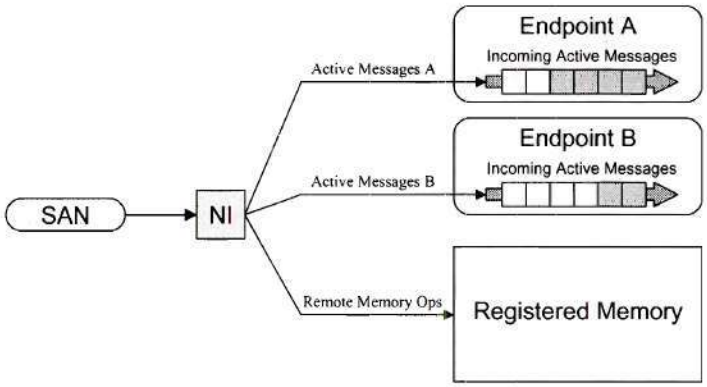


Figure 4.14: The data path for active messages provides extra message buffering before messages are processed compared to the remote memory data path.

A message layer that implements in-order delivery between a pair of endpoints guarantees that messages are processed by the receiver in the same order that that they were injected by the sender. Unfortunately, differences in the manner that messages are buffered in the active message and remote memory programming interfaces make this guarantee undesirable when the two interfaces are used at the same time. As Figure 4.14 illustrates, the issue is that while remote memory messages are processed by the NI as soon as they arrive, active messages are placed in an additional endpoint-level buffer before they are processed. Therefore, in order to

prevent remote memory messages from bypassing active messages, a strictly ordered system would require the NI to delay executing a remote memory message until the endpoint's active message queue is empty. This requirement impedes performance and negates the benefits of using the NI to process remote memory operations.

An alternative approach is to relax the requirement that the two programming interfaces are tightly synchronized in terms of processing order. In this approach, messages are processed in the order in which they arrive at a programming interface, not the NI. An examination of the communication paths of GRIM reveals that this approach only violates out-of-order execution in one case: when an active message is followed by a remote memory operation. Because of the buffering of active messages in the endpoint, this approach can result in a remote memory operation being executed before preceding active messages are completed. However, all other uses of the communication library are guaranteed to take place in the order in which they are injected (AM followed by AM, RM followed by AM, and RM followed by RM). This approach is implemented in GRIM and requires users to be aware that remote memory operations may bypass previously injected active messages.

4.6 Summary

GRIM is a communication library designed for clusters that feature a high-performance Myrinet SAN. One of the key characteristics of GRIM is that core functionality is largely pushed into the NI cards. A NI-based reliable transmission protocol allows the NI to dynamically manage the transfer of data between NIs, and relies on an optimistic approach in order to decrease latency. Each NI is equipped with multiple logical channels in order to provide the various endpoints in the cluster with private communication interfaces to the network. The presence of multiple logical channels in the NI has resulted in changes in the way messages are buffered in the communication pipeline, because each NI has limited a limited amount of on-card memory. GRIM simultaneously supports both active message and remote memory programming interfaces.

These interfaces provide powerful programming abstractions that can be utilized in a flexible manner. This description of GRIM represents the core functionality of the communication library upon which extensions for resource-rich clusters are built upon.

CHAPTER V

HOST-TO-HOST TRANSFERS

Modern message layers for cluster computers are optimized to provide high-bandwidth, low-latency communication between host CPUs in the cluster. While the key design goal of GRIM is flexible communication among host CPUs and peripheral devices, it is also desirable if GRIM is capable of providing reasonable levels of performance for host-to-host interactions. In order to achieve such performance GRIM had to be designed with communication mechanisms that operate in a streamlined manner. The active message and remote memory programming interfaces used in GRIM both rely on the same mechanisms for transferring data between a pair of hosts. These mechanisms use a communication pipeline that is comprised of three sets of data transfers: host-to-host, NI-to-NI, and NI-to-host. By optimizing each of these transfers it is possible to improve the overall performance of GRIM. This performance is further enhanced by end-to-end optimizations that increase the amount of overlap that takes place between the stages in the communication pipeline.

This chapter examines the low-level performance characteristics of GRIM for data transfers between host CPUs. This analysis takes place in two parts. First, the three stages of data transfer in the communication pipeline are examined individually. For each pipeline stage, data transfer characteristics are reported as well as measurements of the amount of overhead that is required by GRIM to perform stage-specific operations. Second, GRIM's performance is examined in the context of end-to-end transfers. In this effort, the effects of pipeline and cut-through optimizations are inspected. End-to-end performance measurements are reported for two sets of hosts and two types of Myrinet NI card. These results are compared to the performance values of other message layers, and reveal that GRIM provides competitive performance levels

for interactions between host CPUs. These measurements also indicate that GRIM is designed in a manner that allows the overhead of its sophisticated functionality to be hidden from the critical path.

5.1 Overview of the Host-to-Host Communication Path

In traditional cluster computers the main goal of the communication library is to rapidly transfer data from one host CPU to another. Since host CPUs are the only resource available for processing an application in these clusters, existing communication libraries have largely been optimized for high-performance host-to-host interactions. In resource-rich clusters the fundamental goal of the communication library is flexible communication, as the cluster provides diverse resources to assist in the processing of an application. However, it is still important that a message layer for resource-rich cluster computers be able to obtain reasonable levels of performance for host-to-host interactions, as host CPUs are expected to provide significant contributions to application processing in these clusters. Therefore, a key part of examining GRIM is evaluating the communication path it provides between two host CPUs.

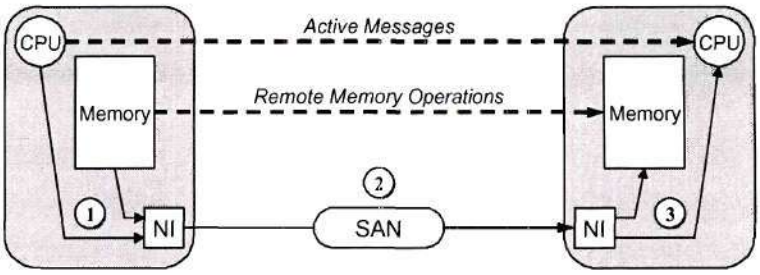


Figure 5.1: The active message and remote memory programming interfaces share the same communication path. The three phases of data transfer include (1) Host injection, (2) NI-NI delivery, and (3) NI ejection.

GRIM offers two different programming interfaces that can be utilized for host-to-host communication: active message and remote memory operations. As Figure 5.1 illustrates both of

these programming interfaces utilize the same communication path between hosts. Data transfer in this path can be divided into three separate phases:

- **Host injection into the NI (Host-NI)**: Both the active message and remote memory interfaces begin the communication process by injecting a new message into the NI. This operation takes place over the local PCI bus with transfers orchestrated by the host CPU.
- **SAN Transfer (NI-NI)**: Data is then transferred across the SAN using reliable transmission mechanisms implemented in the NIs.
- **NI Ejection (NI-Host)**: After receiving a valid message the NI must transfer the message to the appropriate location. For active messages the NI appends an incoming message to the host's message queue. Remote memory operations are performed by the NI, where data is directly transferred to and from the host endpoint's address space.

Implementation details are provided in this chapter for each of the data transfer stages.

5.1.1 Evaluation Environment

Three different clusters were utilized in the performance benchmarks provided in this chapter. The first cluster utilizes hosts that have four 200-MHz Pentium Pro (PPro) processors and 32b/33MHz PCI. Due to the limited performance of these systems, the PPro hosts were only utilized to provide a point of comparison for PCI measurements. The second cluster used in this effort is based on hosts that have a single 550-MHz Pentium III (P3) processor and a 32b/33MHz PCI bus. These systems provide reasonable levels of performance and are characteristic of middle-of-the-road clusters that are commonly utilized in academic research efforts. The final cluster used in these tests employs hosts that contain dual 1.7-GHz Pentium IV (P4) processors and feature both 32b/33MHz and 64b/66MHz PCI buses. While the P4 hosts provide the best performance for all the clusters used in these tests, the motherboard chipset (Intel 860) for these

hosts suffers from unusual PCI performance characteristics that are unsettling. Therefore, the majority of the benchmarks presented in this chapter were performed using the P3 cluster. Measurements using the P4 cluster are provided as part of the overall performance evaluation described in this chapter.

The measured values reported in this thesis are the results of benchmarking software that was constructed to provide practical and repeatable estimates of performance. In all of these tests a measurement is performed several times. The median value for all iterations of a test is reported as the measured value. When applicable the benchmarking programs used in this work employ cache polluting mechanisms between successive iterations of a measurement. The mechanisms help to obtain a better measurement of performance under worst case operating conditions. Finally, it is important to note that benchmarks are performed on unloaded systems in order to obtain fair evaluation environment.

A common form of benchmarking network performance that is used in this chapter is to acquire a round-trip timing measurement for a transmission. In this form of measurement a message is transmitted from one entity to another and then returned to the original sender. The sender measures the amount of time the message is in-flight in the network and records this value as the round-trip time. Dividing the round-trip time in half yields an estimate of the one-way transmission time for the message. One-way transmission times are commonly referred to as the latency for communication. Dividing the amount of application data in a message by the one-way communication time provides an estimate of the bandwidth for the transmission.

5.2 Injecting Data into the Sending NI (Host-NI)

The first stage in the host-to-host communication path is for the host endpoint to inject a message into the local NI. This task is the same for active messages and remote memory operations because the sending host endpoint assembles an outgoing message in host memory and then transfers it to the NI's memory. There is no simple means of efficiently transferring data

from a host-level application to a peripheral device such as the NI. Since the host CPU lacks a DMA engine of its own, it must either use the NI card's PCI DMA engine to perform the transfer or spend CPU cycles moving the data itself with programmed I/O (PIO) operations. While DMA operations rapidly transfer large blocks of data, configuring the DMA engines is a complex and an expensive operation. PIO mechanisms on the other hand are simple to implement but offer limited performance. Therefore, researchers typically employ multiple mechanisms for injecting data from the host CPU into a NI and select the best mechanism for a transfer based on run-time conditions.

Since resource-rich clusters utilize a number of diverse peripheral devices, it is valuable to encapsulate the various host-to-card transfer mechanisms into a single portable library that can be used with different cluster resources. From a user's perspective it is beneficial if the library employs self-tuning mechanisms that allow the application to examine the host's hardware environment and automatically determine the most efficient means of injecting data into a peripheral device. Such a library has been constructed for GRIM named TPIL: the tunable PCI injection library. This library selects from multiple CPU-specific PIO and card-specific DMA transfer mechanisms to maximize the performance obtained for a given injection size. While this section specifically focuses on the use of TPIL to increase host-to-NI injection times, the library is utilized with other cards such as the Celoxica card discussed in the following chapter.

5.2.1 Programmed I/O Transfer Mechanisms

The first method by which data can be transferred from a host application to a peripheral device is through programmed I/O (PIO) operations. In this approach, the device driver for a peripheral device provides an application with a memory map of the peripheral device's on-card memory. This memory appears as virtual memory that the application can read or write. Interactions with the memory are translated by the host CPU and I/O chipsets into individual PCI read or write transactions. PIO operations incur a large amount of overhead (roughly 1 us for

reads, 2 us for writes). In addition to providing a simple means of interacting with a device, PIO operations allow the host CPU to perform virtual memory translations for the data residing in host memory automatically through normal virtual memory operations.

A number of features of the modern x86 architecture can be exploited to accelerate PIO performance [64]. These hardware features include:

- **Write-Combining**: The write-combining MTRR registers included in Pentium MMX and higher processors allow stores to user-specified memory ranges to take place without strict ordering. This allows multiple writes to consecutive memory addresses to be combined for burst transfers.
- **MMX Registers**: The eight 64-bit MMX registers can be used as a temporary buffer for moving 64-byte blocks of data between host memory and the I/O system. This technique allows data writes to take place as burst operations that are efficiently mapped by the chipset into PCI transactions.
- **SSE Cache Control**: The streaming SIMD extensions (SSE) [65] unit adds features to provide user-level control of the CPU's cache. In addition to pre-fetching operations, the SSE hardware provides non-temporal stores where writes can bypass cache memory and be flushed directly to memory.

Previous literature [66] has discussed the use of write-combining to improve the host-to-card performance for transfers less than a kilobyte in size. While this greatly reduces the amount of time an application spends injecting data, there are pitfalls that must be addressed. The main hazard with write-combining is that writes can be reordered in the chipset to improve burst transfer performance. For NIs this could result in a race condition where an update to a queue pointer erroneously bypasses the actual placement of data in the queue. Such hazards must be prevented through careful definitions of memory regions that perform write-combining or by

using fence instructions made available in Pentium III processors. A second pitfall is that there are a limited number of regions that can be marked for write-combining, and that the definition of such operations is a privileged operation. While early versions of GRIM utilized write-combining, it has been abandoned in favor of the MMX and SSE PIO transfer techniques.

5.2.2 DMA Transfer Mechanisms

The second means of injecting data into a peripheral device is to utilize DMA transfers. In this approach, the host CPU configures a peripheral device's DMA engines to transfer data from host memory to card memory. DMA transfers are generally only useful for moving large blocks of data because there is a large amount of overhead involved in having the host orchestrate a DMA transaction. Part of this overhead is due to the fact that the host CPU must use PIO writes to configure the registers of a card's DMA engines for each transfer. Additional overhead can be attributed to the notification mechanisms employed by the DMA engines. After the host initiates a DMA transaction it must wait until the DMA completes before it can proceed. A DMA engine typically notifies the host that a transfer has completed by generating an interrupt, which must be handled by the card's device driver.

The main difficulty in utilizing DMA transfers is dealing with virtual memory. In the x86 architecture PCI devices operate with physical addresses while applications utilize virtual addresses. As a consequence three issues must be addressed when using a DMA operation: (i) virtual to physical address translation must be performed, (ii) data for a single DMA must be contiguous in the physical address space, and (iii) memory must not be changed (i.e. paged out) during a DMA transfer. Based on these issues designers typically employ one of two approaches to using DMAs. The first approach is simply to allocate a large block of pinned contiguous memory which serves as an intermediate buffer for data transferred with a DMA. While this approach simplifies DMA transfers it incurs the overhead of a data copy from application memory to the intermediate buffer. Another approach is to instead pin the user's virtual memory

and perform DMA operations on the individual page frames housing the data. This approach involves multiple DMA operations but generally provides the best performance. In the TPIL library three card-specific operations are provided for transferring memory with DMA engines:

- **One-Copy**: In this approach, user data is copied into a large (128 KB) contiguous buffer. The card then issues a single DMA to move the data. The operation is repeated if application data exceeds the capacity of the transfer buffer.
- **Double-Buffered One-Copy**: Like the previous approach data is copied from user space to a contiguous transfer buffer in host memory. However, this approach splits the buffer in half and overlaps the transfer of data into the buffer with the DMA operation.
- **Zero-Copy**: This approach pins the pages holding user data and configures the DMA engines to transfer data directly from the user pages. While individual DMA transfers are limited to a page in size, this approach removes the need to copy data in host memory, thus greatly improving speed.

5.2.3 TPIL Host-to-NI Performance

TPIL is designed to operate with the GNU/Linux 2.4 operating system and is implemented as a combination of user, kernel, and device level software. The internal benchmarking functions of TPIL were utilized to examine the performance of two types of hosts using two different versions of the Myrinet NI card. In the first set of tests, the Myrinet cards were placed in a 550-MHz Pentium III system that only featured a 32b/33MHz PCI bus. The results of the benchmark are displayed in Figure 5.2. In these tests the PIO methods had the best performance for small to medium sized transfers (less than 10 KB) while large transfers were best served with zero-copy DMA operations. The MMX and SSE methods had similar performance levels until approximately 2 KB, at which point the SSE's cache manipulation operations began to positively affect performance. For the DMA operations the zero-copy method provided the

highest levels of performance for this system, with a maximum transfer rate of approximately 119 MB/s observed for both NI cards. These transfer rates are less than the maximum 132 MB/s performance levels of the 32b/33MHz PCI bus because the data transfers are sourced from virtual memory that is non-contiguous in the physical memory address space.

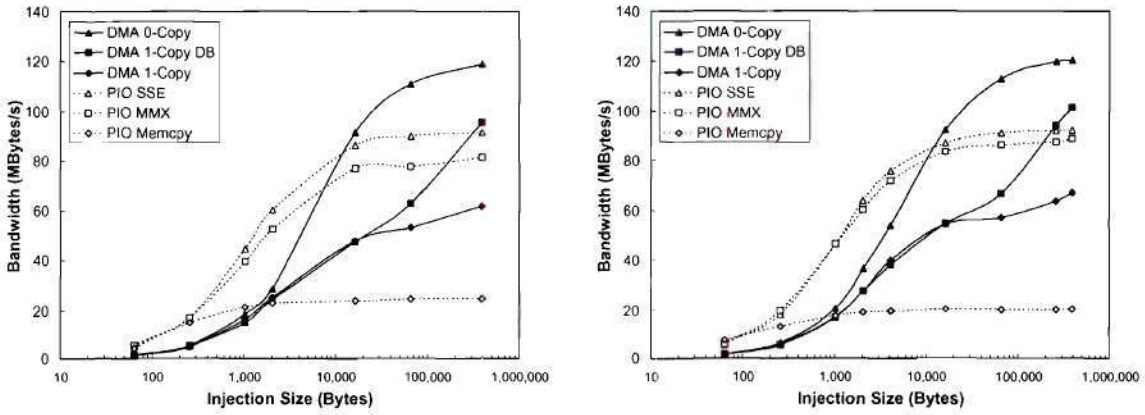


Figure 5.2: Host injection performance for a P3-550 MHz host using the (a) LANai 4 and (b) LANai 9 Myrinet NI cards.

In a second series of tests the benchmarks were repeated using the 1.7 GHz Pentium IV hosts. The results of these tests are presented in Figure 5.3 for the (a) LANai 4 and (b) LANai 9 NI cards. The LANai 4 card was placed in a 32b/33MHz PCI slot and obtained a maximum injection rate of 102 MB/s using SSE PIO transfers. The DMA engines for the LANai 4 card performed poorly in this host and allowed data injection rates of only 89 MB/s using zero-copy DMAs. The LANai 9 card was placed in a 64b/66MHz slot and obtained much better performance. While MMX and SSE PIO transfer mechanisms were limited to approximately 53 MB/s, the DMA operations were able to reach up to 213 MB/s. An interesting observation of this performance is that for very large transfers, the double-buffered one-copy DMA operation provides better performance than the zero-copy mechanism. This trait can be attributed to the fact

that the Pentium IV hosts have large amounts of host-memory bandwidth because the hosts utilize RDRAM for main memory. Therefore, for large transfers it is more efficient for the host to arrange source data so that DMAs take place in large contiguous transfers than it is for the host to schedule a large number of small transfers.

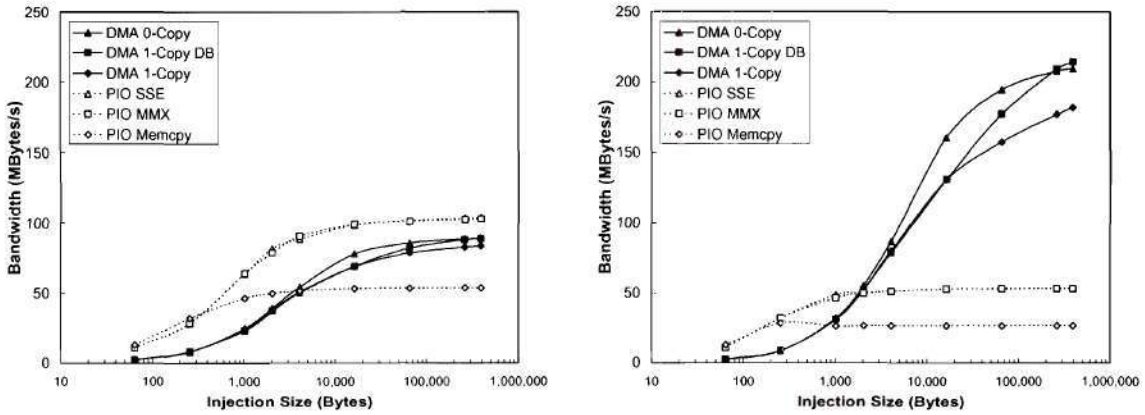


Figure 5.3: Host injection performance for a Pentium IV-1.7 GHz using (a) the LANai 4 (32b PCI) and (b) the LANai 9 (64b PCI) NI cards.

As a means of comparing PCI performance between systems, a LANai 9 card was placed in three different hosts and the injection performance of TPIL was measured. The best transfer rates obtained with TPIL are reported in Figure 5.4 for each host. The PPro-200 MHz system provided the worst performance in these tests because it has poor PCI performance and lacks MMX and SSE units. The P3-550 MHz system provided the most linear performance of all the systems. Linear performance is desirable in a message layer because the user can be assured that reasonable performance can be obtained regardless of the size of the messages that are transferred. Finally the P4-1.7 GHz system exhibited different performance characteristics for its 32b and 64b PCI buses. PIO operations work well for the 32b PCI bus but not for the 64b PCI bus. The converse can be said of DMA operations for this system. Based on the performance measurements it is advantageous to either (a) place the LANai 9 card on the 32b PCI bus if the message layer does not issue injections larger than 4 KB or (b) construct the message layer to use

transfer sizes that are greater than 8 KB when the LANai 9 card is placed on the 64b PCI bus. GRIM is optimized for the latter of these options because of the performance benefits of the 64b PCI bus.

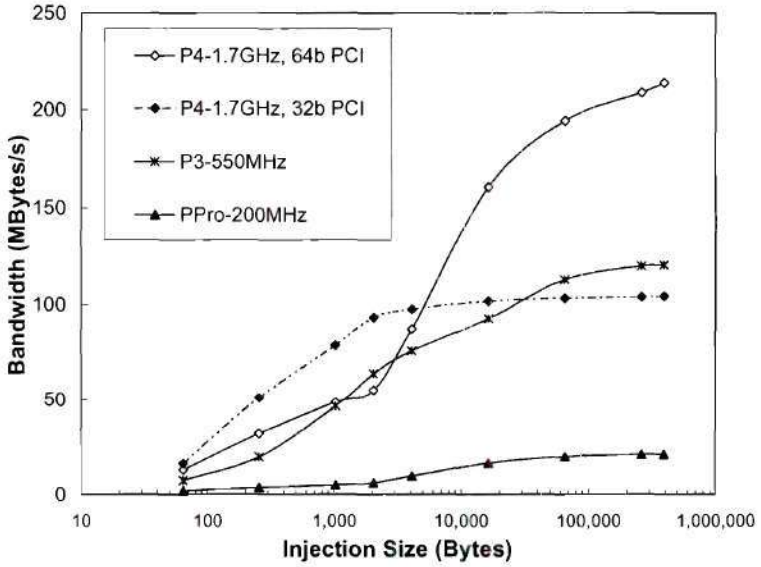


Figure 5.4: Overall TPIL performance for three different hosts.

5.3 Data Transfer between NI Pairs (NI-NI)

The second form of transmission in the host-to-host communication path is the transfer of data from the sending NI the receiving NI. This NI-NI transfer takes place across the Myrinet network and requires the use of a reliable transmission protocol in order to guarantee that messages are transferred in-order from one NI to another. As a first step in examining NI-NI performance, tests were constructed to observe the amount of time required to transfer various-sized messages between NI pairs. These measurements give an estimate of the native performance available in the SAN. Additional measurements were made of GRIM's NI firmware to determine how much overhead is added by GRIM's reliable transmission protocols. These measurements include timings of the individual operations that are performed by sending and receiving NIs during the reliable transmission process.

5.3.1 Native SAN Performance

A first step in measuring the performance of NI-NI transfers in the host-to-host communication path is determining the native performance of the SAN hardware. A benchmarking program was constructed to determine how much bandwidth could be obtained from the SAN under ideal conditions. In this test round-trip timing measurements are performed between a pair of NIs that are directly connected by a SAN cable. NIs detect a new message in this test only when the message has arrived in its entirety at the NI. The test is performed several times using different values for message size.

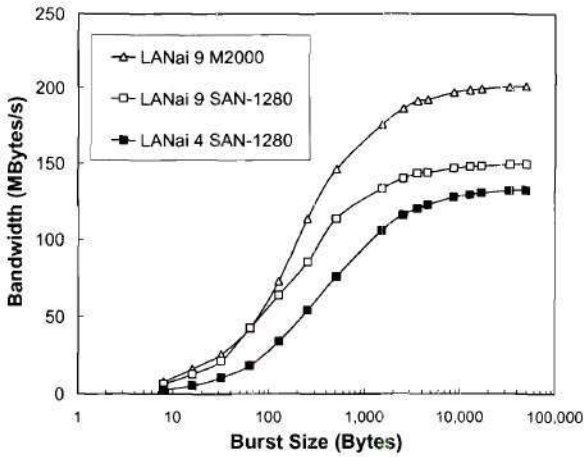


Figure 5.5: Observed bandwidth for different transfer sizes between NI pairs. Measurements are based on round-trip timings.

The results of the bandwidths measured in this test are presented in Figure 5.5 for three pairs of NI cards. The first two tests measured the performance of the SAN-1280 links [67] using pairs of LANai 4 and LANai 9 NI cards. The LANai 4 cards are able to obtain approximately 132 MB/s (1.056 Gb/s) while the faster LANai 9 cards reach close to 150 MB/s (1.2 Gb/s). These transfer rates suggest that NI pairs can obtain a significant portion of the SAN-1280's available 160 MB/s (1.28 Gb/s) bandwidth. In the third test a two LANai 9 cards are configured to use the Myrinet-2000 link standard. These cards are able to obtain 200 MB/s (1.6 Gb/s), which is 80% of

the Myrinet-2000 standard's 250 MB/s (2.0 Gb/s). It is important to note that the cards obtain good performance even for small messages. In each test half of the maximum observed bandwidth could be obtained using messages that were only 256 bytes long. This characteristic is especially important in cluster computing because many parallel applications frequently utilize small messages for state updates.

5.3.2 Overhead for the Sending and Receiving NIs

While the previous tests demonstrate that a significant portion of the available SAN bandwidth can be obtained for data transmissions, there are a number of operations that take place at the sending and receiving NIs that add to the overhead of NI-NI communication. Because these operations degrade communication performance, it is beneficial to determine and measure the individual tasks that must be performed in the NI-NI transmission process. Table 5.1 provides a listing of measurements made to perform various operations in both the sending and receiving NIs for the LANai 4 and LANai 9 NI processors.

Table 5.1: Reliably delivering a message incurs overhead at the sending and receiving NIs.

NI	Function	Time (μs)	
		LANai 4	LANai 9
Sending NI	Detect new message (1 Logical Channel / 8 Logical Channels)	1.0 / 5.5	0.5 / 1.5
	Set values in message	2.5	1.0
	Insert message in scoreboard	3.0	1.0
Receiving NI	Decode and begin processing message	2.5	1.0
	Verify sequencing information	2.0	0.5
	Destination capacity check	2.0	0.5
	ACK/NACK generation	4.0	1.5

As this table indicates, the sending and receiving NIs must perform a number of tasks before and after a message is transmitted. In the sending NI the message must first be detected by scanning the NI's outgoing logical channels. Once a message is detected the sending NI must

mark certain values in the message such as its sequence number and a token value that can be used to track the message. The sending NI must then record information about the message in a scoreboard, which allows the NI to keep track of the messages when retransmissions are necessary. At the receiving NI, an incoming message is first detected by the NI polling the network DMA interface. After a message has been detected the receiving NI must decode the message's header to determine how to process the message. For data messages the NI must verify that the message is marked with the next expected sequence number for the logical channel. Messages passing this test are then examined to determine if the specified destination endpoint has enough buffer space to accept the message. For accepted messages the NI must generate an acknowledgement message that is transmitted to the sender after the data message is transferred to the endpoint.

Performing these actions sequentially adds greatly to the overall overhead of the communication process between two NIs. Therefore, the NI firmware is designed to operate in a manner that allows some of these operations to overlap with DMA transactions. For example in the sending NI scoreboard updates take place after the NI begins transmitting the outgoing message to the network. Likewise in the receiving NI the firmware begins decoding and processing a message as soon as the first few bytes of the message's header begin to arrive. While it is complex to construct such concurrency in NI firmware, doing so shaves overhead off of the critical path in the communication library. The fact that GRIM provides competitive performance to other less sophisticated message layers indicates that the overhead of GRIM's increased functionality can be effectively hidden.

5.4 Ejecting Data from the Receiving NI (NI-Host)

The last stage in the host-to-host communication path is the receiving NI's ejection of data to the destination endpoint. In this phase data from active messages and remote memory operations must be transferred to the proper locations in host memory. The NI accomplishes this

task with the use of an on-card PCI DMA engine. For active messages data is appended to the back of a message queue for the host endpoint that is located in pinned, contiguous memory. A remote memory operation on the other hand requires the NI to transfer a block of data to host memory that is specified in the message. Completing this operation may require virtual to physical address translation by the NI depending on the arguments of the message.

5.4.1 Native NI PCI Performance

A first step in examining the performance of the NI-host ejection process is to determine the speed at which the NI can transfer data to host memory using its on-card DMA engines. A benchmark program was constructed in NI firmware to measure the amount of time required for a PCI device to transfer variable sized blocks of data from card memory to a contiguous region of host memory. This benchmark provides an estimate of the raw PCI performance a peripheral device can obtain from a host system. Three versions of the Myrinet NI card were used in this effort. The first two cards are the LANai 4 and LANai 9 NI cards that were used in the previous tests. The third card is a PCI mezzanine connector (PMC) version of the Myrinet LANai 4 card. This card is attached to one of the PMC daughter-card slots available on the Celoxica RC-1000 FPGA card that is discussed in Chapter 6. The RC-1000 card utilizes a PCI bridge unit to allow the LANai 4 PMC card to appear as a normal PCI device to the host system.

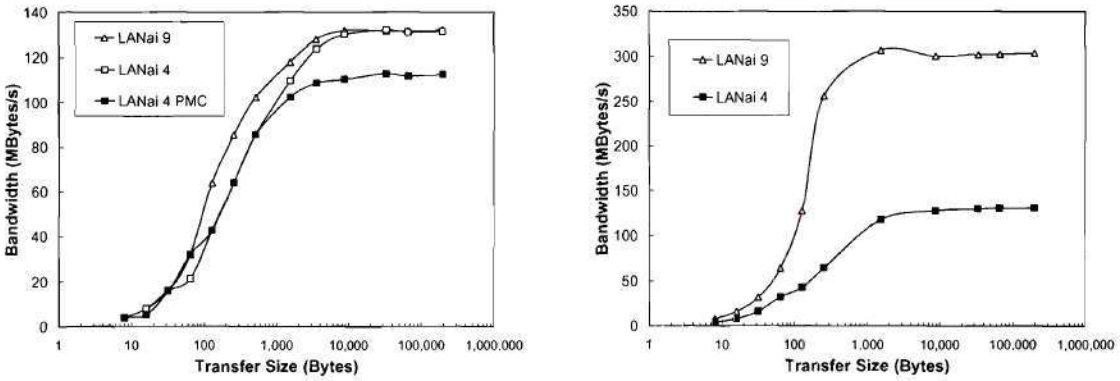


Figure 5.6: Bandwidth measurements for peripheral device DMA transfers into pinned host memory for (a) P3-550MHz and (b) P4-1.7GHz hosts.

The benchmark software was utilized to measure the PCI performance of the P3-550 MHz and P4-1.7 GHz hosts. The results of the experiment are presented in Figure 5.6(a-b). For the P3-550 MHz host (a), the normal LANai 4 and LANai 9 cards were able to obtain up to 132 MB/s using the 32b PCI bus. The LANai 4 PMC card was only able to obtain approximately 112 MB/s. This degradation in performance can be attributed to the fact that the PMC card's PCI transactions are routed through the bridge chip of the RC-1000 card. For the P4-1.7 GHz host (b), the LANai 4 card obtained 130 MB/s from the 32b PCI bus while the LANai 9 reached 307 MB/s from the 64b PCI bus. It is important to note that all of these cards were able to obtain reasonable performance levels, even with small transfer sizes. Half of the maximum observed bandwidth for these cards can be obtained using transfers that are only 256 bytes long.

5.4.2 Active Message Delivery

In the NI-host ejection stage in the host-to-host communication path, the NI implements separate delivery mechanisms for the active message and remote memory programming interfaces. The active message delivery mechanisms operate by inserting an incoming active message into a FIFO queue that is located in host memory. The NI is equipped with front and back indices for this queue so that it can determine where to place the next incoming message

without assistance from the endpoint. Once a message has been transferred to the queue, the NI must notify the endpoint of the arrival of new data. In GRIM this notification is provided by the NI updating a queue pointer that is located in the endpoint's address space.

Table 5.2: The amount of time required for the receiving NI to deliver an active message to the host endpoint for a P3-550 MHz host.

Action	Time (µs)	
	LANai 4	LANai 9
DMA (32B / 64KB) message to host	3.0 / 496.0	2.0 / 494.0
Increment and convert queue pointer	1.5	0.5
DMA queue pointer to host	4.0	1.5

Instrumentation software was added to GRIM's NI firmware to determine how much time is required for the NI to perform its ejection tasks. Table 5.2 lists the amount of time the NI takes to perform the tasks for ejecting an active message. The first step in the ejection process is for the host to transfer the active message to the message to the proper location in the host's message queue. As the previous subsection discussed, it is possible for this task to be performed as data is arriving from the network in a cut-through fashion. The second task is for the NI to increment its local back pointer for the message queue. The new pointer is also converted into a value that has the same byte order as the destination endpoint (e.g., big-endian for the x86 host). The final task in ejecting an active message is to update the endpoint's back pointer using a DMA transfer. This transfer is only four bytes long and cannot be performed until the previous DMA completes.

5.4.3 Remote Memory Execution

For remote memory messages the NI must perform a DMA operation using arguments that are specified in the incoming message. Three different remote memory operations are possible. The first type of remote memory is a remote memory write to a physical memory address (RM-P). The NI processes RM-P messages by transferring the payload of the message to

a physical address specified in the header of the message. As such these messages are executed immediately upon arrival at the NI without any external translation assistance. The remaining two remote memory operations utilize virtual addresses to reference memory and therefore require the NI to perform address translation. The virtual memory write operation (RM-V) writes payload data to a virtual address while the virtual memory read operation (RM-RV) operation transmits the contents of a block of virtual memory to the sender. All remote memory operations provide an optional mechanism for updating a separate lock variable in virtual memory when a remote memory operation completes.

Table 5.3: The amount of time required for the receiving NI to process a remote memory operation in a P3-550 MHz host.

Action	Time (µs)	
	LANai 4	LANai 9
Search NI VM translation cache (hit / miss)	3 / 10	1 / 4
DMA translation request to host	3	1.5
Host interrupt overhead	6.5	6
Host VM translation and NI update (1 page / 17 pages)	2 / 17	3 / 10
DMA message payload (4B / 64KB)	3 / 514	2 / 507
DMA update to lock variable (optional) (cached / non-cached address)	9 / 29	4 / 17

Instrumentation software was constructed to determine the amount of time required for the NI to process remote memory messages. Timings are reported in Table 5.3 for the LANai 4 and 9 NI cards. The first step in processing a remote memory operation is translating the virtual memory addresses supplied in the RM-V and RM-RV operations. For these operations the NI first consults a small cache of translations that is located in NI memory. If a translation cannot be obtained from this cache the NI must DMA a formal translation request to host memory and interrupt the host. The device driver for the NI parses these requests, translates the requested virtual memory addresses, and stores the results back into NI memory. It is important to note that translating virtual memory address is relatively expensive. However, translation overhead can

partially be hidden by allowing translation to take place while the message's payload is arriving. After the NI is equipped with the proper physical address it can begin transferring data between the card and the host. Once this operation completes a remote memory operation can optionally update a lock value in host memory. This operation requires a virtual memory translation as well as a DMA of 4 bytes.

5.5 Performance and Optimizations of the Communication Path

While it is important to consider the performance of individual stages in the communication path, it is also necessary to consider how the stages behave in the context of end-to-end communication. One method of transmitting data through multiple network elements is to utilize a store-and-forward communication model. In this model each network element must receive a data message in its entirety before the message can be transmitted to the next stage. While this model has poor performance for individual transmissions, it is possible to use store-and-forward transmissions in a pipelined manner for improving the performance of a series of transmissions. GRIM utilizes built-in fragmentation mechanisms to allow a store-and-forwards pipeline to be constructed in the communication library. These mechanisms allow data to be transferred in a high performance manner between host endpoints.

The performance of a store-and-forward system can be improved through the use of cut-through optimizations. In cut-through approaches network elements are permitted to begin transmitting a data message before the entire message has arrived at the network element. Cut-through techniques therefore reduce the amount of time between when a message begins to arrive at a network element and when transmission begins to the next stage in the communication pipeline. Cut-through optimizations have been applied in GRIM to both the sending and receiving NIs. While cut-through benefits are more visible at the receiving NI, both the sending and receiving NI cut-through optimizations provide noticeable improvements for the communication

pipeline. This section provides details of the optimizations used to increase the performance of the host-to-host communication path, as well as performance measurements of each optimization.

5.5.1 Store-and-Forward Communication Model

One approach to implementing a system that delivers data messages through a multi-stage communication path is to employ store-and-forward data transfers. In this approach, each stage in the communication path must receive a message in its entirety before it can begin transmitting the message to the next stage. An example of how store-and-forward data transfers take place in the host-to-host communication path is illustrated in Figure 5.7. In this example, the host-NI, NI-NI, and NI-host transmissions of a data message take place sequentially with no overlap. Because a message must serially propagate through all three transmission stages, it should be expected that the performance of the overall communication path will be roughly a third of the performance of an individual transmission stage. Using the maximum bandwidth available for each transmission stage (listed to the right of the figure), it is possible to determine the maximum bandwidth that can be obtained for the transmission of a single message through the overall communication path. Inverting the maximum bandwidth for a transmission stage yields the amount of time required to transfer a single byte through a stage. Assuming 32b PCI buses, the sum of the transmission times for the three stages is 21.39 ns. This value corresponds to a maximum host-to-host bandwidth of 46.75 MB/s.

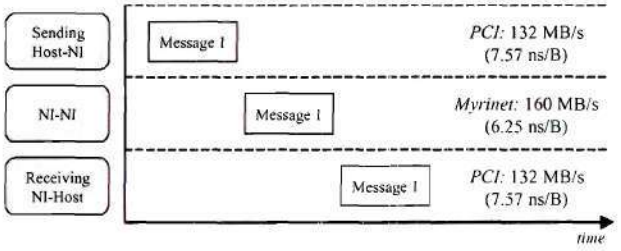


Figure 5.7: Messages are moved in their entirety in a store-and-forward communication model. The minimum amount of time required to transmit a byte can be determined by inverting the maximum bandwidth of the transmission mechanism.

A set of benchmark programs were constructed to observe GRIM's host-to-host performance. These programs use round-trip timing measurements between two hosts to determine the overall bandwidth that can be obtained from the hosts. The tests are performed for both the active message and remote memory programming interfaces. In the active message tests a special function handler is used to either return an incoming message to the sender or stop a timer if the host is the endpoint that originally transmitted the message. In the remote memory tests a block of data is transferred using the remote memory write physical (RM-P) operation. The notification mechanisms of the RM-P operation are used to update a memory location that holds a value that signifies the completion of a transfer. The sending and receiving endpoints in the remote memory test poll this notification variable and transmit blocks of data when necessary. These benchmark programs are used in all of the tests in this chapter that examine host-to-host performance.

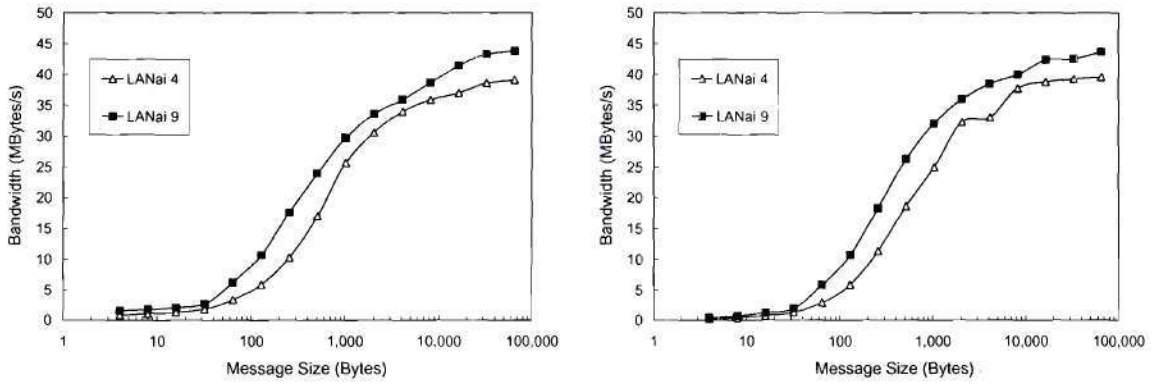


Figure 5.8: The store-and-forward performance for a single message transmission between a pair of P3-550MHz hosts using (a) active messages and (b) remote memory operations.

The host-to-host benchmarking programs were utilized to observe the performance of GRIM for the store-and-forward transmission of a single message between two P3-550 MHz hosts. The results of the experiments are presented in Figure 5.8(a-b) for message sizes ranging from four bytes to nearly 64 KB. For the tests using the active message programming interface (a), the LANai 4 and 9 NI cards reached maximum bandwidths of 39 MB/s and 43 MB/s respectively. The remote memory tests yielded nearly identical results. As expected the end-to-end performance of the system is less than the theoretical maximum bandwidth of 46.75 MB/s. This reduction in performance is due to processing overhead that takes place in the various stages of the communication path.

5.5.2 Store-and-Forward Pipelining

While the store-and-forward model of communication only offers limited performance for transferring a single message, it provides a framework for establishing a high-performance communication pipeline between a pair of endpoints. In a pipelined approach a series of messages is transmitted from one host to another in rapid secession. While each pipeline stage can only forward one message at a time, the abundance of messages to transfer allows each stage to operate at the same time on different messages. An example of how this concurrency can result in increased performance is illustrated in Figure 5.9. In this example, a series of messages are transmitted from one host to another. After the host finishes injecting the first message to the sending NI, it can begin injecting the second message. During this time the sending NI can begin transmitting the first message to the receiving NI. As the pipeline fills it becomes possible for more stages to operate concurrently, increasing the performance of the overall communication path. It is important to note that pipeline performance is dependent on the sizes of the messages that are being transferred as well as the transfer rates of the individual pipeline stages. For example in Figure 5.9, the third message is larger than the second message. Therefore, there is a

gap between when the NI-NI stage finishes transmitting the second message and when it can begin transmitting the third message.

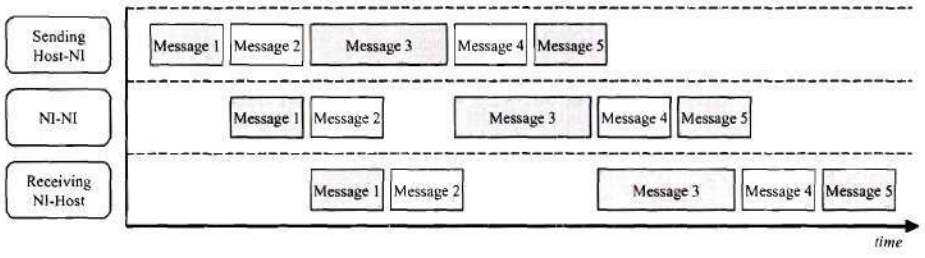


Figure 5.9: Store-and-forward mechanisms can be used to create a communication pipeline for increased performance.

Given the performance advantages of pipelining it is beneficial to include mechanisms in the message layer that allow transmissions to take place in a pipelined fashion. One means of accomplishing this task is to utilize fragmentation and reassembly techniques at the programming interface level. In this approach, large messages are broken into a series of smaller messages that are individually transmitted through the communication path and reassembled at the receiver. Because network hardware generally limits data transfers to a maximum transfer unit (MTU), most message layers naturally provide some form of fragmentation and reassembly. GRIM includes fragmentation and reassembly mechanisms for both the active message and remote memory programming interfaces. Low-level details of these mechanisms are provided in Chapter 8. These mechanisms were adapted to allow pipelining to take place in the communication path.

5.5.3 Pipelined Store-and-Forward Performance

The benchmarking programs used in the store-and-forward tests were modified to examine how fragment size affects the performance of the overall communication pipeline. In these tests the fragmentation size for the communication library was varied from 256 bytes to nearly 64 KB (GRIM's MTU). Host-to-host bandwidth was then measured using different sized

messages for both the active message and the remote memory test programs. The tests were performed for P3-550 MHz hosts using a pair of LANai 4 NIs and a pair of LANai 9 NIs.

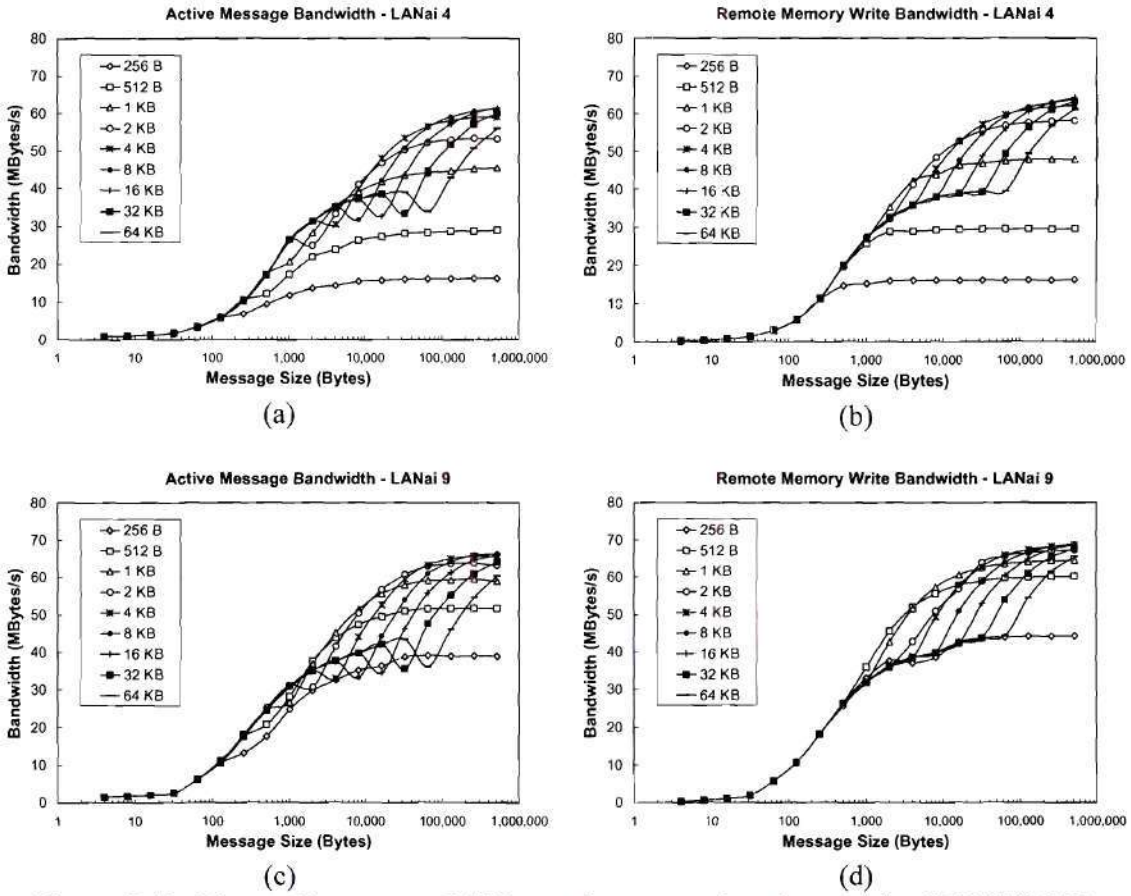


Figure 5.10: The performance of different fragment sizes for a pair of P3-550 MHz hosts using different NIs and different programming interfaces. The tests used (a) active messages with the LANai 4, (b) remote memory operations with the LANai 4, (c) active messages with the LANai 9, and (d) remote memory operations with the LANai 9.

The results of the experiments are presented in Figure 5.10(a-d). The first observation to be made from these measurements is that pipelining does in fact increase communication performance for messages that are larger than approximately 4 KB. For these messages there is enough data being transferred that a message can be fragmented in a manner that allows concurrency between pipeline stages. As expected best results in these tests were obtained when the fragment size was selected in the middle range of possible values. For the LANai 4 cards a

fragment size of approximately 2-4 KB provides good performance for both programming interfaces. The LANai 9 cards operate well with 1-2 KB fragments. The lower desirable fragment size can be attributed to the fact that the LANai 9 card is roughly three times faster than the LANai 4 card, thereby allowing a finer granularity of transmissions in the pipeline.

The benchmark tests also reveal that there are performance differences between the active message and remote memory interfaces. The most notable difference is that the remote memory performance curves generally increase with message size while the active message curves have slight performance drops at certain message sizes. These drops can be attributed to the reassembly mechanisms of the active message interface. Fragmented messages in the active message interface are reassembled in an intermediate message buffer. Therefore, as soon as message fragmentation takes place there is a slight drop in performance because the receiver must perform an extra copy of all message fragments. The remote memory reassembly procedures do not need to perform this procedure and therefore function more efficiently.

5.5.4 Cut-through Optimizations

While pipelining increases the performance of the host-to-host communication path, a criticism is that data transfers are still based on store-and-forward mechanisms. These mechanisms can cause a pipeline stage to delay the transmission of a message until the stage has received the entire data message. An alternative approach is to employ cut-through routing, where individual stages are permitted to begin transmitting a message as soon as the first bytes of the message arrive. For many LAN cards cut-through optimizations are not possible because network interactions take place at the packet level. However, Myrinet NI cards allow users to manage network interactions at the byte level. Therefore, it is possible to implement cut-through optimizations in Myrinet at both the sending and receiving NIs. An example of how cut-through optimization can be applied to the end-to-end communication path is illustrated in Figure 5.11. In this example, a data message is transferred as a series of smaller segments. Each stage in the

communication pipeline can therefore begin transmitting a message as soon as the first segment of a message arrives.

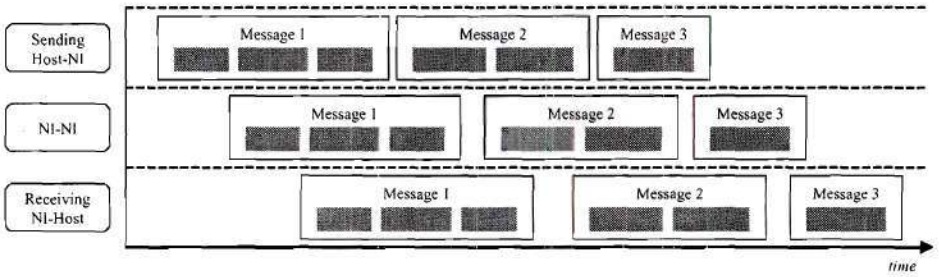


Figure 5.11: Cut-through optimizations allow a pipeline stage to begin transmitting a message before all of its data has arrived.

GRIM implements cut-through transfer optimizations for both the sending and receiving NIs. The receiving NI implementation is the more straightforward of the two because the receiving NI can easily coordinate all necessary data transfers. In the current implementation the receiving NI monitors the capacity of an incoming message buffer and then begins transferring data to the host as soon as the network begins to fill the buffer. Implementing cut-through optimizations in the sending NI is more challenging because of the manner in which data injections are performed in GRIM. In other message layers, the sending NI pulls a data message out of a host endpoint's address space and pushes the message to the network. As such it is trivial to implement the transfers in a cut-through manner because the transfers are performed entirely by the NI. Unfortunately, this approach is not valid for GRIM because messages are injected into the NI by endpoints in a push fashion. This push methodology is due to the fact that some endpoints in GRIM are peripheral devices that operate with simple transfer mechanisms.

Given the benefit of cut-through optimizations, special functions were constructed in GRIM to allow an injecting endpoint to achieve cut-through data transfers without having to resort to using the NI to pull messages into the NI. In GRIM the injecting endpoint and the sending NI can operate in a *cooperative cut-through* manner. In this effort, the endpoint breaks

the injection process of large messages into a series of smaller message segment injections. After transferring a segment to the NI, the endpoint updates a counter in the NI that specifies how much of the message has been transferred. When the sending NI detects a new message it begins transmitting as much of the message as is available to the wire. The NI then appends the network transmission as new segments arrive. This approach is cooperative in that in the common case, the endpoint and NI operate at the same time and transfer a message to the network in an efficient cut-through manner. However, there is no guarantee that the endpoint and sending NI will be synchronized to perform a cut-through transfer. In addition to increasing performance, this approach is advantageous because cut-through transfers can be accomplished without major modifications to the endpoint or NI software.

5.5.5 Performance with Cut-through Optimizations

The host-to-host performance benchmarks were used to examine the impact of cut-through optimizations on the communication pipeline. In these tests a fixed fragment size of 4 KB was selected for the transfers. Sending cut-though procedures were designed to segment a message into 1 KB blocks, while receiving cut-through mechanisms were designed to move data to the host endpoint as soon as it becomes available. The tests were performed for different message sizes using the LANai 4 and LANai 9 cards, with different cut-through optimizations enabled in each run. These tests utilized only the remote memory operation for the transfer, although active message performance provided similar behavior.

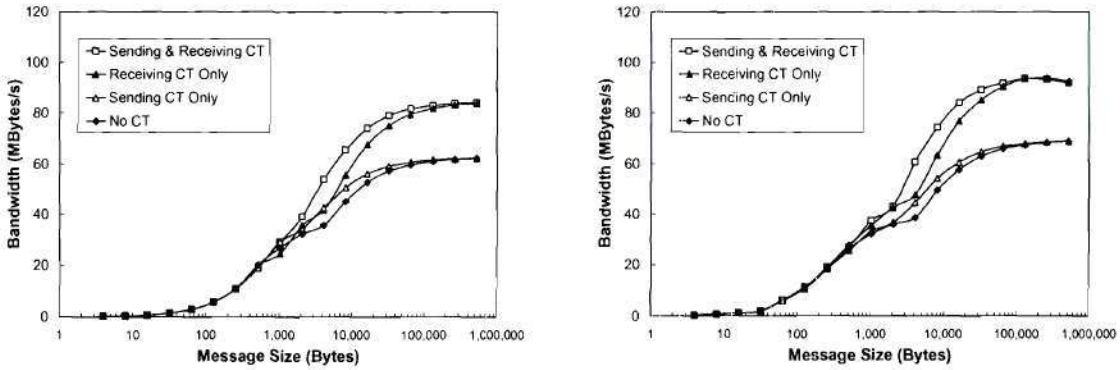


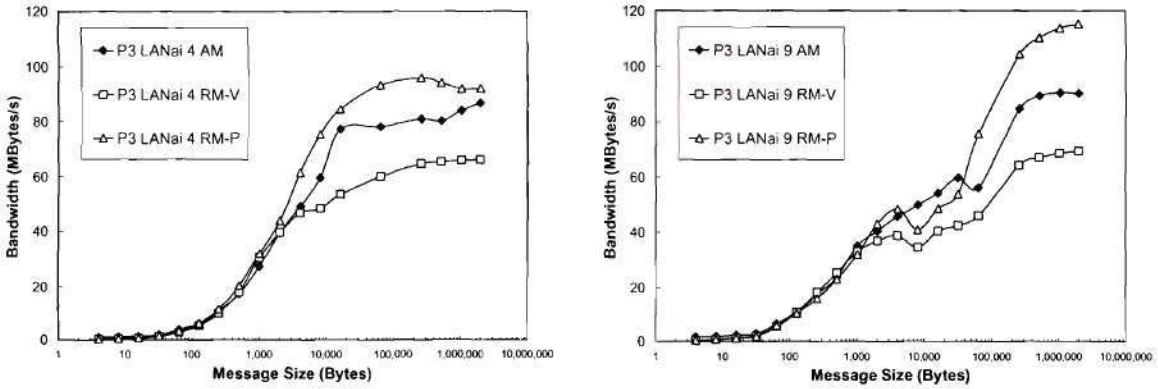
Figure 5.12: The effects of cut-through optimizations on end-to-end performance between P3-550MHz hosts using the (a) LANai 4 and (b) LANai 9 NI cards. These tests use RM-P programming interface, a fixed cut-through injection size of 1 KB, and a pipeline fragment size of 4 KB.

The results of the experiment are presented in Figure 5.12 (a-b). Cut-through optimizations in these tests provided a significant performance boost for both the LANai 4 and 9 NI cards. Receiver-based cut-through provided the most improvement in the tests due to the fact that it can operate at a fine granularity. In the best case the receiver cut-through mechanisms can begin transferring data to the host as soon as the header for the message arrives. Sender-based cut-through provides a slight performance gain. This gain is less than the receiver-based optimizations because the 4 KB message fragment size limits the sender to four injection segments per fragment.

Additional tests were performed varying the GRIM's fragment and segment sizes. Determining a good combination of these settings is highly dependent on the PCI transfer characteristics of the host and NI cards. For both LANai 4 and 9 NI cards, the performance bottleneck is the transfer of data from host to NI card. The LANai 4 card obtains maximum PCI injection performance for 1-2 KB transfers. Therefore, GRIM is configured with a segment size of 1 KB and a fragment size of 16 KB for the LANai 4 card. For the LANai 9 card, the DMA engines provide maximum performance at roughly 16 KB. Therefore, GRIM is configured to use a segment size of 16 KB and a fragment size of 64 KB for the LANai 9 card.

5.6 Overall Performance

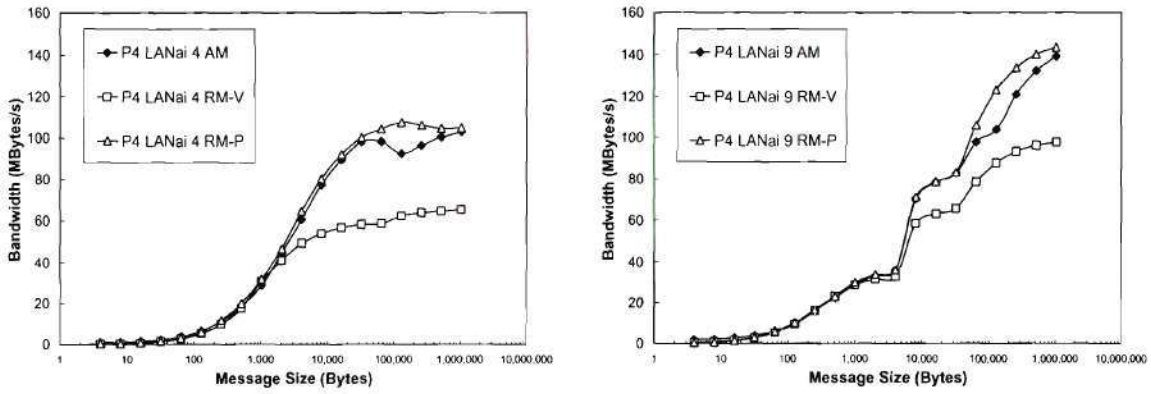
The host-to-host benchmark programs were run a final time with all performance optimizations enabled. The three programming interfaces were independently measured in this effort using message sizes ranging from four bytes to two megabytes. The LANai 4 and 9 NI cards were used to connect pairs of P3-550 MHz and P4-1.7 GHz hosts.



(a) (b)

Figure 5.13: Overall performance of P3-550 MHz hosts in GRIM using (a) LANai 4 and (b) LANai 9 NI cards.

The results of the experiments with the P3-550 MHz hosts are presented in Figure 5.13(a-b). For the LANai 4 NI cards (a), the performance curves are relatively smooth. The RM-P interface provides the maximum performance of 96 MB/s and the minimum latency of 14 µs. The LANai 9 card (b) offers better performance in this host, reaching a maximum bandwidth of 116 MB/s (928 Mb/s) and a latency of 9.5 µs for RM-P operations. The transition from PIO to DMA injection mechanisms for this card results in performance reaching a temporary plateau for messages between 4 KB and 32 KB.



(a) (b)

Figure 5.14: Overall performance of P4-1.7 GHz hosts in GRIM using (a) LANai 4 and (b) LANai 9 NI cards.

The results of the experiments for the P4-1.7 GHz hosts are presented in Figure 5.14(a-b). The LANai 4 cards (a) had to be placed in 32b PCI slots in the P4 hosts. These cards were able to obtain 108 MB/s of bandwidth and 14.5 μs of latency between the P4 hosts using the RM-P interface. Compared to the P3 tests, the P4s provide better bandwidth but slightly worse latency for the LANai 4 card. The P4's superior processing power also helps the more computationally demanding AM interface to provide performance that is closer to the RM-P interface than is observed in the P3 tests. LANai 9 cards (b) were placed in the 64b PCI slots of a pair of P4 hosts. In the performance tests the RM-P interface obtains 146 MB/s (1.168 Gb/s) of bandwidth and a minimum latency of 8 μs. Given that the SAN-1280 links offer a theoretical transfer rate of 160 MB/s, GRIM obtains a substantial portion of the available host-to-host performance.

Table 5.4: The overall performance of GRIM.

Host	PCI	NI	Latency (μs)			Bandwidth (MB/s)		
			AM	RM-V	RM-P	AM	RM-V	RM-P
P3 550MHz	32b/33MHz	LANai 4	16	18	14	87	66	96
		LANai 9	10	10.5	9.5	102	69	116
P4 1.7GHz	32b/33MHz	LANai 4	17	18.5	14.5	105	65	108
	64b/66MHz	LANai 9	9	8.5	8	144	98	146

GRIM's overall performance numbers are summarized in Table 5.4. An important observation of these numbers is that the RM-V interface provides only 60-75% of the

performance of the RM-P interface. This performance degradation can be attributed to the overhead of translating virtual memory addresses at the receiving NI. The reason why this translation has such a negative impact on RM-V performance is that the receiving NI must perform this operation before an incoming RM-V message can be processed. Because the translation is the first step in the NI's receiving process, it is difficult to hide the overhead of the operation with other transfers the NI must perform. Therefore, users should be aware that the RM-V interface is only capable of providing limited performance.

5.7 Comparison with Other Message Layers

Over the years a number of message layers have been constructed for Myrinet. It is beneficial to compare the host-to-host performance of GRIM to these message layers in order to gauge how well GRIM can perform in traditional cluster applications. These comparisons also reveal how well the complexities of GRIM's lower mechanics are hidden from the critical path for end-to-end communication. Unfortunately, many of the existing message layers are no longer supported and cannot be run on modern systems. As a means of comparison, this section provides two forms of performance estimates for existing message layers. First, performance estimates are provided for many message layers using the values reported by the original researchers. Second, performance estimates of the most commonly used message layer, GM, are provided for the same systems used in the GRIM benchmarks.

5.7.1 Reported Performance

It is important to compare the performance of GRIM with existing message layers for Myrinet. Performing such a comparison is challenging to do in an accurate and fair manner for a number of reasons. At a fundamental level GRIM is designed to provide functionality that is not present in other message layers. Therefore, comparisons must be limited to traditional host-to-host metrics. Unfortunately, utilizing previously reported values in this comparison can often be

misleading, as different utilize different hardware and software platforms, and sometimes different definitions of performance metrics. Ideally a fair comparison would run a standard set of benchmarks on the message layers using the same hardware and software environment. The hardship in this effort is that several of the message layers are now legacy software that is no longer supported due to changes in the Linux kernel or incompatibilities with modern hardware. Rather than port these legacy message layers to modern systems, the first part of comparing the performance of GRIM is to provide the performance measurements that were originally reported by the researchers.

Table 5.5: Performance reported for various Myrinet message layers

Message Layer	Host	NI	Latency (µs)	Bandwidth (MB/s)
AM [32]	UltraSparc 167 MHz (SBUS)	LANai 3	10	38
AM-II [33]	UltraSparc 167 MHz (SBUS)	LANai 4	21	31
BIP [40]	PPro-200 MHz	LANai 4	5	126
FM [34]	PPro-200 MHz	LANai 4	11	76.2
GM [39]	P3-1GHz (64b PCI)	LANai 9-200 MHz (Myrinet 2000)	7	240
LFC [36]	PPro-200 MHz	LANai 4	12	65
Trapeze [37]	P3-450 MHz	LANai 4	30	110

Reported performance estimates for a number of Myrinet message layers are listed in Table 5.5. The majority of these measurements are based on older hosts using the LANai 4 NI card. Therefore, the most relevant performance measurements of GRIM are those made of the P3-550MHz hosts that are equipped with the LANai 4 NI. In these tests GRIM obtained a maximum bandwidth of 96 MB/s and a minimum latency of 14 µs. In terms of latency GRIM offers slightly less performance than most of the message layers, but is still within an acceptable range. In terms of bandwidth GRIM is relatively competitive with other message layers.

5.7.2 Measured Performance

As a means of providing a more accurate comparison of GRIM's performance with other message layers, Myricom's GM [39] message layer was benchmarked for the P3 and P4 clusters. GM is the de facto standard for communication in Myrinet clusters and supports a variety of operating systems and NI cards. GM's internal benchmarking programs were utilized to determine how well the message layer performed using the same hardware that the GRIM benchmarks were performed with.

Table 5.6: Measured performance for GM and GRIM.

Host	PCI	NI	Latency (µs)		Bandwidth (MB/s)	
			GM	GRIM	GM	GRIM
P3-550 MHz	32b/33MHz	LANai 4	24	14	79	96
		LANai 9	9.7	9.5	108	116
P4-1.7 GHz	32b/33MHz	LANai 4	24	14.5	69	108
	64b/66MHz	LANai 9	9.44	8	146	146

The results of the GM benchmarking experiments are presented in Table 5.6. In all of these tests the performance of GRIM was observed as being slightly better than that of GM. GRIM particularly excelled in the benchmarks involving LANai 4 NI cards. This characteristic can be attributed to the fact that GM is largely targeted for LANai 9 cards and that a number of GM optimizations have to be disabled in order for the LANai 4 cards to function properly. It is important to note that GM is designed to be the most robust and reliable message layer for Myrinet. It is not the intention of this thesis to claim that GRIM is a better message layer than GM. Instead, these measurements are reported for the sake of demonstrating that GRIM provides comparable performance to state-of-the-art message layers such as GM.

CHAPTER VI

PERIPHERAL DEVICE EXTENSIONS

A key characteristic of message layers for resource-rich cluster computers is extensibility. From a hardware perspective, it must be easy for users to adapt these message layers to support new and diverse peripheral devices in the cluster. In this effort, peripheral devices are visualized as communication endpoints and added to the cluster's global pool of distributed resources. Therefore, the message layer serves as a general framework for interconnecting both host CPU and peripheral device resources. GRIM is unique in that it is a message layer that is specifically designed to provide this framework. Peripheral devices in GRIM interact directly with other resources in the local host (e.g., the NI or other local endpoints) using efficient PCI transactions. Peripheral devices with sufficient processing capabilities are allowed to operate in an autonomous manner without the guidance of the host CPU. For legacy peripherals that are less capable, GRIM can be configured to utilize host-level software to manage the device's interactions with the message layer.

This chapter focuses on the task of integrating peripheral devices into the cluster environment as communication endpoints. The discussion begins with a generalized description of how new peripheral devices are added to the GRIM environment. In order to construct new peripheral device endpoints, designers must be aware of the manner in which endpoint software is expected to function as well as the methods by which GRIM manages peripheral device resources. As a means of illustrating the integration process, the remaining portion of this chapter provides implementation details for four peripheral devices that have been incorporated into GRIM. These devices include an intelligent server adaptor card, an FPGA accelerator card, a

video capture card, and a video display card. These devices offer a diverse range of processing capabilities and help to demonstrate how GRIM can be used in a flexible manner to allow applications to utilize these resources.

6.1 Adapting a Peripheral Device for use with GRIM

GRIM is designed to allow multiple peripheral devices distributed throughout a cluster to be utilized as resources in the virtual parallel-processing machine. The approach taken in GRIM is to allow each peripheral device to function as a communication endpoint that interacts directly with the communication library. Therefore, the process of adapting a peripheral device to operate in the GRIM environment begins by constructing endpoint software for the peripheral device. This software is comprised of a set of device-specific active message function handlers for the device and message-passing mechanisms that allow the device to interact with other resources in the local host. Once endpoint software is available, a designer must construct host-level software to allow the device to be utilized in the GRIM environment. GRIM provides a series of built-in functions that can be used by designers to simplify this task. Finally, end users interact with peripheral device endpoints through a series of resource-management operations provided in GRIM. These functions allow a user to locate, reference, and communicate with a resource that is available in the cluster. The overall organization of the software for a peripheral device endpoint is pictured in Figure 6.1.

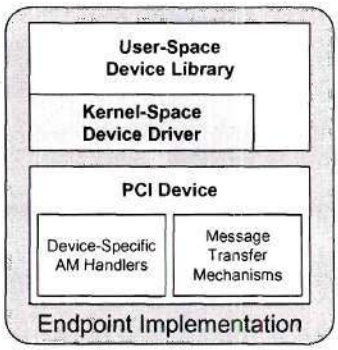


Figure 6.1: The major components of a peripheral device endpoint implementation.

6.1.1 Peripheral Device Endpoint Software

The first task in integrating a peripheral device into the GRIM environment is to construct low-level software that allows the device to function as a communication endpoint. This software is divided into two parts: device-specific active message function handlers and message-passing mechanisms. The active message portion of the software is responsible for serving as a means by which end applications can invoke specific operations at a peripheral device. A designer must therefore construct active message function handlers for a new peripheral device that adequately capture the device's key capabilities. After a device's function handlers have been constructed, a designer can add information about the handlers to a static database that is available in the GRIM library. This database allows a peripheral device's function handlers to be visible to applications in the cluster and removes the need for a peripheral device to register its handlers at runtime.

Low-level endpoint software must also implement message-passing mechanisms for communicating with the NI and other endpoints in the local host. For outgoing messages, an endpoint utilizes PCI DMA operations to transfer data to the message queues of other endpoints. For incoming messages, an endpoint allocates a block of memory for housing message queues that other endpoints can write. An endpoint must periodically poll its incoming message queues to determine if new messages are available. GRIM provides a set of files in C that can be used to simplify the task of implementing this functionality. These files include data structures for messages and message queues, as well as skeleton code for performing key message-passing operations. A designer can use this software by including the files in an endpoint implementation and then defining device-specific functions needed for the message-passing operations, such as a function for initiating a PCI data transfer.

6.1.2 Host-Level Integration

After endpoint software is constructed for a peripheral device, a designer must provide host-level software that allows the card to be utilized by the operating system and the GRIM library. The operating system portion of this software is implemented in a kernel-level device driver. While device drivers are nontrivial to implement, GRIM performs most of its operations in user space. Therefore, device drivers for GRIM can be relatively simple, and require only basic operations such as initializing the peripheral device and providing a memory map of card memory to user-space applications. Once a device driver is available, a designer must construct appropriate user-space software that allows the GRIM library to interact with the card. Typically, this software includes initialization functions and device-specific operations that may be needed by applications.

6.1.3 Library Initialization

The GRIM library must perform a number of initialization functions before the cluster can begin processing an application. The first step in the initialization process is for GRIM to load configuration information for the cluster from a set of user-defined configuration files. In addition to providing basic information such as routing tables, the configuration files specify the peripheral device resources that are available in the cluster. Each host in the cluster uses this information to determine which peripheral devices it is equipped with and how it should initialize its devices. Cluster configuration information is stored in a database that can be accessed by applications at runtime to help locate resource information.

After all devices in the local host have been initialized, GRIM must configure each peripheral device with information that allows the device to interact with the local NI and all of the other endpoints in the host. In this process, GRIM determines how many incoming message queues each endpoint needs and how large each queue should be. GRIM then updates all of the queue pointers for all of the endpoints in the local host. Two sets of pointers must be configured

for each message queue, one for the sending endpoint and the second for the receiving endpoint. When configuring these pointers, GRIM must translate all references to the message queue into values an endpoint can utilize. This procedure is complex, and automatically accounts for virtual-to-physical address translations, byte-order differences, and card-specific memory addressing issues. Needless to say, the automatic endpoint configuration functions are the most mind-numbingly complex part of GRIM. However, these operations are designed in such a way that when a new device is added to GRIM, users simply supply basic information to GRIM's configuration function and configuration takes place automatically.

6.1.4 Runtime Management

From an end user's perspective, a communication library for a resource-rich cluster computer must provide basic mechanisms for allowing users to customize their interactions with peripheral device endpoints. After initialization, the GRIM library provides a set of functions for performing such operations. With these functions, a user can query the communication library to locate a specific type of peripheral device. Users can perform these queries in the context of the cluster's global resources, or limit searches to particular hosts. When the communication library successfully locates a desired resource, it returns an integer identifier that can be used to reference the resource. Applications can then invoke operations at the resource simply by injecting active messages that are marked with the reference. Internally, GRIM provides all of the routing that is necessary for the messages to be delivered to the resource.

It is expected that some peripheral device endpoints will require more complex management functions than the current runtime system provides. For example, if an endpoint needs to use a peripheral device to perform a series of computations, it is beneficial if the endpoint can temporarily obtain exclusive ownership of the device so that the computations can take place without interruption. This type of operation can be implemented in GRIM by utilizing the peripheral device's host CPU to manage ownership of the resource. In this approach, the host

manages a reservation system for a peripheral device that is manipulated with active messages. When an endpoint needs to obtain ownership of the resource, it transmits the appropriate request to the host and waits for a response before accessing the device. Similar approaches can be used to layer additional functionality on top of the existing GRIM software.

6.2 Cyclone Microsystems Server Adaptor Card

The first peripheral device added to the GRIM communication library was the Cyclone Microsystems server adaptor card [68]. While originally marketed as a general platform for evaluating the Intelligent I/O (I_2O) [69] extensions, this card has become a valuable tool for active disk and active network research efforts [70]. The overall architecture of the card is presented in Figure 6.2. At the core of this architecture is an Intel i960 processor [71] that operates at 66 MHz. This processor includes a built-in 32b/33MHz PCI unit that features chained DMA engines and PCI doorbell registers. The card is equipped with 4 MB of on-card DRAM that can be expanded to 36 MB by populating a standard SIMM socket. In order to market the development card for different uses, Cyclone Microsystems placed a custom expansion interface on the card for attaching a daughter card. ATM, Ethernet, and SCSI daughter cards were constructed for the development card. The daughter card used in this research effort features two Fast Ethernet ports and two Ultra-wide, Ultra-fast SCSI ports. Communication between the i960 and the daughter-card components physically takes place using a secondary PCI bus.

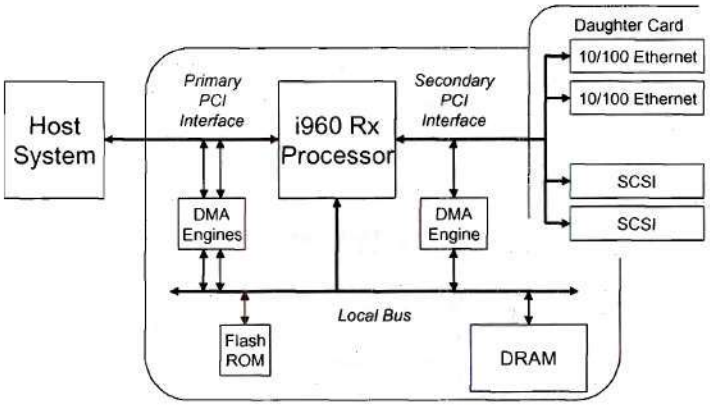


Figure 6.2: Architecture of the Cyclone Microsystems I_2O development card.

The Cyclone Microsystems development card uses a custom version of the VxWorks operating system [72] to control the card's hardware resources. VxWorks provides a UNIX-like, multitasking environment for applications. Application developers write programs for this environment in the proprietary Tornado development system [73], and then load the compiled binaries onto the i960 using either a serial download cable or the card's Ethernet controllers. While the VxWorks operating system greatly simplifies the development process with this card, it is important to note that the i960 may be underpowered for operations required by the operating system. The I_2O card was used in multiple research efforts at Georgia Tech, the most notable of which is the QUIC [74] project.

6.2.1 Endpoint Construction

The process of porting the GRIM endpoint software to function on the I_2O card was relatively straightforward due to the card's rich programming environment. The endpoint software was constructed to run as a normal VxWorks process on the I_2O card. At initialization time, the endpoint process allocates a block of memory for housing incoming message queues and shares this information with the host library. After initialization the process monitors the incoming message queues, processes messages, and ejects outgoing messages to the host's NI or

endpoints. Message ejections are performed using the card's DMA engines, allowing the I_2O card to operate autonomously without the guidance of the host CPU. With the help of Ivan Ganev, Robert Goldman, and Kelly Norton, a Linux device driver was constructed for the I_2O card that allowed the card to be utilized by host applications.

Multiple active message function handlers were constructed for the I_2O card to provide end users with a means of controlling the card's hardware.

Initial development with this card focused on constructing handlers for both network operations and storage operations. However, soon after this development began Cyclone Microsystems announced that the disk controller hardware for this card did not operate in a reliable manner. Therefore, the focus of this work shifted to utilizing this card exclusively for its network hardware. Active message handlers were constructed so that the I_2O card could be utilized as a network bridge, relaying messages between the SAN and the LAN when needed. These types of operations are necessary when a cluster is utilized as a large-scale network server that external hosts connect to through a LAN.

Active message handlers were constructed for the I_2O card to allow messages to be transferred between internal cluster resources and external Ethernet-based hosts. In this system an external host connects to the I_2O card using a long-term socket. Once a connection is established, transactions between the internal resources and external hosts can take place using a special active message that performs bridging operations. This message allows a normal active message to be encapsulated as the payload of the bridging message. When the I_2O card receives a bridge message, it extracts the encapsulated active message from the payload and forwards it to the appropriate resource in the cluster. In order for bridging to take place the I_2O card must maintain a table of Ethernet TCP connections so that it can relay data to the proper connection.

6.2.2 Performance Measurements

The I_2O endpoint implementation was constructed for an early version of GRIM. While this card has several shortcomings, including several documented hardware problems, it was sufficient for an initial proof-of-concept demonstration. Simple performance measurements were made of the endpoint implementation and are presented here. The early version of GRIM used in these measurements was less robust than the current version and thus incurred less overhead in host-to-host transmissions. For example, the early version of GRIM featured 13 µs latencies as opposed to 16 µs latencies for host-to-host transmissions. This difference should be taken into account when evaluating the performance of the I_2O card.

A test program was constructed to determine how efficiently a host-level application could interact with an I_2O endpoint. In this test, the time required to have a message sent to and returned from the I_2O card was measured. The resulting communication path is for host-NI-NI-I_2O, and was performed using the LANai 4 version of the NI card. The one-way travel time for a short message was measured to be approximately 21 µs. This latency is much larger than that required for two hosts to communicate. While the I_2O card is situated in close proximity to the NI card, the i960 processor is much slower than the host processor. Additionally, the I_2O card's DMA engines are designed to transfer large blocks of data and therefore have a large overhead for performing small PCI transfers. However, the I_2O card illustrates that a peripheral device can operate in an autonomous manner in the communication library and serves as an example of how intelligent peripheral devices can directly interact with the NI.

6.3 FPGA Accelerator

Field-programmable gate arrays (FPGAs) are reconfigurable hardware devices that can be programmed to function as application-specific circuits. In recent years, commercial FPGAs have grown significantly in capacity, and are now capable of emulating large blocks of custom

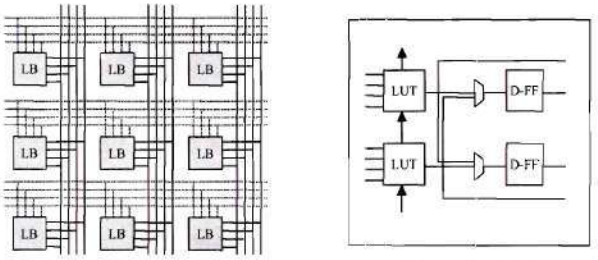
computational circuitry. These circuits can be utilized as a means of accelerating application performance in a cluster, and therefore it is beneficial to consider methods by which FPGAs can be incorporated into a cluster. Currently, there are multiple FPGA cards that are commercially available for accelerating a host's computational performance. One of these cards is the Celoxica RC-1000 [75], which features a modern FPGA and large amounts of on-card SRAM.

In order to better investigate the use of FPGAs as computational resources in the cluster environment, the RC-1000 FPGA card has been adapted to function as a communication endpoint in GRIM. This process was nontrivial, as endpoint software had to be converted into hardware circuitry. This circuitry is referred to as the static frame for the device, and is responsible for managing interactions between the FPGA and other resources in the local host. A second block of circuitry referred to as the circuit canvas is used in the FPGA as a place for housing multiple user-defined computational circuits. These circuits are the hardware equivalent of the active message handlers found in other GRIM endpoints. This section provides basic details of the RC-1000 FPGA endpoint implementation. Additional details are provided in the following chapter as well as in Appendix B.

6.3.1 FPGA Overview

In order to use FPGAs in the cluster environment, it is first necessary to understand the basic characteristics of the technology. FPGAs are reconfigurable hardware devices that can be programmed to emulate custom, application-specific circuits. Unlike application-specific integrated circuits (ASICs), which cannot be reprogrammed, FPGAs can be reconfigured at runtime to emulate different circuits that are needed by applications. A high-level architecture of an FPGA is presented in Figure 6.3(a-b). In this architecture, an FPGA is comprised of (a) a two-dimensional grid of (b) programmable logic blocks. Each logic block contains a lookup table that emulates a desired logic function. Logic blocks are connected to implement more complex operations through a programmable interconnection network inside the FPGA. In addition to

lookup tables, modern FPGA architectures include complex structures such as blocks of memory, high-speed multiplier arrays, general-purpose CPU cores, and high-speed network transceivers [76]. State-of-the-art FPGAs are advertised as being capable of emulating up to 8 million logic gates at a time [77].



(a) FPGA Architecture (b) Logic Block

Figure 6.3: FPGAs are generally (a) large arrays of programmable logic blocks (LBs) that feature (b) lookup tables (LUTs) and D-flip flops.

Designs for FPGA devices are generally constructed in hardware-description languages such as VHDL or Verilog. While these languages can simplify the design process, it is important to note that designing hardware is still significantly more time-consuming than designing software for a general-purpose CPU due to the low-level nature of the work. Once a design is debugged with simulation tools, it is synthesized into a gate-level description that is targeted for a particular family of FPGA devices. This description is then placed and routed for a target FPGA architecture using tools provided by the FPGA vendor. The end result of this process is a configuration file that can be loaded into the FPGA. Depending on the size and complexity of a design, it may take anywhere from tens of minutes to tens of hours for the entire compilation process to complete. Programming an FPGA with a configuration file can take several milliseconds in modern FPGAs.

6.3.2 Celoxica RC-1000 FPGA Card

The FPGA card chosen for integration into the GRIM communication environment is the Celoxica RC-1000 FPGA card. This card features a Xilinx Virtex-1000 FPGA [78], 8 MB of on-card SRAM, and PCI Mezzanine Card (PMC) [79] sockets for connecting two PCI daughter cards. A LANai 4.3 version of the Myrinet NI card was available at Georgia Tech in a PMC form factor, and allowed the NI card to be attached directly to the RC-1000 FPGA card. Figure 6.4 illustrates the overall architecture of our FPGA-enhanced NI card and the major hardware components of the individual cards.

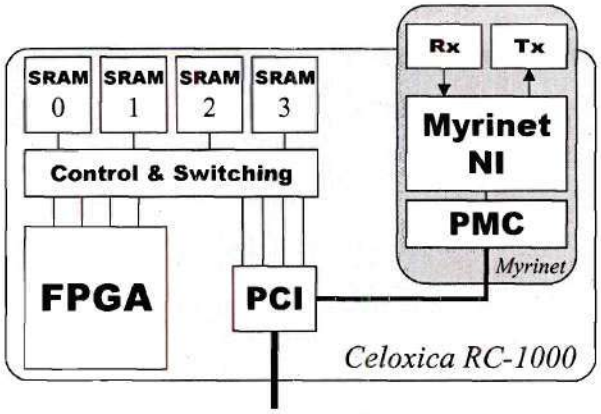


Figure 6.4: Celoxica RC-1000 and Myrinet Peripheral Devices

The architecture of the Celoxica RC-1000 card divides on-card memory into four 2 MB banks of SRAM. Each bank is single ported and operates with 32-bit data values. The RC-1000 provides switching hardware and memory arbitration mechanisms to allow the single-ported SRAM banks to be accessed by either the FPGA or the PCI controller. The arbitration mechanisms are implemented in a CPLD through the use of two 4-bit request registers (one for the FPGA the other for the PCI unit), and one 4-bit grant register. In this scheme, exclusive ownership of a bank of SRAM is acquired by updating a request register and polling the grant register. Memory arbitration does not take place automatically for the PCI controller. Instead, the external entity initiating a transfer of data to the RC-1000's memory must first interact with the

card's CPLD and obtain ownership of the involved SRAM banks. The PCI controller for the RC-1000 is the PLX-9080 chip [80], which provides a chained DMA engine for PCI transfers. Due to the manner in which the memory arbitration mechanisms are implemented for this card, the FPGA cannot initiate its own PCI DMA operations. This implies that the card requires the host's assistance whenever the FPGA exchanges data with the host.

6.3.3 FPGA Endpoint Implementation

Integrating the RC-1000 FPGA into the GRIM environment involved defining a hardware configuration for the FPGA that allows the FPGA to function as a communication endpoint. This hardware configuration must satisfy three design goals. First, it must be capable of managing the card's incoming and outgoing message queues for interactions with the communication library. In addition to injecting and ejecting messages, the FPGA configuration must be capable of parsing an incoming message to determine which hardware circuitry should be used to process the message. Second, the configuration must provide simple mechanisms that allow computational circuits to access the RC-1000's on-card SRAM that is not used for the message queues. This memory can be used to store application data sets in order to improve computational performance. Third, the FPGA configuration must allow multiple user-defined computational circuits to be loaded in the FPGA for use by applications. These circuits are analogous to software-based active message function handlers found in other endpoints.

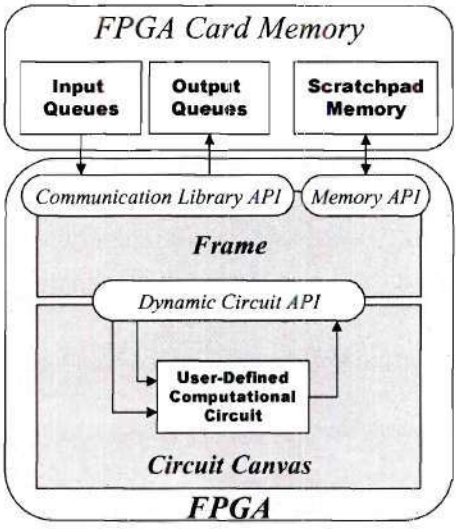


Figure 6.5: FPGA Organization

Based on the preceding requirements, circuitry was designed for the FPGA to allow the RC-1000 to function as a communication endpoint. As presented in Figure 6.5, the design divides the FPGA into two regions, and uses three separate interfaces to allow interactions between hardware units. The majority of the FPGA is used for the *circuit canvas*, a region of the FPGA that houses multiple user-defined computational circuits. These circuits process incoming messages and must adhere to a dynamic circuit API. The other portion of the FPGA is allocated for use as a *frame* for the canvas. The frame is a small region of the FPFA that is utilized to control card-specific interactions between the circuit canvas and the FPGA card's resources. The frame includes state machines for managing interactions with the communication library, the circuit canvas, and user-accessible on-card memory referred to as the scratchpad. Because the frame is designed to insulate the canvas from card-specific features, it is possible to port this work to other FPGA cards by modifying the frame.

6.3.4 User-Defined Circuits

One of the key features of the RC-1000 endpoint implementation is its capability for supporting multiple user-defined computational circuits in a single FPGA. The frame is designed with an interface that allows multiple computational circuits to be physically loaded in the canvas

and dynamically utilized to process incoming messages. The frame employs two sets of signals to accomplish this task. First, each computational circuit is connected to the frame by a set of control signals. These signals allow the frame to activate a computational circuit and detect when the circuit has completed its work. The second set of signals routes vector data streams between the frame and the computational circuit. A vector data stream transfers a linear series of 32-bit data values using an asynchronous transfer protocol. Each computational circuit can use up to two input vector data streams and one output vector data stream.

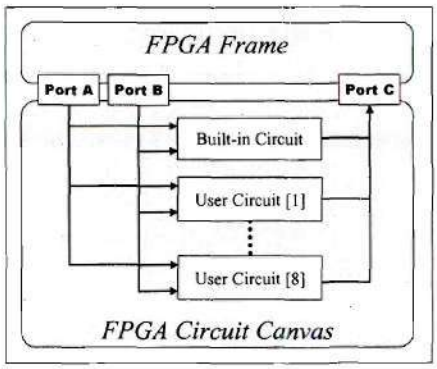


Figure 6.6: The interface for computational circuits in the canvas.

A simplified block diagram of the interface between the frame and computational circuits housed in the canvas is presented in Figure 6.6. Each computational circuit is provided with vector data streams supplied by ports A and B in the frame. Results of the computations are streamed back into the frame through port C. In addition to supporting up to eight user-defined computational circuits, each FPGA configuration is also equipped with a built-in unit for basic ALU operations. This unit provides a set of linear, two-input vector operations (e.g., add, multiply, AND, OR, XOR, min, and max) as well as one-input vector operations (e.g., NOT and a no-operation). The no-operation is beneficial because it can be used to transfer memory from one memory location in the scratchpad to another.

6.3.5 Examples of User-Defined Circuits

Multiple user-defined computational circuits have been constructed for use in the circuit canvas. As a means of illustrating how a modern FPGA can house multiple complex hardware units, an FPGA configuration with a set of cryptography cores was implemented. The post-layout design was examined to determine how much space each core occupies in the Virtex-1000 FPGA. These estimates are presented in terms of the percentage of the Virtex-1000's overall resources that are consumed by the circuits. These percentages can be translated to gate counts using the estimation that a Virtex-1000 can emulate approximately one million logic gates. Each core is briefly described as follows:

- **Digital Encryption Standard (DES)** [81] (6%): A publicly available DES core called free-DES [82] was ported to operate as a user-defined core. This unit can either encrypt or decrypt data supplied by vector data port A using a key supplied by vector data port B.
- **RC6** [83] (13 %): Chris Wood implemented a version of the RC6 encryption standard for encrypting and decrypting data from vector data port A using a key schedule supplied by vector data port B. The engine operates with up to 1024 rounds (R), at 32-bit data value widths (W), with key lengths (B) up to 1024 bytes.
- **MD5** [84] (26 %): The MD5 message-digest algorithm was implemented to generate a 128-bit identifier for data supplied by vector data port A.
- **Built-in ALU Operations** (5 %): The frame features a built-in ALU core for basic vector operations.

The frame for this configuration requires approximately 20% of the Virtex-1000's resources. A significant portion of this allocation is for two blocks of the FPGA's internal SRAM

so that the frame can buffer messages that are being processed. This design illustrates that multiple useful circuits can be loaded in the FPGA at the same time.

6.3.6 RC-1000 Interactions with GRIM

In order to allow the RC-1000 to function as a communication endpoint in GRIM, low-level message-passing mechanisms were constructed to facilitate data transfers between the RC-1000 and other resources in the local host. The first difficulty in this effort is that the FPGA cannot initiate DMA transfers. This problem was resolved through the construction of host-level software that initiates DMA transfers on behalf of the RC-1000 card. This software detects when the card needs to perform a transfer and issues the proper DMA operation. A more challenging issue in implementing the RC-1000's communication mechanisms is dealing with its card-specific memory-arbitration mechanisms. As discussed earlier, the RC-1000's PCI controller and FPGA share access to the card's single-ported SRAM banks through an arbitration scheme. Thus, when an endpoint needs to inject a message into one of the RC-1000's incoming message queues, it must first obtain exclusive ownership of the memory bank that houses the queue. Ownership is acquired by updating the card's request register and then polling a grant register to detect when the card has assigned ownership to the endpoint.

While the RC-1000's memory-arbitration mechanisms are adequate when only the host CPU uses the card, there is the possibility of a hazardous race condition when multiple resources in the host (e.g., the NI and the host CPU) access the arbitration mechanisms at the same time. The problem is that there is only one register for all off-card resources to place memory-arbitration requests. If an endpoint does not have knowledge of the current requests made by other endpoints in the host, it is possible for access to an SRAM bank to be released mistakenly. For example, consider the case where the host CPU and NI are each injecting a message into different queues located in the same SRAM bank of the RC-1000. If the NI finishes before the host CPU, it could mistakenly update the RC-1000's bank request register to indicate that access

is no longer needed to the SRAM bank. Because the memory arbiter only observes the most recent update to the request register, it is possible for the arbiter to change ownership of the SRAM bank from the PCI interface to the FPGA. This series of events results in the host utilizing an SRAM bank for which it no longer has access.

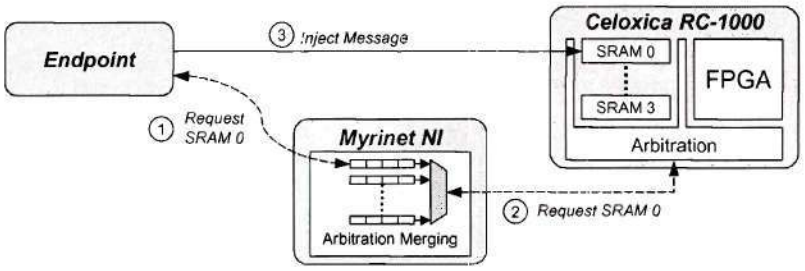


Figure 6.7: The use of the Myrinet NI to merge arbitration requests. An endpoint (1) passes a request to the NI to access RC-1000 memory. The NI merges the request (2) and interacts with the RC-1000 card. After access is granted the endpoint (3) injects the message.

A solution to the problem of allowing multiple resources in the host to coherently share access to the memory-arbitration mechanisms of the RC-1000 is to merge access requests at a single location in the host. In the GRIM implementation, this task is delegated to the Myrinet NI due to its proximity to the FPGA card on the PCI bus. As illustrated in Figure 6.7, a series of request queues are implemented in the NI. When the NI detects a new request in a queue, it updates a local set of registers for the queue and compares the sum of all the current requests to the last request issued to the FPGA card. If there is a difference, a new request is sent to the RC-1000's arbitration unit through a PCI transaction. If this update is due to a new request for memory (as opposed to a release), the NI periodically polls the RC-1000's arbitration unit until the request is granted. The endpoints in the host system must poll the NI's arbitration registers to determine when access is granted for each request.

Table 6.1: Performance measurements for RC-1000 memory arbitration.

Operation	Resource Requesting Arbitration	Resource Performing Arbitration	Time (µs)
Acquire SRAM	Host	Host	6
	Host	NI	13
	NI	NI	5.5
Release SRAM	Host	Host	2
	Host	NI	7
	NI	NI	3

Tests were performed to characterize the RC-1000 memory arbitration mechanisms. In these tests both the host and the NI acquire and release ownership of a bank of RC-1000 SRAM. In the first set of experiments, arbitration is performed directly by the host or the NI. In the second set of experiments, the host utilizes the NI to perform arbitration on behalf of the host. The results of these experiments are presented in Table 6.1. As expected, utilizing the NI to perform arbitration for the host results in a significant performance penalty for the host. The host's indirect arbitration scheme incurs twice as much overhead as a direct approach. These measurements imply that it is beneficial for a host endpoint to invoke arbitration mechanisms infrequently, and that the host software should be designed to bundle multiple transactions with FPGA memory into a single operation whenever possible.

6.3.7 TPIL Performance for the RC-1000 Card

The TPIL software was adapted to operate with the RC-1000 card in order to improve host injection performance. The internal benchmarking features of TPIL that were used to measure the performance of the Myrinet cards in Chapter 5 were utilized to measure the performance of the RC-1000. The results are presented in Figure 6.8 for a P3-550 MHz host that is equipped with a 32b/33-MHz PCI bus. As expected, MMX and SSE PIO based transfers provide the best performance for injections smaller than approximately 3 KB. After this point, zero-copy DMAs become the most profitable transfer method, eventually reaching a performance of approximately 116 MB/s.

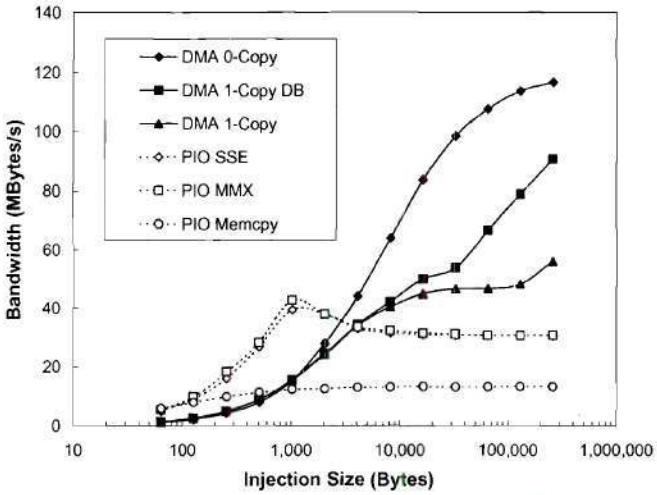


Figure 6.8: RC-1000 PCI injection performance for P3-550 MHz hosts.

An interesting characteristic in these performance measurements is that PIO operations reach a peak value of 42 MB/s for transfers that are 1 KB in size. Performance gradually drops for transfers larger than this size until a steady-state value of approximately 32 MB/s is reached. The RC-1000's PCI chipset is likely to be the cause of this performance drop because the PCI chipset has a limited capacity for buffering incoming data from the PCI bus. PIO transfers can therefore be slowed if the host CPU attempts to inject data faster than the card can empty the buffer. DMA transfers do not experience this performance degradation because the DMA engine can throttle its transfers to match the capacity of the buffer.

6.4 BrookTree Video Capture Card

The third peripheral device used as a resource in the GRIM environment is a video capture card based on the popular BrookTree BT8x8 chipset [85]. Many commercial video capture cards utilize this chipset because it is economical, and because device drivers exist for both Windows and Linux operating systems. The chipset integrates a PCI controller, video capture circuitry, and control logic in a single chip. BT8x8 card utilizes a block of host memory to serve as a frame buffer and employ a highly-configurable data-transfer engine to move

incoming video data to the host. Unfortunately, this card does not feature a programmable CPU that could be used to implement GRIM endpoint software. Therefore, a host-level library was constructed to allow GRIM applications to access this device. This effort demonstrates that GRIM can be utilized to include even simple peripheral devices in the cluster communication model.

6.4.1 Overview of the BT8x8 Video Capture Card

Many commercial video capture cards are based on the BrookTree BT8x8 chipset. This chipset is popular because it implements all capture hardware in a single chip, can process NTSC, PAL, and SECAM analog video sources, and can perform operations such as scaling, clipping, and pixel-format transformations in hardware. In order to reduce the cost of the chipset, the BT8x8 chipset is designed with minimal buffer space for housing video data. Instead of buffering frames of video data in card memory, the BT8x8 uses a programmable DMA engine to stream captured data directly into a region of the host's memory. A desirable aspect of this approach is that a frame of video data can be streamed to any location in the host, including the on-card frame buffer of a video display card.

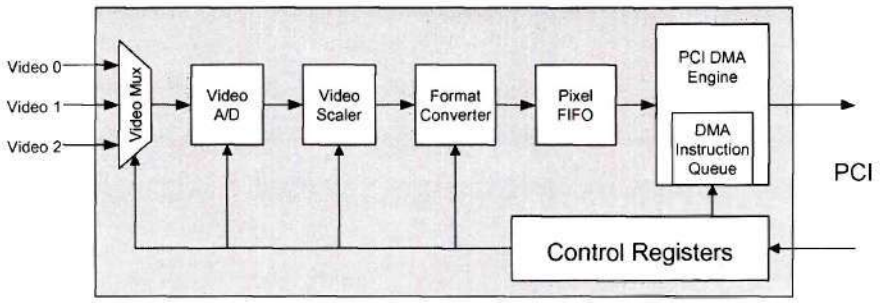


Figure 6.9: The high-level organization of the BT848 chipset.

An overview of the hardware pipeline that transforms analog video into a raw digital data stream in the BT848 chipset is presented in Figure 6.9. The input to the pipeline can be selected from one of three analog video sources using a programmable multiplexer. This input is

transformed into a digital data stream using an analog-to-digital converter. Digital data at this point is represented in a 640x480 pixel frame using the 4:2:2 YCrCb format [86]. The next stage in the pipeline is a video scaling unit that can downsample the data stream to user-defined dimensions. The data is then passed through a format-conversion engine that transforms the 4:2:2 YCrCb image into other formats (e.g., RGB565 or RGB32). Video data is then queued in a series of pixel FIFOs, where data is held until a PCI DMA engine transfers the data to host memory. The DMA engine performs data transfers based on a list of transfer instructions that are assembled by the host's device driver for the card.

Open source device drivers are available in the Linux operating system for BT8x8 video capture cards. These drivers are part of the video-4-linux (v4l) effort [87] and provide a basic API for user-space applications to interact with video capture cards. Because the BT8x8 chipset does not employ an on-card frame buffer, the driver must allocate a block of host memory for housing captured data and program the card with a list of DMA operations to store data into the memory. The driver allows users to create two frame buffers for each card so that the card can write data into one buffer while an application reads data from the other buffer. Using this double-buffered approach, it is possible for an application to acquire 640x480 pixel video data at 30 frames per second (NTSC's frame rate).

6.4.2 Endpoint Construction

In the process of examining how the BT8x8 video capture card can be integrated into the GRIM environment, it was observed that adapting endpoint software to run on the card would be infeasible for three reasons. First, while the card does provide a programmable RISC engine, this engine is primarily designed to simply transfer data between the card and host memory. Second, the card does not provide any memory that could be used for implementing on-card message queues. Finally, the fundamental nature of the card makes it impractical for use as an autonomous endpoint. Outside of configuration operations, the card performs the single function of writing

data to a memory location in the host. Therefore, it was decided that the integration of the BT8x8 card into the GRIM environment should leverage the existing v4l work and utilize the host as a unit for managing cluster interactions with the card.

The software that allows the BT8x8 to be used in the GRIM environment is comprised of initialization functions and active message handlers. The initialization functions connect to the v4l device driver and acquire allocations of host memory for housing video frames. After initialization, users can utilize a number of active messages for controlling interactions with a video capture card. In addition to active messages that allow users to configure the video capture software, a special active message is available for requesting that a frame of video data be transmitted to an endpoint. This request message allows users to specify the active message handler that is used in the reply message that carries the video data. This interface allows users to construct their own interfaces for processing incoming data without having to specify the mechanics of how the video capture card obtains the data. If the user does not specify a handler to use in the reply message, a built-in handler is selected that simply transfers the message's payload to a user-specified memory address.

Additional active message handlers were constructed to allow the node with the video capture card to transmit video data to another node in the cluster using remote memory operations. With these functions a node can have a frame of video data transferred directly to a frame buffer or a display device. Given that the dimensions of a frame of captured video may not match the dimensions of the output display, it is necessary to break the remote memory write operation into a series of smaller transfers. For example, if the frame size of the captured video is smaller than the size of the display, the captured data must be transmitted one row at a time in order for the rows to be rendered properly. The GRIM active message function handlers can perform this operation if the user supplies the dimensions of the target display.

6.4.3 Driver Modifications

The V4L device driver was modified to allow captured video data to be stored to a region of memory specified by the user. These extensions allow users to pass a physical memory address to the driver for a block of contiguous memory that is large enough to house a frame of video data. The GRIM software was also extended to allow users to reserve a contiguous block of NI card memory for housing application data. This effort provided the fundamental means by which the capture card could transfer video data directly to the Myrinet NI card in order to reduce the number of PCI transactions required by a system that needed to stream video data from one host to another. Test applications were constructed to demonstrate that the video card could in fact transfer data directly to the proper location of the NI card.

Table 6.2: Characteristics of BT8x8 video streams.

Frame Size (Pixels)	320x240	640x480
Bytes/Frame (16b Pixels)	150 KB	600 KB
Bytes/Second (@30 Frames/Second)	4.5 MB/s	18 MB/s
Bits/Second (@30 Frames/Second)	36 Mb/s	144 Mb/s

While the basic mechanisms for streaming video data directly into the NI have been constructed, development was halted due to a lack of practicality. Table 6.2 lists the characteristics of two video streams the BT8x8 card is capable of creating. If the video capture card is configured to store data directly to the NI, a buffer large enough to house two frames of data must be allocated from NI card memory. With only a megabyte of SRAM, the LANai 4 NI would only be able to support the 320x240 pixel video stream. While the LANai 9 could support 640x480 pixel frame buffers, doing so significantly reduces the amount of memory that is available for the NI to buffer normal messages. In contrast, the 640x480 data stream requires only 18 MB/s of bandwidth from the host's PCI bus. Because this data rate is only 14% of the available bandwidth of a 32b/33-MHz PCI bus, it was decided that the extra effort required to reliably stream data directly into the NI was not worth the possible performance gain. Instead,

software was designed to store captured video data in an intermediate host buffer and then transfer data to the NI as needed by applications. This method is able to transfer 640x480 pixel data streams in 30 frames per second.

6.5 Video Display Cards

Video display devices are another form of multimedia peripheral device that can be utilized in the cluster environment. While there are numerous commercial video display cards, the vast majority of these cards operate on the principle that display data is housed in an on-card block of memory known as the frame buffer. Writing graphical data into this block of memory results in changes in the rendered output. Therefore, video display cards can be incorporated into the GRIM environment by making the frame buffer accessible to end applications. GRIM has been extended with functionality to allow the frame buffer in a host to be identified to end applications. End users can then use this information with remote memory write operations to update the display of a remote host in an efficient manner.

6.5.1 Video Display Card Overview

Modern video display cards generally employ a large (2-128 MB) block of memory known as the frame buffer for housing video display data. The frame buffer accelerates system performance by allowing video data to be stored locally on the video card. The output display engine for the graphics card therefore continuously reads from the frame buffer and uses digital-to-analog converters to generate the appropriate VGA signals. A video display card is typically connected to the host system through the accelerated graphics port (AGP) [88]. This port is similar to PCI, but is situated closer to the host's memory system and has asynchronous transfer characteristics (i.e., 'writes' to card memory are faster than 'reads'). While PCI devices can usually store data to an AGP card with write operations, most motherboard chipsets do not allow a PCI device to utilize read operations to fetch data from AGP card memory.

The Linux kernel provides a simple, universal driver that allows user applications to directly access a video display card's frame buffer. This driver provides a means for an application to map the frame buffer into the application's address space, where it can be updated using normal PIO operations. The device driver also provides basic functionality that allows applications to query the frame buffer's settings. User-space applications can use these calls to determine the dimensions of the display and the current pixel depth. The frame-buffer driver operates regardless of whether or not a window manager is running. Therefore, it is possible for user-space applications to update the graphical display even if X windows software [89] is not running.

6.5.2 GRIM Integration

It is generally infeasible to port communication endpoint software to a video display card due to the architecture of these cards. Video display cards simply render graphical data and therefore are designed to function as data sinks. Thus, the approach taken in GRIM to integrate a video display card into the cluster architecture is to simply present the video display card's frame buffer as a block of memory that cluster applications can update. The first task in accomplishing this goal was adapting GRIM to interact with the device driver that controls the frame buffer. At initialization time, GRIM opens the driver and maps the frame buffer into the application's address space. Next, GRIM utilizes a custom-built device driver to transform the virtual address mapping of the frame buffer into a physical address that can be referenced by the NI card. Finally, the physical address of the frame buffer is shared with other endpoints in the system. These endpoints can then use remote memory write operations with the physical address (RM-P) to render changes to the output display.

An example host-level application was constructed to demonstrate how the frame buffer could be used in the distributed cluster environment. In this application, multiple hosts are equipped with video capture cards and configured to capture live video streams. At runtime, each

of these hosts is instructed to transmit its captured video stream to a different area of one host's frame buffer. The result is that multiple video streams can be displayed simultaneously on a single output screen. The advantage of this approach is that the incoming video streams can be routed directly to the output display without buffering the data in the display host's memory. This type of operation can be beneficial in other tasks such as distributed rendering systems, where individual workstations perform the task of rendering different portions of an overall scene.

6.6 Summary

Multiple peripheral devices have been integrated into the GRIM communication environment. This work has been simplified by the fact that GRIM implements a common core of its communication functionality in the NI. Peripheral devices in this environment implement mechanisms to facilitate interactions with resources in the local host such as the NI or other endpoints. GRIM is a flexible substrate for this type of work because it can be adapted for use with peripheral devices that have a wide array of characteristics. The I_2O adaptor integration was the least challenging of these efforts because endpoint software could be ported directly to the card. The RC-1000 endpoint was the most challenging because endpoint software had to be implemented as FPGA circuitry. The video capture and display integration work complete the survey of peripheral device work because they represent devices that cannot natively run endpoint software, and therefore require host-level management. All of these examples illustrate that GRIM is extensible in that it can be adapted with device-specific functionality to allow new peripheral devices to be incorporated into the cluster environment.

CHAPTER VII

STREAMING COMPUTATIONS

Resource-rich cluster computers feature a large number of host CPUs and peripheral devices that can be used by applications as a pool of available resources. A challenge in working with such resource models is constructing applications that effectively utilize the cluster's distributed resources. This chapter describes support for a pipelined computing model where accelerators available as peripherals in distinct nodes can be configured through GRIM to operate as a single computational pipeline. Such a model can support a wide range of applications, including streaming media and signal processing applications.

Adapting a message layer to support streaming computations requires an examination of how cluster resources can be utilized as elements in a computational pipeline. A streaming computation is visualized as a connection-oriented service, where a number of operations are performed on data that is passed through a connection. This programming abstraction requires two specific features from an implementation. First, individual resources in a connection must be capable of performing a specified computation on an incoming data stream. In GRIM, this functionality can be accomplished through the use of GRIM's built-in active message mechanisms. Second, a resource in a connection must be equipped with mechanisms for forwarding computational results to the next resource in the connection. This functionality is implemented in GRIM through both the library's native reliable delivery mechanisms and a programmable *forwarding directory*. This directory allows users to configure the exact functionality of a streaming operation in flexible manner.

As a motivating example, the Celoxica RC-1000 FPGA endpoint has been adapted to support streaming computations. In addition to equipping the RC-1000 endpoint with a

forwarding directory, several enhancements were made to the endpoint's architecture. These modifications include a virtual memory system that allows on-card memory to be shared by applications and a system for dynamically reconfiguring the FPGA with hardware circuits needed at runtime by applications. This chapter provides implementation details of the streaming computation extensions, as well as performance measurements of the RC-1000 endpoint that relate to the streaming environment.

7.1 An Overview of Streaming Computations

In pipelined implementations, a complex computational task is divided into a linear series of subtasks that can be performed by individual resources. Each resource is then configured to function as a pipeline stage, performing a specified computation on incoming data and forwarding the results to the next resource in the pipeline. The benefit of this approach is that when streams of data are injected into the pipeline, it is possible for the pipeline stages to concurrently operate on different portions of the stream. The desired result is for the system to be capable of producing output results at the same rate that data is injected into the pipeline.

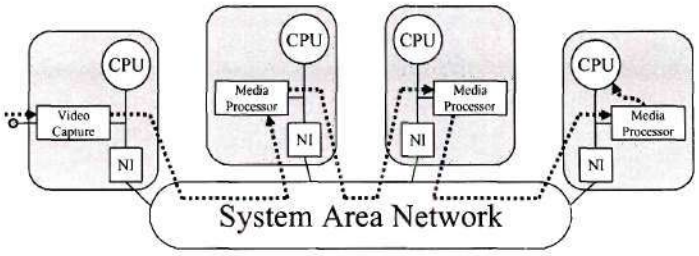


Figure 7.1: A streaming computation example.

Multimedia applications provide a strong motivation for developing systems that are capable of performing high-throughput streaming computations. In a number of these applications, raw multimedia data streams must be processed in real time. Unfortunately, it is often infeasible to use a single host computer to perform this processing because of the high data

rates that are involved and the computational complexity of the operations that need to be performed. Therefore, it is beneficial if a series of resources in a cluster can be utilized as a streaming computational pipeline. An example of such a pipeline is depicted in Figure 7.1. In this system, a video capture card generates a video stream that is relayed through multiple peripheral devices distributed throughout a cluster. The devices perform specific operations on the data stream until the data is properly prepared for consumption by a host-level application.

7.1.1 Connection-Oriented Streaming Computations

While streaming computations can be implemented in a variety of manners, a particularly useful abstraction is to visualize a streaming computation as a form of a connection-oriented service. In this abstraction, data injected into a connection is processed by a series of computational stages that are defined when the connection is established. As Figure 7.2 illustrates, any endpoint in the cluster can inject data into a connection, but computational results are only transmitted to a single endpoint. A new connection can be created by any endpoint in the system. After obtaining a unique identifier for a new connection, an endpoint must configure the individual resources that are to be used in the connection. Configuration information specifies the operation a resource should perform as well as where the results of the operation should be transmitted in the cluster.

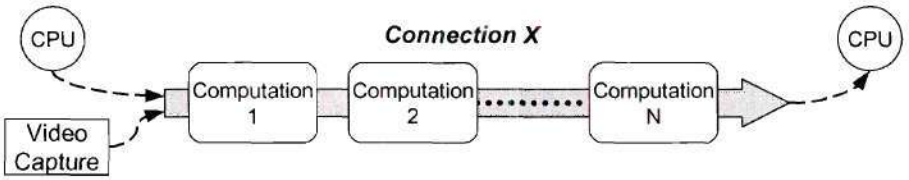


Figure 7.2: An example of a connection-oriented streaming computational pipeline.

There are multiple benefits to implementing a streaming computation as a connection-oriented service. First, connection-oriented communication is well understood by programmers

and is therefore a programming abstraction that can be adopted without much difficulty. Second, this approach can be used in a flexible manner to implement a variety of useful computational systems. For example, users can chain multiple connections together by forwarding the results of one connection to another. Therefore, users can construct complex operations by using a set of basic operations as building blocks. Finally, computational connections provide a simple programming abstraction that allows any endpoint to invoke complex operations without having to know the underlying mechanics of the connection. This feature is especially valuable for simple peripheral device endpoints in resource-rich clusters.

7.1.2 An FPGA-Based Pipeline Unit

GRIM has been extended via software to support a connection-oriented form of streaming computations. In this effort, peripheral devices can be configured to function as the pipeline stages of a streaming computation. Of the peripheral devices that are currently supported in GRIM, the most attractive device for this work is the Celoxica RC-1000 FPGA card discussed in the previous chapter. This card is a natural candidate for use in streaming operations because it is designed to function as a computational accelerator. Therefore, the current RC-1000 endpoint implementation has been modified to support streaming computations. These modifications are implemented as extensions to the FPGA's frame, which is the block of logic that implements the endpoint state machines for the RC-1000. While the focus of this chapter is on implementation details for adapting the RC-1000 endpoint for streaming computations, other endpoints can be extended with this functionality in a similar manner.

7.2 Pipeline Computations

The first of two functional requirements for an endpoint to behave as a pipeline stage is for the endpoint to be capable of performing a predefined computation on incoming messages for a data stream. This functionality can be implemented in a relatively straightforward manner using

GRIM's active message programming interface. For the RC-1000 FPGA endpoint, the active message function handler is used to select the computational circuit that processes an incoming message for a data stream. Observing that FPGAs have a limited capacity for housing computational circuits, the RC-1000 FPGA endpoint has been extended with software that allows the FPGA to be dynamically reconfigured with different circuitry as needed by applications. The FPGA frame in this approach detects when it does not have the circuitry necessary to process a message and signals a *function fault* to the host. The host software is designed to resolve these faults, allowing the FPGA to be reconfigured on demand as needed.

7.2.1 The Use of Active Messages to Control Pipeline Computations

An endpoint that functions as a pipeline stage in a streaming computation must be configured to perform a user-specified operation on a data stream's incoming messages. This functionality can be accomplished through GRIM's active message programming interface. In this approach, a message arriving at a pipeline stage is labeled with a stream identifier and an active message handler that specifies the operation the endpoint should perform on the message. Since all pipeline processing instructions are included in an incoming message, it is necessary for the endpoint transmitting the message to format the message. While it may seem counterintuitive to have to place an endpoint's processing instructions at the preceding endpoint in the pipeline, doing so simplifies the configuration process. In this system, forwarding information (used to transmit results to the next pipeline stage) and processing instructions (used to specify the operation the next pipeline stage performs) are stored at the same location (the preceding pipeline stage).

Most endpoints can be easily adapted to perform the computational part of streaming operations, because this approach relies on the existing active message infrastructure. Similar to other messages, endpoints simply process streaming-computation operations by executing the proper active message function handler. For the RC-1000 FPGA endpoint, the FPGA frame's

current active message interface is sufficient for implementing this functionality. Messages arriving at the RC-1000 endpoint for a streaming computation are examined by the FPGA frame and processed using the user-defined circuit that matches the arguments specified in the message's header.

7.2.2 Dynamic FPGA Circuit Management

One of the difficulties involved in utilizing an FPGA as a computational resource is that each FPGA is only capable of housing a limited amount of user-defined circuitry. While the industry is constantly increasing the gate capacity of commercial FPGAs, it is unlikely that a single FPGA will ever be able to house all of the computational circuits that could be utilized by end applications. This limitation becomes a significant issue as the number of streaming computational pipelines used in a cluster increases. If these pipelines require diverse types of processing, it is likely that the number of computational circuits needed by the pipelines may outnumber the total space available for housing the circuits in the cluster's FPGA resources. What is needed is a system that can dynamically reconfigure the cluster's FPGAs to emulate the hardware operations that are needed by applications at runtime.

Modern commercial FPGAs generally provide two forms of reconfiguration that can be utilized by software that dynamically manages an FPGA endpoint's circuits. First, all FPGAs support a form of *full reconfiguration*, where an FPGA is reprogrammed in its entirety. Circuit-management software can utilize this operation to reprogram an FPGA at runtime with a configuration that contains a circuit that is required by an application. In this approach, multiple FPGA configurations are generated offline and stored in a database managed by the software. Second, some FPGAs support *partial reconfiguration*, where a region of the FPGA can be reprogrammed without affecting the rest of the chip. With this option, circuit-management software can be designed to replace one computational circuit for another. Unfortunately, partial reconfiguration operations can incur significant overheads due to the amount of effort that is

required in rerouting an FPGA's active signals. While the FPGA management techniques described in this section can be applied to both forms of reconfiguration, the focus of this effort is on utilizing full reconfiguration mechanisms.

7.2.3 Function Faults in the FPGA Frame

In order to support dynamic circuit management the RC-1000 FPGA endpoint had to be modified with functionality for assisting the reconfiguration process. These extensions allow the FPGA frame to detect the need for reconfiguration, and provide a means for the FPGA to save and restore its runtime state information during the reconfiguration process. The extensions operate as follows. Whenever the host system loads new computational circuits into the FPGA, it stores a list of function identifiers for the circuits in the FPGA card's SRAM. After the host activates the FPGA, the frame pulls these identifiers and other runtime state information into the FPGA. The frame uses this information at runtime to determine if an incoming message can be processed by the circuits that are available in the FPGA. If the FPGA is not equipped with the proper circuits, it initiates a *function fault* that must be resolved by the host's dynamic circuit management software.

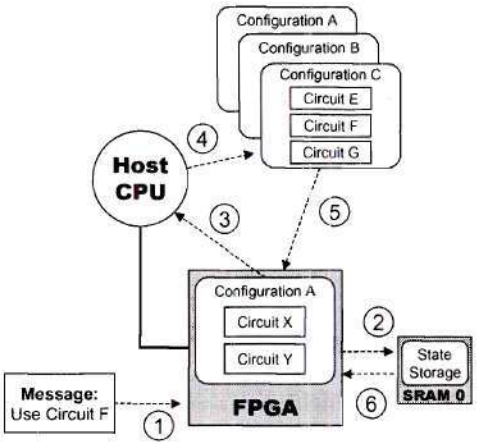


Figure 7.3: The process of reconfiguring an FPGA during a function fault.

Figure 7.3 depicts the steps that are taken during a function fault. After (1) the FPGA detects that it is not equipped with circuitry to process a message it (2) stores its dynamic state information in on-card SRAM. The FPGA then suspends its execution and (3) sends an interrupt request to the host processor. The host software detects the fault, determines which function is needed by the FPGA, and (4) retrieves an appropriate configuration from a database. The host (5) loads the FPGA with this configuration and updates the FPGA's list of available circuits. The host then restarts the FPGA which (6) fetches its dynamic state information from SRAM. The FPGA uses this information to begin processing the message that caused the function fault. The message can now be processed because the FPGA is loaded with the computational circuit that is needed by the message.

7.2.4 Function Fault Overhead

Measurements were performed to estimate the amount of overhead that is involved in processing an FPGA function fault. For the FPGA portion of this overhead, the FPGA frame's state machines were examined to determine how many FPGA clock periods are required by the frame to generate and recover from a function fault. Clock periods can be related to wall-clock time by dividing the number of clock periods by the FPGA's clock frequency (20 MHz). For the host's portion of a function fault's overhead, instrumentation software was added to the host library to measure the amount of time required to perform fault-resolution operations. A P3-550 MHz host was used in these measurements.

Table 7.1: The amount of time required to manage an FPGA function fault.

Resource	Action	FPGA Clocks	Time (µs)
FPGA	Store queue pointers	6	0.30
	Store missing function	2	0.10
	Trigger function fault	1	0.05
	Release SRAM bank 0	1	0.05
Host	Acquire SRAM bank 0	-	13
	Process fault		8
	Load configuration from file (optional)		10,205
	Reconfigure FPGA		95,114
	Set FPGA clock		2,405
	Set function IDs		2
	Reset FPGA		56,813
	Release SRAM bank 0		7
FPGA	Acquire SRAM bank 0	8	0.40
	Reload queue pointers	4	0.20
	Reload functions IDs	9	0.45

The results of the measurements are presented in Table 7.1. While the FPGA operates at a relatively slow clock rate, it is able to perform all of its function-fault operations in only a few microseconds. Unfortunately, there are significant overheads for the host to resolve a fault. The two dominant operations in this procedure are for the host to reconfigure and then reset the FPGA. The reconfiguration process is time consuming because approximately 700 KB of information must be serially loaded into the FPGA using PIO operations. Resetting the FPGA is time consuming because the operation requires a 50 ms delay for proper execution. Newer FPGA cards will reduce this overhead by a factor of 5-10, with custom architectures performing even better. However, the current model is on par with connection-oriented programming models where pipelines are constructed and changed infrequently.

7.3 Pipeline Forwarding

The second operation that a pipeline stage must perform is forwarding computational results to the next resource in the pipeline. This task is an integral part of a streaming computation because it allows a collection of distributed resources to be utilized in a connection. At a

fundamental level, forwarding mechanisms should allow pipelines to be constructed in a flexible manner. In addition to routing messages between resources, it should also be possible for users to route data through the same resource multiple times. The benefit of using the same resource to implement multiple pipeline stages is that dynamic application data can be more readily shared among the pipeline stages. One means of constructing a flexible system for managing the transfer of data between pipeline stages is to employ a forwarding directory at each endpoint in the pipeline. A forwarding directory is a user-programmable table that contains information that specifies how a pipeline stage should transmit the results of a streaming computation to the next stage in the pipeline. These tables are easily updated and serve as a simple means by which users can configure both the routing and computational operators used in a streaming computation.

7.3.1 Forwarding

After a pipeline stage generates computational results for an incoming message, it is necessary to forward the results to the next stage in the pipeline. Forwarding mechanisms must be flexible enough to be utilized in a number of manners. Figure 7.4(a-c) illustrates three fundamental examples of how data may be forwarded between resources in a pipeline. In the first example (a), a resource is configured to function as a single stage in a pipeline. Results from this operation are forwarded to another resource in the cluster. It is expected that most applications will utilize resources in this manner because the approach is the most straightforward to manage and implement. The second example (b) illustrates a more elaborate case where a resource is utilized to perform two sequential operations in a computational pipeline. This approach requires a means of buffering results at a resource and is beneficial for applications where data locality can be exploited. In the final example (c), a resource is utilized to process data for multiple independent streaming computations. This approach allows a resource to be utilized by multiple applications and requires mechanisms for isolating data streams.

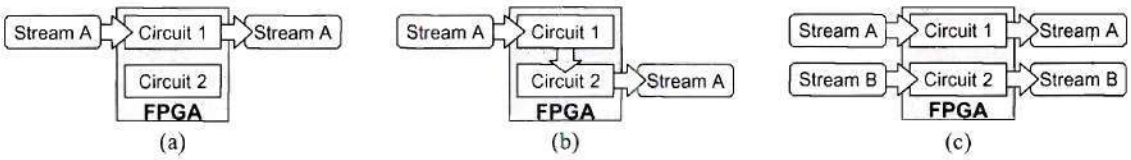


Figure 7.4: Forwarding examples for (a) a single computation on a single stream, (b) multiple computations on a single stream, and (c) multiple streams.

7.3.2 Forwarding Directory

One method by which diverse forwarding operations can be implemented in a streaming environment is to store forwarding information at the resources utilized in a connection. In this approach, each endpoint is equipped with a *forwarding directory* that contains information specifying where and how the endpoint should transmit the results of a streaming computation operation. A message arriving at an endpoint contains information that identifies the message as belonging to a particular computational stream. This stream identifier is used to extract information from the forwarding directory that specifies how the computational results of the operation should be formatted for transmission in the network. Therefore, users can construct new connections or modify the flow of existing pipelines simply by updating the appropriate forwarding directory entries of the resources that are involved. Updates can be performed using a built-in set of active message handlers that modify forwarding directory entries.

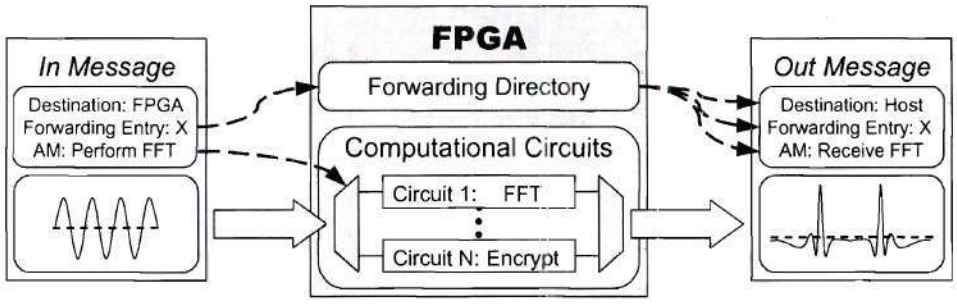


Figure 7.5: The forwarding directory provides information for transmitting a pipeline stage's results to another endpoint.

Figure 7.5 illustrates how a forwarding directory at an FPGA endpoint can be utilized as a means of forwarding data from one pipeline stage to another. In this example, an active message

arriving at the FPGA contains information specifying that the message belongs to computational stream X and requires processing by the FFT active message function handler. After decoding the message's header, the FPGA utilizes an FFT computational circuit to process the payload section of the incoming active message. The results of this computation are stored in the payload section of an outgoing active message. The header for this message is supplied from entry X of the forwarding directory. This header specifies where the communication library should transmit the message as well as the operation that should be performed at the next pipeline stage.

7.3.3 FPGA Implementation

The RC-1000 endpoint's frame was modified to support a forwarding directory. The directory consists of 256 entries that are stored in the first SRAM memory bank of the card. An entry in the forwarding table is comprised of eight 32-bit words that house all of the values necessary for generating the header of an outgoing message. The frame provides a special active message function handler that allows users to program individual entries of the forwarding table. When the FPGA frame detects the arrival of a new message, it examines the message's header to establish the necessary data paths between resources in the FPGA. Users can store the results of an active message operation in on-card scratchpad memory, a recycling buffer, or in the outgoing message queue. The recycling buffer allows the message generated by one FPGA computation to be routed back to the input of the FPGA endpoint. In this manner, a single FPGA can be configured to implement multiple pipeline stages for a computational stream. Forwarding directory performance is included in Section 7.5 as part of the overall performance of the RC-1000 FPGA endpoint when it is used for streaming operations.

7.4 Managing Pipeline State Information

In many streaming applications, it is beneficial if application data can be stored at the individual resources utilized in the computational pipeline. This data can include dynamic state

information or static arguments such as filter parameters that are used to process incoming messages. Unfortunately, peripheral device endpoints have a limited amount of on-card memory for housing application data. As the number of streaming computations using a resource increases, the amount of on-card memory available to each application decreases. It is beneficial to consider mechanisms that allow peripheral device memory to be shared in a more flexible manner. For the RC-1000 endpoint, a basic virtual memory system has been constructed that allows the card's scratchpad memory to be treated as a paged resource. Scratchpad pages are dynamically swapped with host memory as needed by applications. These mechanisms provide a basic form of protection for applications sharing the FPGA card and allow the endpoint to be transparently utilized by different applications. Similar mechanisms can be implemented for other peripheral device endpoints.

7.4.1 Managing On-Card Memory for an Endpoint

Peripheral device endpoints have a limited amount of on-card memory that can be utilized for housing application data. This memory is valuable to application designers because it allows application data to be stored at the endpoint. In the case where multiple applications utilize the same endpoint, it is necessary to provide some form of management for on-card memory to prevent conflicts between applications.

The simplest approach is to utilize an allocation scheme where each application obtains a block of on-card memory that is exclusively owned by the application. While this method may be suitable for some endpoints and applications, it has three major drawbacks. First, as the number of applications utilizing an endpoint increases, the amount of available on-card memory for each application decreases. Second, applications must be designed to work in a cooperative manner with the memory system. Depending on how memory is allocated, this approach may make it more challenging for applications designers to work with peripheral devices. Finally, this system

provides no protection between applications. Therefore, an application can erroneously overwrite another application's data.

A better approach to managing on-card memory is to implement a virtual memory system for the endpoint. In this approach, card memory is divided into page frames, and applications reference on-card memory with virtual addresses. Before the endpoint begins processing a message, it determines if the message's memory references can be satisfied with the pages that are currently loaded in the card's page frames. If a page is not loaded, the endpoint must replace the current page with the requested data. Unloaded pages can be stored anywhere in the system, although the most practical location is host memory. While page faults for on-card memory can incur substantial overheads, implementing a virtual memory system for a peripheral device provides basic protection for applications that share the device.

7.4.2 Virtual Memory for the RC-1000 FPGA Endpoint

A basic virtual memory system has been constructed for the RC-1000 FPGA card's scratchpad memory. This system operates on a coarse granularity with a virtual memory page being defined as a 2 MB block of SRAM. SRAM memory banks 1 and 2 of the RC-1000 are therefore used as page frames for housing virtual memory pages that can be accessed by user-defined circuits. Incoming messages that utilize scratchpad memory reference data with a virtual memory address. This address is comprised of a page identifier and an offset into the page. Before the frame begins processing a message, it examines the page identifiers of the virtual memory addresses supplied in the message to determine if the page is currently loaded in one of the two page frames. If a message's pages are loaded, the frame establishes the necessary data paths for the computational circuits to access the memory. The offset value of the virtual address is used as the starting address within the page for accessing data.

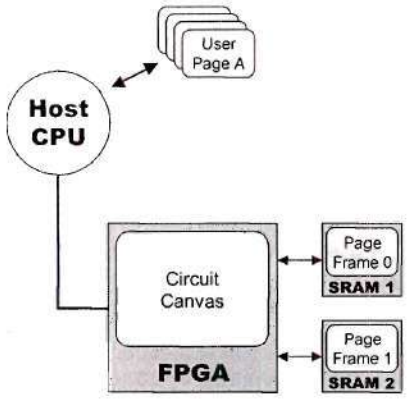


Figure 7.6: A virtual memory system is implemented for on-card SRAM. SRAM banks 1 and 2 serve as page frames for an application's scratchpad data. Unloaded pages are swapped into host memory.

If a requested page is not loaded in one of the page frames, the FPGA frame must invoke mechanisms for loading the proper data into card memory. Figure 7.6 illustrates the organization of the memory system used in this procedure. First, the FPGA frame stores the missing page identifier in SRAM. It then suspends the FPGA's execution and sends the host an *FPGA page fault* signal. The host receives this signal, determines which page frame needs to be updated, and then performs the necessary page swap. A page swap involves transferring the page frame's current data to a buffer in host memory and then transferring the desired page from host memory to the card. After a swap, the host updates the FPGA's list of loaded page identifiers and restarts the FPGA. The restarted FPGA loads the new page identifiers and continues processing the message that originally caused the fault. From the user's perspective, these operations take place automatically in a transparent manner.

7.4.3 Page Fault Performance

The main drawback to implementing a virtual memory system for a peripheral device endpoint is that there can be significant overheads in resolving page faults. In addition to using the host CPU to resolve a fault, large blocks of data must be transferred to and from host memory.

Performance of the RC-1000 endpoint software was characterized to determine how much overhead is involved in a page fault.

Table 7.2: Overhead for managing an FPGA page fault.

Resource	Operation	FPGA Clocks	Time (µs)
FPGA	Detect fault	1	0.05
	Store page identifiers	3	0.15
	Issue fault signal	1	0.05
	Release SRAM banks 0-2	1	0.05
Host	Acquire SRAM banks 0-2	-	13
	Process fault		8
	Unload 2 MB page		43,494
	Load 2 MB page		17,927
	Notify FPGA		1
	Release SRAM banks 0-2		7
FPGA	Acquire SRAM banks 0-2	8	0.40
	Reload page identifiers	2	0.10

The results of the page fault measurements are listed in Table 7.2. As these tests reveal, the most time-consuming portion of this procedure is the transfer of scratchpad-memory pages between the card and host memory. The differences between loading and unloading a page are due to the fact that in the current implementation the load operation is performed by a zero-copy DMA, while the unload operation is performed by a one-copy DMA. Based on these measurements, page faults are expensive operations in this implementation. In order to reduce the number of page faults that take place at run time, users should implement exclusive ownership mechanisms for the card that guarantee that only one application will utilize the RC-1000 endpoint for a period of time.

There are several means by which the virtual memory system could be improved for this card. First, the page size could be reduced in order to allow multiple pages to be stored in the scratchpad memory banks. This approach allows the data sets of multiple applications to be concurrently loaded in card memory, thereby reducing the frequency of page faults. Another interesting approach is to physically attach and utilize a storage device to the FPGA card for housing unloaded pages. The RC-1000 card features a large number of I/O pins that can be

utilized to attach a hard drive or other storage devices such as flash memory. A disk controller can be constructed in the FPGA for managing disk interactions. Therefore, page faults could be managed entirely by the card, swapping card memory to disk without the intervention of the host. The downside of this system is that it is challenging to implement, and adds to the overall complexity of the FPGA frame.

7.5 Performance of an FPGA as a Pipeline Stage

Timing measurements were performed to determine how much overhead is involved when the RC-1000 FPGA is utilized as a pipeline stage in a streaming computation. In these experiments, message data arrives at the RC-1000 endpoint from either the host endpoint or the NI card. Messages contain 4 KB of payload data (i.e., 1024 words of 32b data) and specify a pass operation for the active message handler. This operation simply transfers the incoming payload data to the outgoing message's payload. FPGA clock times are extracted directly from the state machines and related to wall-clock time by dividing clock periods by the FPGA clock speed (20 MHz).

Table 7.3: RC-1000 overhead involved in processing a 4 KB message.

Resource	Operation	FPGA Clocks	Time (µs)
(Host / NI)	Acquire SRAM bank 0	-	(13 / 5.5)
	Inject 4 KB message		(107 / 32)
	Release SRAM 0		(7 / 3)
FPGA	Acquire SRAM banks 0,3	8	0.40
	Fetch incoming message header	7	0.35
	Fetch forwarding information	5	0.25
	Fetch payload data	1024	51.2
	Computation latency	1	0.05
	Store results	3072	153.6
	Store outgoing header	48	2.4
	Update message queue pointers	3	0.15
	Release SRAM banks 0, 3	1	0.05
Host	Acquire SRAM bank 3	-	13
	Perform DMA		69
	Release SRAM bank 3		7

The results of the measurements are listed in Table 7.3. Starting with the resources that inject the message into the RC-1000 endpoint, it is clear that the NI can insert messages into the RC-1000 more efficiently than the host endpoint. This is because the NI controls the RC-1000's memory arbitration mechanisms and the NI has better control over its injection mechanisms because it directly manipulates a PCI DMA engine.

For the FPGA endpoint, the majority of the overhead in processing the message comes from streaming the individual data values through a computational unit. In this system, the fetch, compute, and store operations take place in a pipelined fashion, allowing the operations to overlap. This feature demonstrates how an FPGA can be beneficial for processing data because it illustrates how custom pipelines can be constructed in the hardware to achieve high throughputs. It is important to note that the store operations require 3 clock cycles in the current implementation, as opposed to reads which can fetch a new data value every clock period. After processing a message, the FPGA must format the outgoing message with a header obtained from the forwarding directory. Control is then passed to the host system, which detects the message and initiates the DMA that transfers the message to either the NI or another endpoint in the local host on behalf of the RC-1000 endpoint.

7.6 Summary

Streaming computations are a means of utilizing a collection of distributed resources to improve the throughput of a complex operation. In this effort, a series of cluster resources are utilized to implement a computational pipeline. While the cluster resources function as the computational stages in the pipeline, the message layer provides the framework for delivering data between the pipeline stages. Each resource in a pipeline is equipped with a forwarding directory that allows users to specify how data flows through the pipeline, and the operations that are performed on the data streams. As a means of investigating implementation details, the RC-1000 FPGA endpoint has been extended to support streaming computations. Additional

enhancements were made to the endpoint to allow multiple applications to utilize the resource at the same time.

CHAPTER VIII

MESSAGE LAYER EXTENSIONS

In addition to hardware extensibility, message layers for resource-rich cluster computers must also be designed to be support user-defined software extensions. These extensions allow users to customize their interactions with the message layer in order to implement functionality needed by applications. In GRIM, users can easily add new functionality to the core communication library at different levels of the message layer. At the network level, users can define new communication operations (e.g., multicast) by extending the message layer's NI firmware. At the endpoint level, users can implement extensions in a straightforward manner by constructing specialized active message function handlers.

This chapter deals with the issue of message layer extensibility in the context of application-related software extensions. Three specific software extensions have been constructed for GRIM to illustrate how new functionality can be easily incorporated into the library. First, multicast mechanisms have been added to the core library to allow users to easily transmit the same message to multiple receivers. These mechanisms result in a reduction in transmission overhead for the sender because the message is replicated in the network. These extensions demonstrate how users can insert new functionality at the NI level of the library. Second, fragmentation and reassembly mechanisms have been constructed for the active message and remote memory programming interfaces, as well as for multicast operations. These extensions illustrate how endpoint-level software can be added to the library to provide increased end-to-end performance. Finally, a reliable sockets emulation has been implemented using the active

message programming interface. This emulation allows GRIM to be used as a replacement for the sockets library in legacy applications.

8.1 Multicast

A common operation utilized in parallel processing applications is multicast. Multicast is a form of communication where an endpoint transmits the same message to multiple receivers in the cluster. A subset of multicast is broadcast, where the list of receivers includes all endpoints in the cluster. Multicast messages are often used to distribute state information or provide synchronization among a number of cluster endpoints. In [90], researchers observed that these types of interactions have a strong impact on the overall performance of parallel processing applications. Therefore, it is worthwhile to investigate means by which multicast can be performed efficiently in a communication library for resource-rich clusters.

While multicast operations can be implemented simply by layering this functionality on top of existing unicast mechanisms, doing so results in limited performance as the number of endpoints in a multicast distribution grows. Therefore, it is beneficial to examine how the low-level mechanics of a communication library can be extended to support multicast more efficiently. Communication libraries supporting multicast typically perform the task of replicating multicast messages for a distribution tree in the NI [91]. Because these approaches recycle an incoming message back into the network, it is necessary for multicast mechanisms to be designed in a manner that prevents deadlock. The GRIM communication library has been extended with multicast support. In order to avoid deadlock GRIM employs an ordering scheme on multicast trees that prevents cyclic dependencies between NI cards. In addition to preventing deadlock, this scheme utilizes a single message queue for recycled messages as opposed to two or more, resulting in an increased utilization of NI buffer space for multicast operations.

8.1.1 Multicast through NI Recycling

Multicast can be implemented on top of any unicast communication library simply by constructing endpoint software that transmits a separate copy of a multicast message to each receiving endpoint in a multicast distribution. An example of this approach is illustrated in Figure 8.1(a). While trivial to implement, this approach suffers from several drawbacks. First, this approach requires an endpoint to inject multiple copies of a message into the NI. As discussed in Chapter 5 the endpoint-to-NI transfers are the slowest part of the end-to-end communication pipeline. Therefore, significant overheads may be accumulated by the endpoint for injecting multiple copies of the same message into the NI. Second, all transmissions for a multicast message must be serially transmitted through the sending endpoint's NI. Because the NI has limited amounts of buffer space for housing in-transit messages, it is likely that large multicast transmission will saturate the sending NI and delay message delivery. Finally, if endpoints are responsible for replicating a multicast message then every endpoint must be equipped with an up-to-date list of multicast receivers. This requirement makes updating a multicast distribution's membership list an expensive operation.

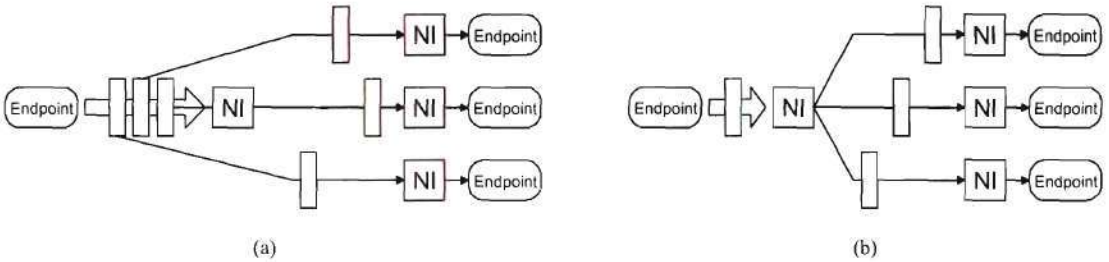


Figure 8.1: Replicating a multicast message can be performed by (a) the sending endpoint or (b) in the NI.

An alternative approach is to perform the task of replicating multicast messages in the communication library at the NI level. As illustrated in Figure 8.1(b), this approach is advantageous because a multicast message is only transferred once from the sending endpoint

into the NI. This optimization reduces the load of the host's I/O system and greatly simplifies the amount of work an endpoint must perform to transmit a multicast message. In the context of resource-rich clusters, this approach is also beneficial because it is possible for all endpoints in the host to utilize multicast mechanisms because message replication is deferred to the NI.

Moving the task of replicating multicast messages into the NI requires consideration of how the NI should perform the task. Simply using the sending NI to transmit the multicast message to all receivers results in similar issues to the endpoint-based replication scheme: all multicast messages are serially transmitted by the sender and each NI must have knowledge of the entire list of multicast receivers. Therefore, most NI-based multicast implementations are based on a distributed approach where the task of replicating messages is divided among the NIs in the multicast group.

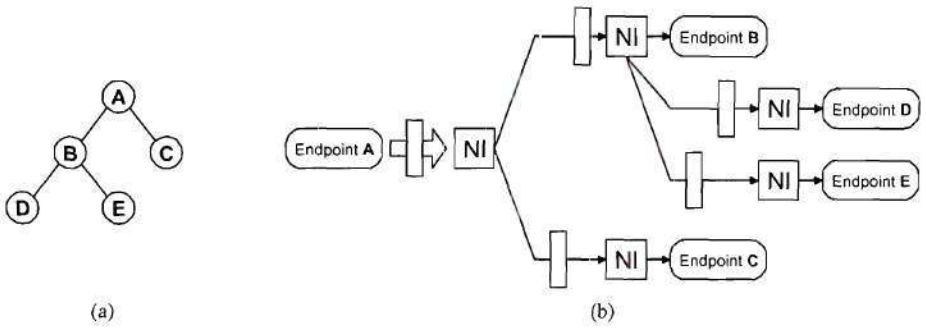


Figure 8.2: The task of replicating messages can be distributed among NIs through (a) constructing a distribution tree and (b) performing a limited number of multicast injections at each NI.

One approach to distributing the task of replicating multicast messages in the NIs is to organize the multicast group into a tree structure and then perform message replication in the individual NIs. An example of this approach is presented in Figure 8.2(a-b) with a five endpoint tree (a) that results in three NIs transmitting multicast messages to other NIs in the network (b). There are multiple advantages to distributing the task of message replication among NIs in the multicast group. First, multicast distribution can be accelerated because it is possible for multiple

NIs to concurrently work on replicating a multicast message. Second, the workload for distributing messages is shared among all nodes in the group compared to requiring the sending NI to perform all of the work. Finally, individual NIs do not need to have knowledge of the entire multicast tree. Instead each node only needs to be equipped with the IDs of its children and the root of the tree. This approach is sometimes referred to as recycling [92], as multicast messages are recycled back into the network during the distribution process.

8.1.2 Deadlock Issues in NI-Recycling Multicast

A hazard of using NI-recycling to perform multicast message distribution is that without precautions, it is possible for the network to become deadlocked. As illustrated in Figure 8.3(a), a NI that performs recycling takes an incoming multicast message and injects multiple copies of the message back into the network. Therefore, if two or more NIs perform multicast recycling at the same time, it is possible for a cyclic dependency to be formed to between the NIs that can result in deadlock if the network becomes congested. An example of this type of condition is depicted in Figure 8.3(b-c). In this example, two multicast trees have nodes 2 and 4 in common (b). Unfortunately, these trees distribute messages in the reverse order, which leads to a cyclic loop between the two nodes (c). If the network becomes congested it is possible for the messages from these trees to reach a deadlock condition where incoming messages cannot be accepted because outgoing messages cannot be transmitted successfully to their destinations.

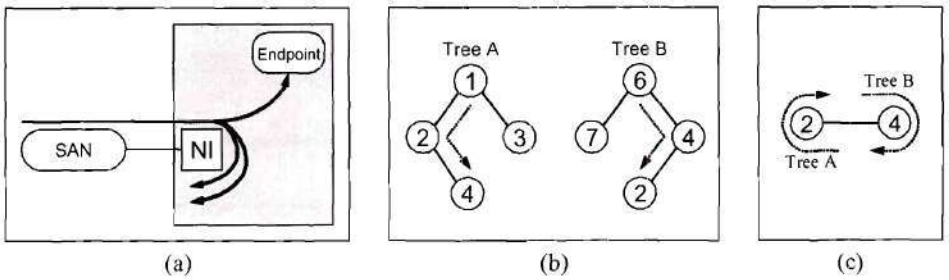


Figure 8.3: Replicating multicast messages in the NI results in a turn that could lead to a cyclic dependency loop.

A common means of preventing this form of deadlock is to utilize the up*/down* routing algorithm first described for the Autonet network [93]. Up*/down* routing works with irregular topologies and is deadlock free [94]. In this approach, a spanning tree graph is constructed for the cluster with one node serving as the graph's root. Links in the graph are labeled with directions so that a node's "up" direction is towards the root and there are no directed loops in the graph (i.e., traversing the graph in the up direction never leads to a previously visited node). Nodes have two separate buffers for outgoing messages: one for upward-bound outgoing messages and another for downward-bound outgoing messages. At each node an incoming message can be ejected from the network, transferred from an incoming down link to an outgoing down buffer, or transferred from an incoming up link to either an up or down outgoing buffer. By preventing messages from traveling from a down link to an up link, cyclic dependencies are removed from the channel dependency graph resulting in deadlock freedom.

Up*/down* routing can be applied to prevent NI-recycling multicast mechanisms from reaching deadlock [95]. In this effort, a directed acyclic graph is defined for all of the NIs in the cluster. This graph is fully connected because point-to-point networks allow any NI to directly communicate with any other NI. It is required that all multicast distribution trees be mapped onto the directed acyclic graph for the cluster. Each NI is equipped with separate up and down labeled logical channels for housing multicast messages that are recycled into the network. When a multicast message arrives at a NI from the network, the NI determines which multicast logical channel to recycle the message into based on the current direction of the message and the rules of up*/down* routing. Through these conditions cyclic dependencies between multicast buffers at different NIs are broken, allowing multicast transfers to take place without deadlock.

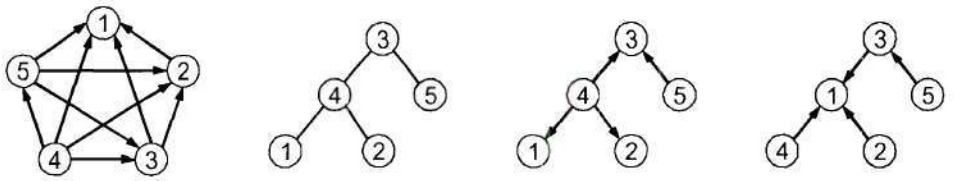


Figure 8.4: (a) A directed acyclic graph for a cluster's multicast transmissions. (b) A desired multicast distribution tree. (c) The desired multicast tree when labeled with link directions. (d) A reordering of the multicast tree that does not violate the up*/down* routing rules.

One of the conditions of using up*/down* routing for multicast is that the multicast distribution trees are arranged in a manner that agrees with the cluster's directed graph and routing rules. Some multicast distribution trees would violate these conditions and must be rearranged in order to prevent deadlock. An example of such a situation is presented in Figure 8.4(a-d) for a five-node network that has the directed acyclic graph shown in (a). When the multicast distribution tree presented in (b) has its links labeled (c) using the cluster's directed acyclic graph (a), there is a routing problem at NI 4. In this tree NI 4 is unable to transmit messages from NI 1 to NIs 2 and 3 because doing so involves a transfer from a down-directed link to an up directed link. Therefore, the multicast delivery tree must be rearranged to a topology such as that presented in (d). This topology allows adheres to the routing rules and is guaranteed to not to create deadlock situations when used with other valid distribution trees.

8.1.3 Multicast with a Single Recycle Queue

A criticism of up*/down* routing is that it requires the use of two separate message queues or logical channels for housing outgoing messages. While dynamic buffer space issues can be addressed through techniques such as escape channel routing [96], the primary issue in GRIM is that increasing the number of logical channels in the NI reduces the overall performance of the NI. Therefore, it is beneficial to consider means by which the up*/down* routing algorithm can be applied in which multicast traffic can be housed in only a single outgoing logical channel.

Reducing the number of logical channels used for multicast traffic to a single logical channel can effectively be accomplished by making restrictions on the manner in which multicast distribution trees are arranged in the cluster. Given that a two-channel up*/down* routing scheme already requires some multicast trees to be reordered, these conditions do not significantly impact end users.

The approach taken in GRIM to achieve deadlock-free multicast transmissions is to use a two-channel up*/down* routing scheme with the restriction that multicast trees are arranged in a manner that multicast messages always flow in the down direction of the cluster's directed acyclic graph. Because this system prevents messages from flowing in the up direction, the up logical channel can be removed from the NIs. One implementation of these conditions that is used in GRIM is as follows. A directed acyclic graph is constructed for the NIs in the cluster, where each NI has an up-directed link to every NI that has a smaller identification number in the cluster. An example of such a graph is presented in Figure 8.5. When multicast trees are constructed for the cluster, they are ordered in a manner such that a NI's ancestors in the tree are NIs with smaller id values and its descendants are NI's with larger id values. As a result multicast transmissions always propagate from a NI to a NI that has a larger id. Because data flowing from a NI with a smaller id to a NI with a larger id is always a down-traversal in the up*/down* graph, the up channel is not needed in this approach. The system however still operates under up*/down* routing rules and is therefore guaranteed to be deadlock free. Messages injected into a multicast tree must be transmitted to the root of the tree for transmission. However, since these messages are maintained in a separate logical channel at the sending NI, and incoming multicast messages cannot be recycled into this buffer, there is no dependency or possibility for deadlock.

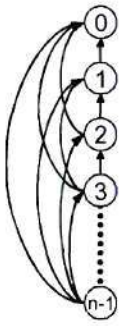


Figure 8.5: A directed acyclic graph for cluster nodes.

The single recycle queue approach has both positive and negative characteristics for multicast transmissions. As discussed earlier the primary benefit of this approach is that the sending NI needs to manage only a single message queue instead of two. Another benefit is that recycled messages are always placed in the same outgoing logical channel no matter where the destination is. In a two-channel approach it is possible that the NI would have to store the same multicast message in both channels if the next two receivers in the tree had different link directions. As for drawbacks, this approach creates hotspots in the network. NIs with lower IDs are more likely to be used for forwarding messages because they have more routing options than nodes with higher ids. Another negative aspect of this approach is that updating a multicast group is more complex in this approach because there are restrictions as to where a node can be placed in the distribution tree. Given that multicast tree updates are infrequent this factor is not a critical issue.

8.1.4 Implementation

The GRIM communication library has been extended to support a subscription-based form of multicast that is based on the preceding arguments. In this implementation the distribution of multicast messages to a group of subscribing endpoints is handled in the network by NIs that are equipped with a single, multicast recycling logical channel. The subscription nature of the implementation allows endpoints to dynamically join or leave a multicast group

without significant overhead. The NIs for a multicast group are arranged in a binary tree topology, with each NI being responsible for distributing an incoming multicast message to (1) all the subscribing endpoints in the local host as well as (2) up to two other NIs in the cluster. In order to prevent deadlock, multicast messages may only flow from a NI to another NI that has a larger id number.

Applications utilize the multicast facilities of GRIM through a library of function calls. Each multicast group is statically labeled with a string name and dynamically assigned a globally unique integer identifier. One node in the cluster manages a database for translating multicast string names into runtime integer IDs that can be referenced by all endpoints and NIs in the cluster. When an endpoint attempts to translate a string name that is not in the database, a new integer identifier is created and ownership of the multicast group is assigned to the endpoint requesting the translation. The owner of a multicast group serves as a central reference point for the multicast group and is responsible for dynamically managing the subscription list. Multicast management functions take place transparently in GRIM through the use of specially designed active messages. Therefore, an endpoint needing to communicate with a multicast group simply locates the multicast id for the group and injects a message that is marked with the id into the communication library. Multicast delivery is performed automatically by the communication library.

In order to perform the task of multicast distribution, individual NIs must be configured with two pieces of state information for each multicast tree. First, each NI is loaded with information that specifies the root NI for each multicast tree in the system. This information allows a message injected into the NI to be routed to the root of the multicast tree so that distribution can begin. If the originating NI is also the root of the multicast tree the message is simply moved from its outgoing logical channel into the NI's outgoing multicast logical channel when buffer space is available. The second piece of state data that NIs are loaded with is forwarding information. This information is used to determine which (if any) endpoints in the

local host require a copy of an incoming multicast message, and which (if any) NIs in the cluster should be forwarded a copy of the message. When a NI is required to forward a message to other NIs, it inserts the incoming message into the NI's outgoing multicast logical channel once for all intended destinations and marks the message with appropriate forwarding information.

8.1.5 Multicast Group Updates

In the subscription-based form of multicast, endpoints can join or leave a multicast group dynamically. This operation is performed by transmitting a subscription update request message to the host-level endpoint in the cluster that manages a specific tree. After processing this request the endpoint determines the new multicast tree that needs to be constructed to satisfy the subscription update. The endpoint then transmits a special tree update message to the root of the new tree that contains the complete list of NIs that are part of the new tree. Upon receiving this message a NI will update its own local forwarding tables and then insert the message into its multicast logical channel for forwarding to its new children. These in-band update messages allow the multicast tree to be updated without involving the subscribing endpoints.

One of the challenges in implementing a system where multicast trees can be updated dynamically is correctness. In the ideal case all multicast messages are distributed to all of the endpoints that were part of a multicast group when the message was initially injected into the distribution tree. The use of in-band updates partially upholds this characteristic because when a NI updates its forwarding tables, it does not modify multicast messages that are already waiting for transmission in the outgoing multicast queue. Therefore, forwarding changes are applied only to messages that follow the update message.

Another challenge in implementing multicast updates is preventing update messages from bypassing previously transmitted multicast messages for a tree. This condition is possible because update messages propagate through the cluster using the new multicast distribution tree instead of

the old tree. For example, if linear tree A-B-C-D-E is being updated to linear tree A-D-F, it is possible that the update message will arrive at node D while multicast data is still queued in nodes B and C. Therefore, GRIM implements a ticketing scheme that forces multicast message to be processed in the order they were injected. In this scheme the root NI labels each message with a ticket from a counter that is incremented after the transmission. NIs in the tree refuse to accept an incoming multicast message if its ticket value does not match the NI's expected value for the multicast tree. Therefore, this system places strict ordering on multicast messages that prevents multicast messages for a tree from bypassing each other.

8.1.6 Multicast Communication Path

As a first step in evaluating the performance of GRIM's multicast mechanisms, it is beneficial to examine the low-level details of the multicast communication path. An example of how data flows through the network components for a multicast tree is depicted in Figure 8.6. In this example, endpoint A injects a new multicast message into its outgoing NI message queue. The NI is the root of the multicast tree and therefore after the NI detects the message it transfers it to the outgoing multicast queue. The NI then transmits the message to two other NIs where the message is relayed to endpoints B and C. While node C is a leaf in the distribution tree, node B must forward the message to node D. Therefore, node B's NI copies the message into its outgoing multicast queue and transmits the message when the outgoing link becomes available. Node D receives the message and transfers it to its endpoint, completing the multicast operation. It is important to note that all network transactions in this process take place using per-hop reliable transmission mechanisms.

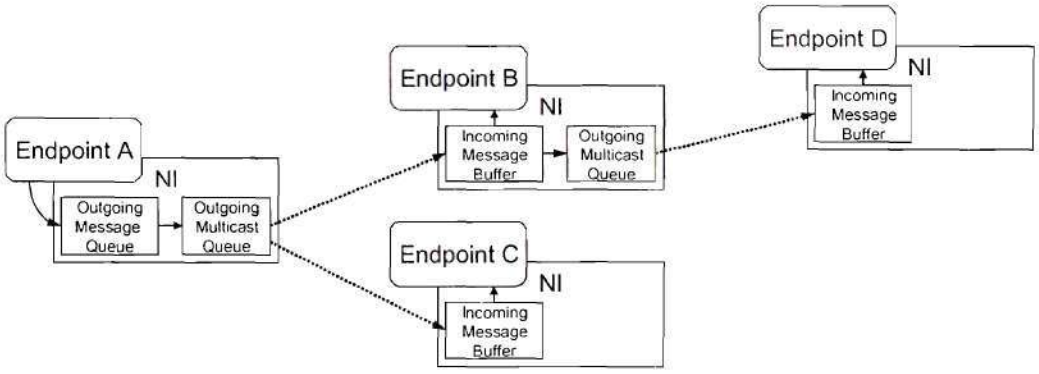


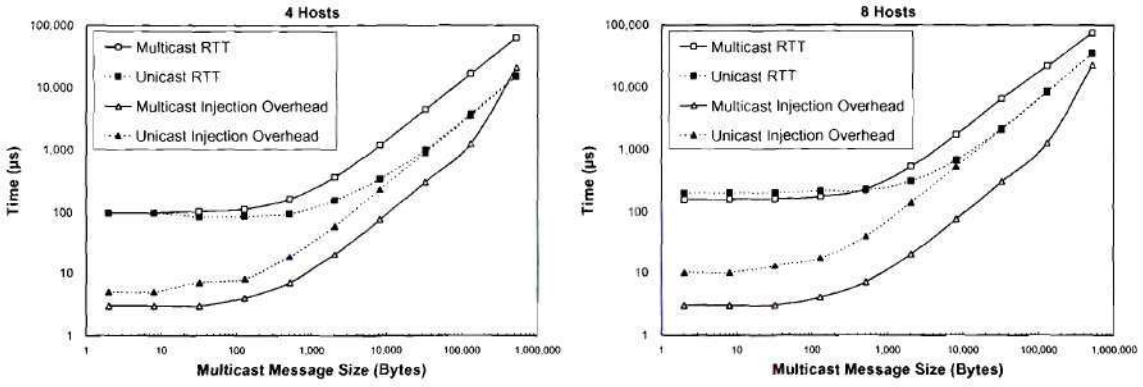
Figure 8.6: An example of the communication path for a multicast transmission.

There are two observations that must be made bout the multicast communication path. First, at the injecting node a multicast message is buffered in an outgoing message queue before it is placed in the outgoing multicast queue. This buffering is necessary in order to guarantee that when endpoints inject multicast messages, the messages are properly inserted into the multicast queue. The downside of this approach is that compared to unicast messages, multicast messages always have a bit of added delay before they are transmitted into the network. A second observation of the multicast communication path is that there are multiple locations where a NI must copy a message from one NI buffer to another. While it is possible to perform some of these transfers in a cut-through manner, NIs often have limited bandwidth for local memory transfers. A test program was constructed to measure the memory copy performance of the Myrinet cards. This program revealed that the LANai 4 and 9 cards were only capable of transferring data at 19 and 66 MB/s respectively. Because of this poor performance it should be expected that NI-based recycling methods may not reach peak transfer levels observed in unicast procedures.

8.1.7 Multicast Performance

A series of benchmarks were constructed to observe the multicast performance of GRIM. In these tests multicast messages of variable sizes are transmitted to a set of receivers, which promptly transmit a null reply message back to the sender using unicast mechanisms. The sender

measures the amount of time required to inject the multicast message and the amount of time required to injected the message and receive replies from all destinations (i.e., the overall round-trip time). The benchmark uses two methods for transmitting a multicast message: the native multicast interface and a unicast system where the injecting endpoint injects multiple copies of the message using unicast calls.



(a) (b)

Figure 8.7: Performance of multicast and unicast messages for (a) 4 and (b) 8 P4-1.7 GHz hosts using LANai 4 NI cards.

The results of the benchmark are presented in Figure 8.7 for (a) four and (b) eight P4-1.7 GHz hosts. LANai 4 NI cards were used exclusively in these tests due to a lack of LANai 9 cards. The first observation of these measurements is that the multicast operations in general require less overhead to inject but have higher round-trip timings than the unicast operations. The reduction in injection overhead can be attributed to the fact that the multicast operation only has to inject a single message while the unicast operation must inject as many copies of the message as there are hosts receiving the message. The increased round trip latency for the multicast operation is due to the relatively high overhead involved in performing NI-recycling.

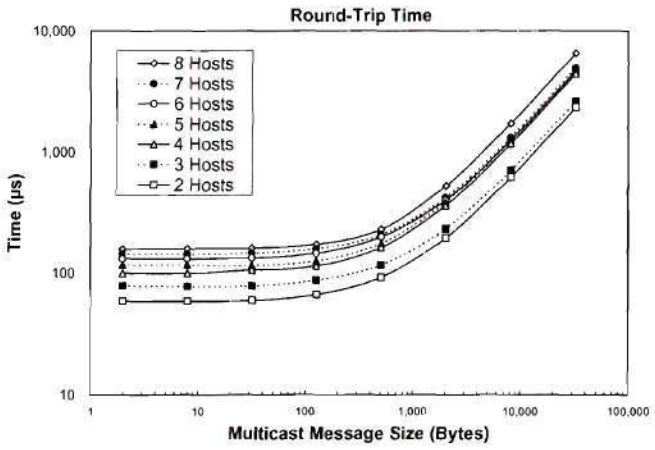


Figure 8.8: The measured round-trip times for different sized multicast groups.

The timing experiments were repeated using multicast subscription sizes ranging from two to eight hosts. The round-trip timing measurements for the multicast transmission mechanisms are presented in Figure 8.8. In these tests, subscription sizes of 2-3 and 4-7 hosts were observed to converge in performance as the multicast message size was increased. This convergence can be attributed to the fact that the depths of the binary distribution trees for these subscription sizes were equal.

8.2 Message Fragmentation and Reassembly Mechanisms

In most packet-switching and wormhole-routed networks, messages are limited in sized to a fixed maximum transfer unit (MTU). Therefore, it is beneficial to extend a communication library with functionality that allows a large message to be fragmented into a series of smaller transmissions that can be reassembled at the receiver. In addition to simplifying the communication interface for end users, these fragmentation and reassembly mechanisms can be used as a means of providing increased communication performance through message pipelining. GRIM provides a built-in mechanisms for fragmenting and reassembling active messages, remote memory messages, and multicast messages. For each category of message the fragmentation and reassembly mechanisms had to be designed to allow the messages to be transported reliably to the

application in a transparent manner. Details of these procedures are provided in this section. Performance measurements are found in Chapter 5 for active messages and remote memory messages, as well as in the preceding section for multicast messages.

8.2.1 Active Message Fragmentation in GRIM

The first effort in providing fragmentation and reassembly procedures in GRIM is for active messages. Performing fragmentation on active messages is moderately challenging because of the manner in which the messages are processed by the receiver. In GRIM a receiver cannot execute an active message until all of its data has been transferred. Therefore, the fragmentation procedures for active messages must be designed such that the receiver buffers a message's fragments and then executes the appropriate active message handler when all fragments have arrived. These procedures have been implemented in GRIM using a small number of active message handlers. The handlers use three types of active messages: one message to initialize the receiver, several messages to transfer the body of the message, and a finalization message to complete the transfer. The fragmentation and reassembly process using these messages is depicted in Figure 8.9.

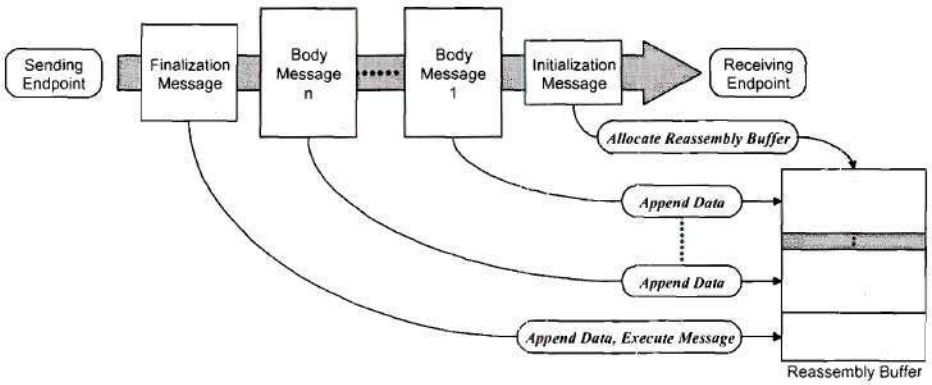


Figure 8.9: Fragmentation and reassembly of a large active message is performed by three types of active message handlers.

The first message transmitted for a fragmented active message is an initialization message, which is designed to prepare the receiver for the incoming message fragments. The initialization handler allocates a reassembly buffer large enough to house all of the fragments and then provides the receiver with basic information about the original message, such as its active message arguments and function identifier. Following the initialization message is a series of body messages that contain fragments of the original message's data. The active message handler for a body message locates the reassembly buffer being used for the transfer and then copies the body message's payload into the proper offset of the buffer. The last message in the fragmentation process is a finalization message. The function handler for this message copies the last block of data into the reassembly buffer and then invokes the original message's active message handler. Once this operation completes, the finalization handler frees the reassembly buffer and clears the data structures used in reassembling the message.

There are multiple characteristics of this implementation that are beneficial to end users. From a performance perspective this approach is designed to hide overheads incurred by the fragmentations process. Most notably, the initialization message is designed to be small so that it can be transferred to the receiver quickly. This property allows the receiver to begin allocating space for the reassembly buffer while the first body message is being transferred. Another benefit of the fragmentation process is that it is implemented using existing active message communication mechanisms. Therefore, fragmentation is easily layered on top of the system and operates in a transparent manner to end users. The use of active messages is also beneficial because end users can easily replace the fragmentation mechanisms with their own implementation by defining new active message handlers. This characteristic is particularly valuable when custom interactions with an endpoint are required by an application.

8.2.2 Remote Memory Message Fragmentation

It is much easier to implement fragmentation and reassembly procedures for remote memory operations due to the manner in which these messages are executed at the receiver. From an application programmer's perspective, remote memory operation simply transfers a block of memory from one endpoint to another and then optionally updates a user-space lock variable. Therefore, a remote memory operation can easily be divided into a series of smaller transfers that are executed individually. If the user specifies that a lock update operation is to be performed at the end of a transfer, the last message in the series of fragmented messages can be configured to perform the operation. Because GRIM guarantees that remote memory messages are processed in order, the lock update takes place after all fragments for the transfer have been executed.

8.2.3 Multicast Message Fragmentation

Fragmentation and reassembly mechanisms for multicast messages in GRIM are similar to the mechanisms used for active messages. The key difference between these efforts is the manner in which endpoints maintain information about fragmented messages. In the active message procedures a unique id to reference a fragmented message is generated from the IDs of the sending and receiving endpoints and a counter value. For multicast messages this reference value had to be modified because message fragments are transmitted to multiple receivers. Therefore, fragmented multicast messages are references with an id generated by the sending endpoint id, the multicast tree id, and a counter value.

8.3 Protocol Emulation: A Sockets Interface

One of the challenges in constructing a communication library is addressing the issues involved in presenting users with a new API. Being human, application designers are naturally resistant to adopting new APIs. Additionally, new APIs generally prevent existing applications from being utilized with the communication library due to interface incompatibility. As there may

be a significant amount of work involved in adapting existing applications to work with a new programming interface, it is desirable to provide mechanisms that allow a communication library to directly support a legacy API. This work is referred to as *protocol emulation* because the communication library provides a programming environment resembling that of a legacy API.

The GRIM communication library has been extended with functionality to support a basic emulation of the sockets API. In this emulation endpoints manage socket state in user-space and use a set of active message handlers to transfer socket data between endpoints. Macros are used to map socket API functions into the appropriate active message transactions. The emulation is able to detect whether a socket connection is for an internal cluster resource or an external host and provides the necessary connections in a transparent manner. Basic performance measurements have been made and suggest that while the sockets emulation is not as efficient as the SAN APIs, they are faster than traditional Ethernet-based mechanisms. These measurements indicate that legacy applications can directly use the communication library and benefit from its increased communication performance.

8.3.1 Sockets

The Berkeley sockets [97] interface is a well-understood mechanism for providing inter-process communication between two applications located on the same or different networked computers. At creation time a socket is specified as being either reliable (TCP based) or unreliable (UDP based). A reliable socket opens a bi-directional byte stream connection between two endpoints. While costly to establish, a reliable socket is suitable for long-term interactions between applications. An unreliable socket provides the user with a means of sending and receiving messages or datagrams between two applications. As the name suggests unreliable sockets leave the task of managing the reliable transport of data to the end application. While the extensions provided in GRIM are designed to only support reliable sockets, it is possible to implement an unreliable socket emulation in a similar manner.

Extending a communication library to support a reliable sockets interface can be beneficial for a number of reasons. First, since sockets-based programs are widely available, the application base for a communication library can be significantly increased through a sockets emulation. Second, the performance of sockets-based applications may be enhanced through the use of a properly equipped SAN communication library. In addition to using a high-performance SAN instead of a LAN, there may be performance benefits in this approach because the socket operations are performed in user space instead of kernel space [98]. Finally, a sockets emulation for a SAN communication library allows an application to use the sockets API at the same time as the library's native API. Therefore, users can construct applications that rely on the sockets API for routine endpoint interactions and then use the native SAN functions when increased performance is needed.

Because of the benefits of the benefits of a socket-based API, researchers have constructed sockets protocol emulations for existing communication libraries. One of the first and more notable of these efforts is the Fast Sockets project [99]. In this work researchers extended the FM communication library [34] to support a sockets API. The software would intercept calls made to the socket library and determine if the operations could instead be performed using the high-speed SAN and specially designed FM mechanisms. This work demonstrated that sockets calls could efficiently be layered on top of an existing SAN communication library. The researchers noted that while performance did not reach the peak levels offered by FM, there were significant gains over the traditional LAN mechanisms.

8.3.2 Planning a Reliable Sockets Emulation

There are at least three areas of development required to allow a SAN communication library to support an emulation of the sockets API. First, a conceptual model of the flow of data must be defined for the emulation. Socket data may be buffered at the receiving endpoint, the sending endpoint, or a combination of the two. While receiver-based buffering is more traditional,

the other approaches may reduce the number of memory copies involved in transferring data in the emulation. Second, the communication library must be equipped with a set of functions for facilitating the emulation. These functions must be able to transport data between socket endpoints and maintain state information used in the emulation. Finally, wrapper functions must be constructed to allow the emulation to intercept calls to the sockets API. Wrapper functions translate socket functions into appropriate SAN transactions as well as convert traditional LAN information (i.e., IP addresses) into references that can be utilized with the SAN.

8.3.3 Implementation of a Reliable Sockets Emulation

The GRIM communication library has been extended with a software package that provides an emulation of the reliable sockets API. This software utilizes a small number of active message handlers for managing sockets and C-language macros to intercept a user application's socket operations. From a data transfer perspective this package is designed to buffer socket data at the receiving endpoint. This approach was selected for latency reasons, as buffering messages at the sender requires the receiver to perform a network fetch operation when an application attempts to receive data from the socket.

Internally each endpoint in the emulation maintains a list of open socket connections. An endpoint marks a port in this database as being available when an endpoint performs a socket operation for accepting a new connection. A connection is established when another endpoint in the cluster attempts to open a connection to the endpoint at an available port. Once connected two endpoints allocate data structures for buffering incoming socket data. When an endpoint injects data into the socket, an active message is used to transport the data to the remote socket endpoint and append the data to its socket data buffer. The socket's read operation then examines the local buffer and extracts data as it becomes available.

One of the hardships in constructing a sockets interface is distinguishing between sockets used for internal SAN interactions and sockets used for other operations. For example, an

application uses the same read and write operations to interact with a socket as it does a file. Therefore, if read and write operations are intercepted by the emulation, the emulation must be able to determine whether to use the SAN library or to use the traditional library function. This task is performed in GRIM by intercepting all of the calls that manage file operations. When a new socket is opened, GRIM determines if the destination is a cluster resource and assigns these resources a specially marked file handler. The file handler returned for interactions with non-cluster resources is simply the file handler returned by the initialization operation. At runtime when a socket or file is accessed, GRIM can determine whether to use its SAN functionality simply by examining the file handler.

8.3.4 Performance

A benchmark program was constructed to measure the performance of the GRIM sockets emulation. This program was written using the traditional sockets API and therefore can be used with both a GRIM-based Myrinet network and a TCP-based 100 Mb/s Ethernet network. Selecting the communication library and network to use in the benchmarks is performed by setting a switch in the compile process. The benchmark program is designed to establish a connection between two hosts and transfer a block of data between the hosts in a round-trip fashion.

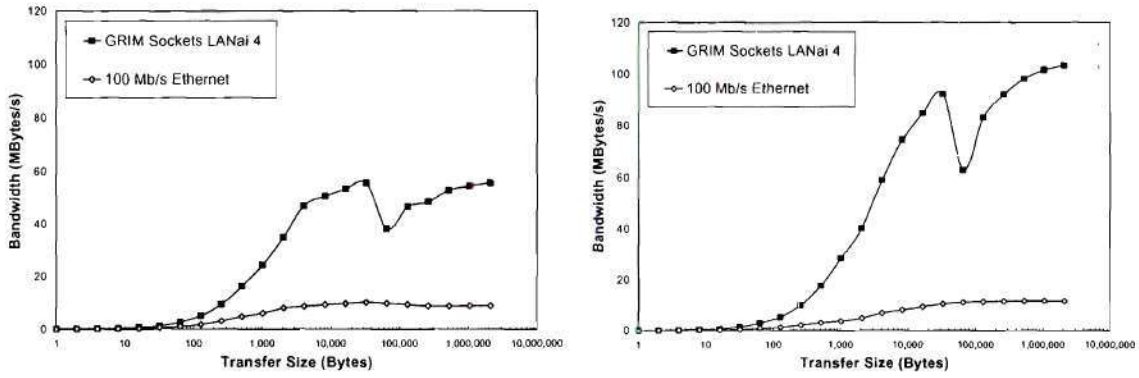


Figure 8.10: Performance of the GRIM sockets emulation using LANai 4 NI cards compared to 100 Mb/s Ethernet for (a) P3-550 MHz and (b) P4-1.7 GHz hosts.

Table 8.1: Comparison of the performance of TCP and GRIM Sockets.

API	Network	P3-550 MHz		P4-1.7 GHz	
		Latency (µs)	Bandwidth (MB/s)	Latency (µs)	Bandwidth (MB/s)
TCP	100 Mb/s Ethernet	58.8	10.1	62.5	11.7
GRIM Sockets	LANai 4 Myrinet	22.7	55.6	22.2	103.3

The results of the benchmark experiments are presented in Figure 8.10(a-b) and summarized in Table 8.1. As expected the GRIM sockets emulation outperformed TCP-based Ethernet for all transfer sizes due to the superior performance of the Myrinet SAN. GRIM provides roughly a third of the latency of TCP sockets and up to nearly nine times the bandwidth. The performance characteristics of the GRIM sockets API reveal that as messages become larger GRIM is able to provide better performance until a transfer size of approximately 64 KB. At this point GRIM begins fragmenting transmissions, resulting in a dip in performance. While performance begins to increase after this dip, it should be noted that the performance is not as high as that observed with the active message and remote memory interfaces. This characteristic can be attributed to the fact that the active message socket handlers allocate a new block of memory for every incoming socket fragment and add the block to a linked list. This differs from the fragmentation mechanisms in the active message interface where memory for a large transfer

is allocated one time in advance and then filled with a series of transfers. However, the performance of this approach is reasonably high and is therefore beneficial as a means of improving the performance of legacy applications.

CHAPTER IX

CONCLUSION

Resource-rich clusters are an emerging form of cluster architecture where both host CPUs and peripheral devices are utilized by distributed applications. While these clusters can be physically constructed from commercially available hardware, the enabling technology for these systems is specially designed message layer software. Current generation message layers are ill equipped to handle the communication needs of these clusters because they are by design CPU centric. Therefore, this thesis has addressed the design of new message layers that are able to support efficient interactions between applications and a cluster's host CPUs and peripheral devices.

The work presented in this thesis advocates migrating communication functionality in the message layer from the communication endpoints into the NI when possible. This migration reduces the workload of the endpoint and simplifies the task of adding new peripheral devices to the cluster architecture. Three primary design characteristics for message layers have been discussed as a means of accomplishing this task. First, end-to-end flow control in the message layer is managed on a per-hop basis in order to simplify communication protocols for endpoints and provide better dynamic buffer management. Second, logical channels are employed in the NI to allow multiple endpoints in a host to efficiently share a singe NI. Finally, two programming interfaces are defined for the message layer to support a rich set of communication functions. An active message interface provides a powerful means of controlling peripheral devices while a remote memory interface allows users to perform low-level transfers of memory between cluster resources.

A critical characteristic of a message layer for resource-rich clusters is extensibility. Users of resource-rich cluster computers frequently need to perform custom operations and therefore need to be able to layer new functionality on top of existing message layer software. More precisely a message layer must be extensible in at least two dimensions. In a hardware dimension, the message layer must provide means by which new peripheral devices can easily be incorporated into the cluster environment. In a software dimension, a message layer must be designed to allow users to add new application-specific functionality. These additions can be made at the endpoint level (e.g., the sockets emulation) or at the NI level (e.g., NI support for multicast).

GRIM is a message layer that has been constructed with the preceding design principles. GRIM has been utilized to integrate four different peripheral devices into a cluster architecture. The fact that these devices have a diverse range of capabilities illustrates that GRIM's communication mechanisms are flexible and sufficient for the needs of resource-rich cluster computers. Multiple application-specific extensions have also been constructed for GRIM. These extensions include methods for performing streaming computations in the cluster, NI-supported multicast, fragmentation and reassembly, and an emulation of the sockets API. These extensions demonstrate that new functionality can easily be layered on top of GRIM's core communication operations.

GRIM's performance has been evaluated and compared to existing message layers. For host-to-host transmissions GRIM obtains a maximum bandwidth of 146 MB/s (1.168 Gb/s) and exhibits latencies as small as 8 μs. This performance is comparable to existing message layers, indicating that it is possible to add resource-rich cluster functionality to the message layer without severely impacting the performance of traditional communication operations. Attempting to implement GRIM's functionality in other message layers is impractical and at the very least highly inefficient due to the manner in which these message layers are designed.

9.1 Implementation Challenges

A significant portion of the work presented in this thesis deals with the challenge of overcoming the limitations of commodity hardware. Some of the more challenging aspects of working with commodity hardware, peripheral devices, and cluster computers in general include the following.

- **Low-level Operation**: Peripheral devices operate as low-level hardware appended to the host system. Programming at this level can be challenging for a number of reasons. Errors at this level often have catastrophic effects on the host. For example, programming a DMA engine with bad pointers can result in the entire operating system being relocated in physical memory. These errors can be difficult to locate as it is more likely that a bad DMA will simply corrupt a random location of host memory that may not cause an immediate system crash. A key to working in this environment is to construct protective debugging mechanisms around functions that pose a risk to system stability.
- **Blind Debugging**: Another programming difficulty in dealing with peripherals is that is often difficult to observe the behavior of firmware. While some high-end cards such as the I_2O adaptor are equipped with a serial debugging connection, many cards have no other monitoring equipment other than LEDs and memory. A significant amount of the work in dealing with the peripheral devices used in GRIM involved constructing debugging frameworks, such as a journaling systems to record card operations. These facilities are essential to observing low-level card behavior.
- **Device Limitations**: One of the most significant problems encountered in this work is simply dealing with the fact that peripheral devices are utilized in ways they were not originally intended for. Peripherals devices are typically built on the assumption that only the host CPU will communicate with the card. This assumption is often used to justify

minimizing peripheral device functionality when a host driver can perform the same functions. Therefore, work in resource-rich clusters often requires defining new mechanisms by which existing devices can be adapted.

- **Byte Endian and Alignment Issues**: It is common for the processor of different peripheral devices to use a different byte endian order than the host processor. For example, network cards are often big endian (to match network byte order) while x86 processors are little endian. All communication between the host and the NI must be translated to match the destination's endian order. Alignment is a similar issue in that some peripheral devices require data to be aligned on specific byte boundaries. For example, the LANai 9 NI's DMA units require memory address to be align on 64-bit boundaries. Therefore, communication software must be designed to place the right data in the right locations.
- **Limited Data Transfer Mechanisms**: Each peripheral device generally has a card-specific set of hardware for performing operations such as DMA transfers. To complicate matters, some peripheral devices do not provide all of the desired mechanisms for performing data transfers. For example, while the I_2O card features DMA engines, the engines can only be initiated by the card. This adds to the complexity of transferring data to the device from entities such as the host CPU that do not have a built-in DMA engine. Therefore, it is beneficial to use a library such as TPIL to accelerate I/O operations.
- **An Evolving OS Kernel**: Over the last five years, GRIM has had to be adjusted to operate with three different versions of the Linux kernel (2.0, 2.2, and 2.4). Each of these transitions required a number of modifications to the device drivers built for GRIM. While it is natural and desirable for an OS to evolve with improvements, maintaining both a working knowledge of the kernel and a functional custom device driver can require a significant amount of effort. One method of dealing with a changing kernel is to move application functionality from the kernel-level device driver to user-space software.

- **Poor Documentation**: A universal problem with working with peripheral devices is that usually there is a lack of decent documentation. Vendors often do not release low-level details for a peripheral device to prevent competitors from leveraging their work. Therefore, the only options for developers are reverse engineering and methods based on trial and error. Discussing driver issues with other Linux developers can greatly help in this work.
- **Deadlock**: Deadlock is an important issue that needed to be addressed at all levels of GRIM's development. Any time new functionality is added to a message layer the designer should check to observe whether the operation holds one resource while waiting for another. Deadlock prevention techniques do not have to be complicated and can often be implemented with sufficient buffering.

9.2 Future Directions

The work presented in this thesis provides the first steps in constructing extensible message layers for resource-rich cluster computers. This work can be continued in multiple directions.

9.2.1 GRIM Enhancements

The current version of GRIM provides a basic, flexible substrate for allowing cluster resources to communicate efficiently. However, there are a number of improvements that can be made to the implementation. First, since GRIM is designed to operate with both the old and new versions of the LANai NI processor, the NI firmware does not take advantage of hardware features found in the new card. GRIM's performance could therefore be enhanced by making use of card-specific functions such as the PCI doorbells. Second, like many Myrinet message layers, GRIM only allows one host-level application in a host to be connected to the network at a time.

GRIM could be modified to support multiple applications at a time by allocating each application a separate logical channel in the NI. Finally, GRIM can be enhanced by adding new and different peripheral devices to the communication model. Given the flexibility of GRIM and the four existing peripheral device examples, this work can be performed in a relatively straightforward manner.

9.2.2 Gigabit Ethernet Substrates

It is useful to consider how the Myrinet SAN currently used in GRIM could be replaced with commodity Gigabit Ethernet LAN equipment. Gigabit Ethernet is in general more affordable than Myrinet hardware and is widely utilized for clusters. Adapting GRIM to support Gigabit Ethernet would therefore provide an opportunity for a large number of existing clusters to function as resource-rich clusters. The first task in this effort is selecting a Gigabit Ethernet NI card that can be programmed with GRIM's low-level NI functionality. Multiple Gigabit Ethernet cards can be utilized in this effort, including the Alteon AceNIC [100] card, the Intel Pro/1000 series [101] cards, and network cards based on the Intel IXP processor [102]. Emerging IXP cards provide the most promising environment for this work as the cards are very powerful and are well supported by Intel. The IXP cards feature multiple Gigabit Ethernet ports, up to 256 MB of memory, and multiple threaded microengine processors.

Adapting GRIM's NI software to a Gigabit Ethernet NI platform would require changes to some of the basic functions of the NI. At a fundamental level, message data structures would have to be modified to meet the formatting requirements of the new network. A more challenging aspect however is dealing with the communication differences between the Myrinet SAN and a Gigabit Ethernet LAN. While Myrinet provides highly-reliable data transmissions, messages may be dropped or reordered in Gigabit Ethernet LANs. Therefore, it is necessary to modify GRIM's reliable transmission mechanisms to account for these factors. While GRIM's in-order delivery mechanisms can be utilized to sort out-of-order messages, extensions are necessary to protect

against dropped messages. These modifications involve adding timeouts mechanisms for data message transmissions so that dropped messages are automatically retransmitted. Once a reliable network transmission protocol is established for Gigabit Ethernet NIs, adapting the remaining portion of GRIM's functionality should be a relatively straightforward process.

9.2.3 Active SANs

Another direction for future research related to this thesis is in the field of active SANs. As the streaming extensions of this thesis have demonstrated, it is possible to utilize FPGAs as processing elements in a cluster's network substrate. The next step in this effort is to reduce the distance between the FPGAs and the NI. One such approach is to include an FPGA on a NI card. The advantage of this architecture is that the FPGA can process network messages without costly traversals of the PCI bus. Since the FPGA can process messages at a finer granularity, it is possible for the FPGA to play a more pivotal role in streaming computations.

Emerging FPGA architectures provide another opportunity for research exploration related to active SANs. Recently announced FPGA chips such as the Xilinx Virtex-II Pro [103] include large amounts of reconfigurable logic, network transceivers, as well as dedicated processor cores. These chips will be capable of directly interacting with the physical links of networks such as InfiniBand. Therefore, these FPGAs can be visualized as the next generation of high-performance NI chips. In these chips a portion of reconfigurable logic and CPU processing time will be utilized to implement network interactions. The remaining resources of the chip can be utilized to implement custom computations for streaming operations. Since the NI and FPGA are implemented in a single chip, it is expected that constructing a streaming computational system will be much more efficient and straightforward than the effort required to incorporate the RC-1000 FPGA card as a coprocessor. However, the system for performing streaming computations on network messages presented in this thesis is applicable to this architecture and provides a starting point for future research in this field.

APPENDIX A

SUMMARY OF MYRINET NI PERFORMANCE CHARACTERISTICS

The following are architectural and performance characteristics for the Myrinet LANai 4 and 9 NI cards utilized in this work.

Feature	LANai 4	LANai 9B
NI Clock Frequency	33 MHz	133 MHz
NI Memory	1 MB	2 MB
NI PCI Interface	32b/33MHz	32-64b/33-66MHz
SAN Interface	SAN-1280	SAN-1280/M2000

Operation		Min/Max	LANai 4	LANai 9B
PCI DMA time		Min	2 µs	2 µs
SAN DMA time		Min	3 µs	1 µs
Interrupt service time		Min	6.5 µs	6 µs
On-card memory copy bandwidth		Max	19 MB/s	66 MB/s
PCI bandwidth	32b/33MHz	Max	131 MB/s	132 MB/s
	64b/66MHz		-	303 MB/s
SAN bandwidth	SAN-1280	Max	132 MB/s	149 MB/s
	M2000		-	200 MB/s
Scan N logical channel data structures	1	Min	1 µs	0.5 µs
	8		5.5 µs	1.5 µs
	16		9 µs	3 µs

APPENDIX B

THE FPGA FRAME API

Field-programmable gate arrays (FPGAs) have steadily evolved over the last decade as a means of accelerating a number of computational tasks through the use of reconfigurable hardware. Given the potential for this technology it is beneficial to investigate methods by which FPGAs can be integrated into the cluster computer architecture and efficiently utilized by end applications. Unfortunately, integrating an FPGA into a cluster can be extremely challenging due to the limited types of resources these cards employ. Most commercial FPGA cards employ one or more FPGAs, a cache of on-card memory, and a simple PCI controller. Because these cards often lack a general purpose CPU, it is often necessary to construct a state machine in the FPGA that serves as an interface between a user's computational circuits and external resources such as on-card memory or the host CPU.

In order to integrate the Celoxica RC-1000 FPGA card into a cluster computer utilizing the GRIM communication library, it was necessary to design and implement a block of FPGA circuitry that managed interactions between the FPGA's computational circuits and end applications. This block of logic is known as the FPGA's static *frame* because it allows a *canvas* of user-defined computational circuits to be insulated from the card-specific features of the RC-1000 device. This section describes the low-level mechanics of the frame and provides an API by which end users can interact with the FPGA device. While the frame is designed to operate specifically with the RC-1000 card, it is possible to adapt this work for use with other similar FPGA cards.

B.1 Architecture Overview

As depicted in Figure B.1, the RC-1000 implementation of a GRIM communication endpoint is divided into two contexts: the *static frame* unit and the *dynamic circuit canvas*. The frame serves as a reusable block of hardware that allows different computational circuits to be dynamically plugged into one of the cluster's FPGA devices. The frame provides three specific interfaces to insulate a user's circuits from the device specific characteristics of the target FPGA card. First, the frame implements a communication library API that is responsible for handling messages coming from or going to the communication library. Second, the frame provides an interface to the dynamic circuit canvas that allows multiple user-defined circuits to be connected to the frame. Finally, the frame provides an interface that allows applications to access a region of on-card memory known as the scratchpad.

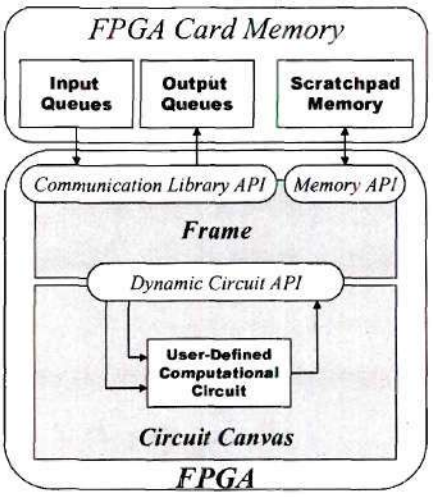


Figure B.1: The three interfaces managed by the FPGA frame.

B.1.1 Data Path of the Frame

A simplified view of the Celoxica RC-1000 frame's low-level data path is depicted in Figure B.2. The four SRAM banks available on the RC-1000 are allocated as follows. Bank 0 houses incoming message queues for the communication library as well as runtime information for the frame. SRAM banks 1 and 2 are utilized as scratchpad memory for storing application

data. SRAM bank 3 houses the outgoing messages for the communication library. The control/status port on the RC-1000 provides a simple means of transferring 8-bit data values with the host. This port can be configured to transmit an interrupt to the host and is used to pass simple state information between the host and card.

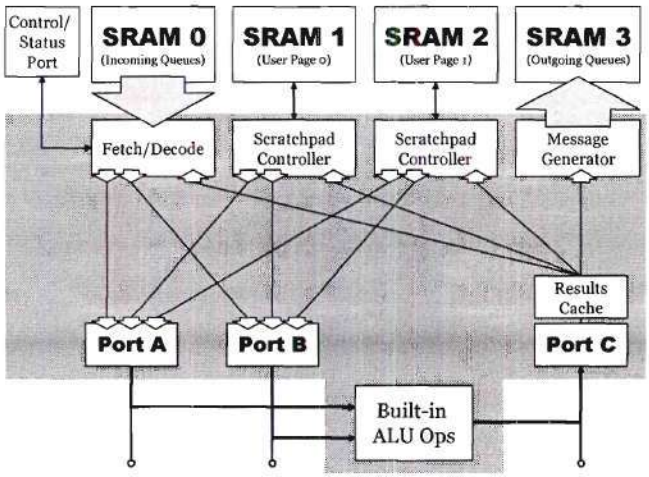


Figure B.2: The internal structure of the frame for the RC-1000 implementation.

The individual units in the frame architecture are described as follows:

- **Fetch/Decode unit**: This unit fetches the next message to be processed by the frame and establishes the necessary data paths through the frame to process the message. A message can originate from either the endpoint's message queues (housed in SRAM bank 0) or from a recycle buffer which contains the previously generated outgoing message.
- **Scratchpad controller unit**: This unit is used to exchange data with the scratchpad memory (SRAM banks 1 and 2). A single SRAM bank can supply both input vectors and accept the output vector of the user-defined circuit if needed. Vector data is fetched and stored linearly starting at memory offsets provided in the incoming message's header.
- **Results cache**: The results cache is used to buffer the output of a computation until the frame is able to write the data into its proper destination. The cache is utilized only when

an operation needs to fetch and store data with the same scratchpad memory bank, or when input data is fetched from an incoming message and output is written to the recycle buffer.

- **Message generator**: This unit takes results generated by the computational circuit, formats the data into an outgoing message, and inserts the data into an outgoing message queue (located in SRAM bank 3).
- **Vector data ports**: The frame provides three vector data ports, to which all user-defined circuits are connected. Ports A and B provide input streams to the circuits while port C receives output data generated by the circuits.
- **Built-in Ops**: The frame provides a simple built-in computational circuit that can perform a variety of common vector operations, including add, subtract, multiply, min, max, invert, and pass.

B.2 Communication Library Interface

The first interface that the frame provides allows the FPGA to interact with the communication library. This interface is responsible for managing incoming and outgoing message queues, and must be designed to work with the message format specified for a given communication library. The RC-1000 implementation of the frame utilizes messages formatted for the GRIM communication library, although it is possible to adjust the implementation to operate with other libraries.

B.2.1 GRIM Message Format

The RC-1000 implementation of the frame processes messages that are formatted for the GRIM communication library. Like other communication endpoints found in GRIM, information included in the header of each message is used to specify how the RC-1000 should process a message. The active message function handler identifier for the RC-1000 corresponds directly to

the globally unique user-defined circuit that is used to process the message. Because of the flexibility that the frame provides in processing a message, it is necessary to encapsulate additional information in the message header. This information resides in the arguments section of the active message header and is used to configure the frame's data paths to meet an application's needs. The fields used to configure the FPGA are listed in Table B.1.

Table B.1: The data fields in an active message header that control the operation of the frame and the corresponding bit lengths.

Arg [0]	Forward ID (8)	A Driver (1)	B Driver (1)	C Driver (1)	Reserved (7)	Sub-Op (4)	Op-Length (10)
Arg [1]	Port A Virtual Address (29)						
Arg [2]	Port B Virtual Address (29) or Port B Constant (32)						
Arg [3]	Port C Virtual Address (29)						

B.2.2 Message Queues

The frame implements three different types of message queues. The first category of message queue is used to house incoming messages for the card. These queues are located in SRAM bank 0 and adhere to the append-style of queuing utilized throughout GRIM. The current implementation of the frame provides two separate incoming message queues, with the intention that one queue is for the host CPU and the other for the NI. The frame periodically polls each of these queues to determine if new messages are available. This polling operation takes place every 300 FPGA clock cycles and requires less than a dozen clock cycles to poll for new data.

The second place where messages can be stored is in the recycle buffer. This buffer is housed in SRAM bank 0 and has room for exactly one message. This buffer is utilized when an application needs the FPGA to perform a series of operations on a set of data. Users can specify a message be recycled by setting the C-Driver bit in the message's header to zero. The frame will then route the results of the computation into the recycle buffer and insert the proper header from

the forwarding table. The frame provides a guarantee that if a message is placed in the recycle buffer, it will be selected as the next message processed by the FPGA. This guarantee is necessary to prevent multiple messages from being inserted into the recycle buffer. Therefore, it is imperative that users prevent endless recycling loops in the forwarding table.

The third type of message queue controlled by the frame is for outgoing messages. Currently there are two outgoing message queues that are housed in SRAM bank 3. In order to simplify the task of managing these queues, the frame implements a slotted queuing system. Because the FPGA card cannot directly trigger the DMA engines, the frame must notify the host when data is available in the message queues. This operation is performed by updating the RC-1000's status register, which the host periodically polls. When the host detects new messages in the card's outgoing message queues, it can perform the necessary transfer of data to the proper endpoint.

B.2.3 Forwarding Registers

A key design point for the RC-1000 frame is that it is able to process incoming messages and generate outgoing messages. This allows the card to be utilized as an intermediate computational stage as opposed to simply a unit that sinks data. Therefore, mechanisms have to be present in the frame to allow messages to be ejected by the frame into the communication library. Like other peripheral devices in the GRIM environment, it is expected that the card will generate messages only in response to a stimuli, such as the detection of a new incoming message. The hardship in implementing such a system is providing an interface where users can specify the types of response messages that the frame generates. The implementation of the RC-1000 frame utilizes a set of forwarding registers to solve this problem.

The term forwarding registers in the GRIM environment refers to a database in a communication endpoint that contains information used to format outgoing messages. For the RC-1000 frame this database is implemented as a large table of user-programmed message

headers. All incoming messages have a field in the message header that specifies which table entry (if any) the frame should reference to generate an outgoing message. The frame copies the information from the specified entry to the outgoing message and places the results of the computation in the payload section of the message. Users can adjust the forwarding register table entries through a set of built-in active message handlers for the frame. The set_pipeline handler simply copies 64 bytes of payload into the specified forwarding register entry. It is the responsibility of the user to allocate and manage forwarding registers in this table.

The forwarding registers for the RC-1000 are located in SRAM bank 0 starting at address 0 in the current frame implementation. The table contains 256 entries, with each entry housing a single message header (64-bytes). The frame is designed to reference the forwarding registers only when a header needs to be placed on a message that is generated. For these situations the frame operates as follows. First, the frame processes a message in a normal manner. The message header is fetched, the frame data paths are established, and data is streamed through a specified computational unit. The results of this computation are routed to the payload section of the generated message, whether the generated message is assembled in an outgoing message queue slot or the recycle buffer. Next, the frame uses information from the incoming message to generate an index into the forwarding register table. The message header located at this entry is then streamed into the header section of the generated message. Finally, the frame updates the sender id and the payload length fields of the message header to guarantee that the generated message is properly identified.

B.2.4 Active Message Circuit Identification

Once the frame receives an incoming message it must determine which user-defined circuit is utilized to process the data. Conceptually, user-defined circuits are similar to function handlers found in any other communication endpoint in GRIM. Therefore, each user-defined circuit is labeled with a unique active message handler identifier that end applications can

reference to perform a desired computation. However, unlike other function handlers used in GRIM, user-defined circuits are statically assigned active message identifiers. When creating a new circuit, a user must define a new static active message handler identifier for the circuit in the grim_handlers.h file. This file contains a static list of handler IDs for various functions utilized in the GRIM library. Once identified, users can reference a user-defined circuit with a simple constant as opposed to locating an identifier for the circuit through the runtime handler database. An advantage of this approach is that it simplifies the task of forwarding data between FPGA computational circuits because all circuit identifiers are known in advance.

At runtime the frame must be able to determine which user-defined circuit is utilized to process an incoming message. In the RC-1000 implementation of the frame this is accomplished by comparing an incoming message's active message handler id to a list of the FPGA's user-defined circuits. This list is managed by the host and updated whenever the FPGA's configuration is updated. Specifically, the host stores the list of a configuration's user-defined circuits in the card's SRAM before a configuration is loaded into an FPGA. After the FPGA is reset, the frame loads this list of functions from SRAM into an internal set of registers. When the frame observes an incoming message, it compares the active message handler to the list of available circuits. If the requested circuit is available the frame establishes the data path necessary to connect the user-defined circuit to process the message. If the circuit is not available, the host is notified of the problem with a function fault.

B.2.5 Function Faults

A function fault is when an incoming message requests an active message handler that cannot be satisfied with the user-defined circuits that are currently available in an FPGA. The first phase in a function fault is for the FPGA to store all of its runtime state information in on-card SRAM. This data includes the id of the function that caused the fault as well as the frame's current set of message queue pointers. Future versions of the frame may also include the runtime

state information of individual user-defined circuits in this operation. After runtime information is stored the frame notifies the host of the function fault through the card's status register. The frame then suspends operation until the host passes an activation signal to the frame through the control register.

Once the host detects a function fault it must load the id of the missing user-defined circuit and determine how the FPGA should be reconfigured. In the current implementation the FPGA is reconfigured in its entirety. Therefore, the host simply locates an FPGA configuration in its database that features the missing hardware and loads the configuration onto the FPGA. This process includes writing the new FPGA configuration's list of user-defined circuits to the card's SRAM, loading the FPGA with the new configuration, and triggering the FPGA reset. The FPGA then loads its runtime state information from SRAM and continues processing where it left off.

B.3 Computational Circuit Interface

The frame allows multiple user-defined computational circuits to exist in the dynamic circuit canvas, as illustrated in Figure B.3. Each user-defined circuit is connected with two vector inputs (labeled as ports A and B) and one vector output (labeled as port C). For simplicity the frame is designed to allow only one user-defined circuit to be active at any given time. When the frame detects a new incoming message, it sends an activation signal to the user-defined circuit specified in the message's header and then routes data into and out of the vector data ports. Vector data ports are asynchronous and provide sequential streams of data using a simple control protocol. Circuit designers are free to utilize these ports in any manner they desire, as long as unused ports are properly grounded.

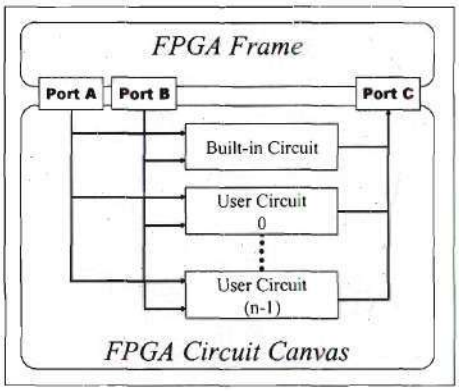


Figure B.3: The interface between the FPGA's frame and circuit canvas.

B.3.1 Vector Data Port Signaling

Vector data ports are designed to operate in an asynchronous fashion. A simple valid/acknowledge handshaking protocol is utilized to allow either the sender or receiver of a vector data port to stall the passing of data. By design the sender and receiver of a vector data port operate on opposite clock edges. The frame inverts the clock supplied to user-defined circuits so both units have the appearance of operating on the rising edge of the clock. Because the sender and receiver are on opposite clock edges it is possible for a new data value to be transferred every clock period. The interface for a port is depicted in Figure B.4.

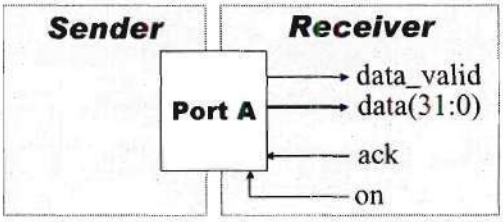


Figure B.4: The signals for a vector data port.

The signals for a vector data port are as follows.

- **On:** The user-defined circuit must assert the on signal for the entire time it needs to transfer data with the port. Therefore, the first action a user-defined circuit must perform

when it is activated by the frame is asserting the on signals for all data ports that will be used when processing a message. Once the on signal is turned off the frame will stop attempting to transfer data with the port. All vector port on signals must be set to low before job_done can be triggered.

- **Valid:** The transmitter for a port signifies that the next word from the vector port has been placed on the data lines. Valid remains high until the receiver of a port asserts an acknowledgement. Note that because sender and receiver are on opposite clocks, it is possible for the valid signal to remain high for multiple clock periods if the receiver can accept data every clock signal and assert the acknowledgement signal.
- **Data:** The data lines provide the next 32-bit data value when valid is asserted.
- **Acknowledge:** The acknowledge (ack) signal is asserted for a single cycle when the receiver reads a valid data value. An acknowledgement triggers the sender to set the valid signal to low, unless the next vector data value can be placed on the data lines immediately.

B.3.2 Circuit Interface Signals

The signaling API for a user-defined circuit is depicted in Figure B.5. In addition to the three vector data ports, a user-defined circuit must manage a set of control signals in order to properly communicate with the frame. The failure to correctly generate these signals can result in either erroneous data or the entire frame being suspended in an endless loop. All signals for a user-defined circuit are active only on the rising edge of the provided clock signal. Data presented to the user-defined circuit is generated on the clock's falling edge.

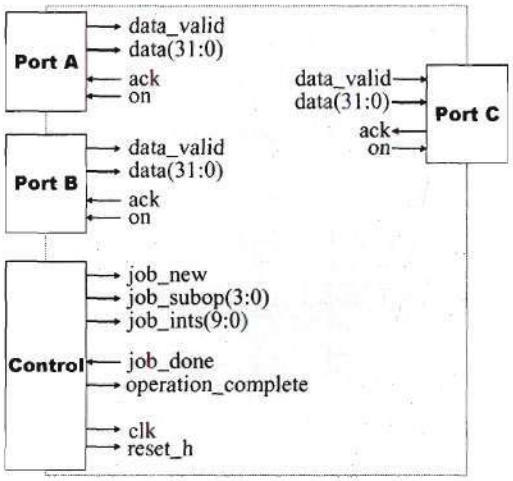


Figure B.5: The interface for a user-defined circuit.

The control signals passed between the frame and the user-defined circuit are as follows.

- **clk:** This is the clock signal provided to the unit. A user-defined circuit must assert signals on the rising edge of this clock. Note that this clock is an inverse of the clock used in the frame, so that both the frame and user-defined circuits can operate with rising edge clocks.
- **reset_h:** This signal is an active high reset for the unit. After observing a reset the user-defined circuit must initialize itself and move to a state where it waits for notification from the host that it is to perform a computation on a message.
- **job_new:** The job_new signal is activated for a user-defined circuit when the circuit is the unit that is needed to processes a new message. This signal remains asserted until the unit completes its task and the frame completes all message processing tasks.
- **job_subop:** This 4-bit signal is an optional field that can be used to allow a user-defined circuit to perform different operations. For example, a cryptography circuit may be capable of performing encryption and decryption operations. The user can specify which of these operations to perform with the subop field.

- **job_ints:** This 10-bit signal specifies how many 32-bit words of processing the user expects the circuit to perform.
- **job_done:** When the user-defined circuit completes all of its operations it asserts the job_done signal to notify the frame that it is finished processing a message. The job_done signal must remain asserted until the frame pulses the operation_complete signal. It may take several cycles for the frame to issue the operation_complete signal because it may need to flush cached data.
- **operation_complete:** This signal indicates that the frame has completed all processing necessary for a message. After receiving this signal all user-defined circuits must revert back to an initial wait state where they wait for job_new to be triggered. All control signals for a user-defined circuit must be set low after receiving an operation_complete in order to prevent any false starts in the system.

B.3.3 Example Operation

The timing diagram in Figure B.6 provides an example of the signaling required for a user-defined computational circuit. In this example, the circuit is designed to simply read four values from vector data port A and then signal its completion. The clock signal provided in this example is from the user-defined circuit's perspective.

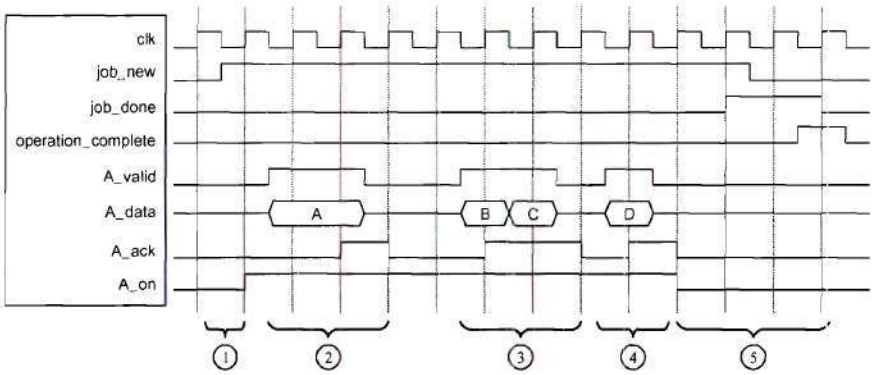


Figure B.6: An example timing diagram for an asynchronous data vector port.

Details of the signaling for this example are as follows.

1. The first phase of operation is for the user-defined circuit to be activated. In this process the frame asserts the job_new signal and waits for the circuit to respond with the A_on.
2. Once activated the port can begin exchanging data values. In the first transfer the frame asserts the A_valid signal and places 'A' on the A_data bus during a falling clock. After some time the user circuit acknowledges the transfer by pulsing the A_ack signal. In this case the frame does not have the next data value ready so the A_valid signal is set to low. This example illustrates how the user circuit can delay the transmission of data.
3. After some time the frame has two data values to transmit. The frame begins the transfer by placing 'B' on the A_data bus and setting A_valid. The user circuit is able to accept this data and replies by setting A_ack high. This process is repeated immediately for 'C'. This transfer illustrates that the protocol can transfer multiple data values, with each transfer taking a single cycle.
4. The final transfer is for the single data value 'D'. In this example, both the frame and user circuit are ready to transfer the value, resulting in a total transmission time of two clock periods.
5. The final task is for the circuit to complete its operations. After the circuit has completed all of its computations it sets A_on low. The circuit must then wait for the

operation_complete signal to be pulsed, which signifies that the frame has finished storing the results of the computation. After this point the user circuit resets itself to a state where it waits for a job_new signal.

B.4 Scratchpad Memory Interface

The third interface that the frame provides allows user-defined circuits to access data located in a block of on-card memory defined as the scratchpad. The scratchpad is designed as a temporary storage space for housing large sets of application data on the FPGA card. Its primary benefit is that it allows application data to be stored in close proximity to the FPGA. This locality can improve the computational performance of user-defined circuits by decreasing the latency at which data can be supplied to the inputs of the circuits. A second benefit of the scratchpad memory is that it increases the flexibility of the FPGA as a computational resource in the cluster architecture. The frame is designed to be able to connect the scratchpad to a user-defined circuit's input and output vector data streams. Therefore, the results of one computation can be stored in the scratchpad for use in future computations. This mode of operation allows the FPGA to be utilized in a more practical manner since the scratchpad can be used as a means of storing dynamic state for an application.

There are a number of design issues involved in constructed a scratchpad memory interface. Primarily these issues are related to two challenges: defining fundamental elements of operation for the interface and mapping these elements into card-specific architectures. This section discusses three aspects of the design of the scratchpad interface for the RC-1000: meeting the needs of user-defined circuits, maintaining correctness in the flow of data, and providing expansion mechanisms to overcome the limitations of the architecture.

B.4.1 Supplying and Sinking Circuit Data

The first requirement for the scratchpad memory system is that it is able to meet the needs of user-defined circuits. Fundamentally this task is a relatively straight-forward procedure due to the API of user-defined circuits. The worst-case scenario under this API is when the scratchpad is connected to both input ports and the output port of a computational circuit. Because of this case the scratchpad must be capable of concurrently supplying two vector data input streams to and accepting one vector data output stream from a user-defined circuit. The challenge in implementing such a system is managing each of these data streams in an efficient manner using the resources that are available on a card.

The RC-1000 implementation of the frame utilizes the card's multi-channel memory architecture to improve the performance of the scratchpad interface. In this system SRAM banks 1 and 2 are dedicated exclusively to housing scratchpad data. A benefit of this approach is that the scratchpad interface can simultaneously service two vector data streams if the streams are located in different memory banks. On the other hand if two streams are located in the same memory bank, the scratchpad interface must take turns servicing each of the data streams. Therefore, it is beneficial for users to strategically stripe data sets in different memory banks in order to improve performance. A state machine was written to implement the necessary flow of data into and out of a bank of scratchpad memory. This state machine was replicated for the second bank of scratchpad memory, illustrating that this approach can be easily extended to card architectures where several separate banks of memory are available.

B.4.2 Maintaining Correctness in Scratchpad Data Streams

An important element of the scratchpad interface is maintaining data correctness for a computation. In the scratchpad API a user specifies the starting address and length for each data vector utilized in a computation. Therefore, it is possible for a user to specify an input vector and an output vector that overlap in scratchpad memory. The hazard here is that it is unclear whether

the user intended for the input vector to be fetched in its entirety, or whether the user is constructing a form of feedback loop where output values are utilized as inputs. In order to resolve this ambiguity, the scratchpad interface adheres to a simple rule: *for an individual FPGA computation, input and output data streams are isolated until the computation completes.* As a consequence feedback loops are not permitted within a computation.

The RC-1000 implementation utilizes simple mechanisms to guarantee that an output data stream does not overwrite either of the input data streams. In this system a FIFO buffer is utilized to cache the outputs generated by circuits until all inputs data values are read from the scratchpad. Once a circuit notifies the frame that it has read all input values, the scratchpad interface will begin streaming data values out of the FIFO into memory. Currently the system utilizes the FIFO for any computation that utilizes the scratchpad for both input and output.

The FIFO approach has both positive and negative aspects. From a positive perspective the FIFO satisfies the scratchpad rule and also simplifies the state machines for the scratchpad controllers. This simplification is based on the fact that the controller performs read operations in their entirety until write operations begin. As for negative aspects, the system cannot overlap reads and writes and therefore suffers in performance. Additionally, a computation is limited in size to the capacity of the FIFO (currently 1,024 32-bit words). This system can be improved through a more sophisticated implementation which determines where outputs and inputs overlap and dynamically manages these regions.

B.4.3 Virtual Scratchpad Memory

FPGA cards have a limited amount of physical, on-card memory that can be utilized to house scratchpad data. Additionally, it is possible that multiple applications may utilize the FPGA card at the same time and require different sets of data be stored in the scratchpad. Therefore, it is beneficial if the scratchpad interface provides a means of transparently managing scratchpad memory for different applications as needed by the runtime environment. This form of

management is similar to traditional *virtual memory* operations found in modern operating systems.

Reserved (3)	Virtual Page Number (8)	Reserved (2)	Page Word Offset (19)

Figure B.7: The fields of a scratchpad virtual memory reference.

The RC-1000 implementation provides a simple virtual memory system for on-card scratchpad memory. SRAM banks 1 and 2 on the FPGA card serve as 2 MB page frames that house application data. The host in this system maintains a database of scratchpad pages that are not in use. When a page is swapped out of a card's page frame, it is buffered in host memory until it is required again by the FPGA. In order to facilitate the virtual memory system, all references to scratchpad memory must be based on the 29-bit virtual memory address shown in Figure B.7. The bottom 19-bits of this address provide an offset into a scratchpad page. Since SRAM banks are 32-bits wide, the bottom 2-bits of an 21-bit memory reference are always zero. The top 8-bits of the virtual address refer to a unique virtual page number. A memory location in on-card SRAM is used to identify the pages numbers that are currently loaded in the page frames. The host must update this location any time a page is loaded or swapped out. The FPGA pulls this information into internal registers after being reset or when signaled by the host. At runtime the frame compares the scratchpad references in a message header to the IDs of the currently loaded pages. If the message's scratchpad requirements can be satisfied with the currently loaded pages, the frame establishes all of the necessary data paths and configures both of the scratchpad controller units.

If the frame detects that an incoming message cannot be processed due to missing scratchpad pages, the frame must perform a page fault operation. The first step in this process is for the frame to store the missing page numbers in an SRAM memory location, halt operation, and notify the host through the status register. After the host detects the page fault it loads the missing page number(s) from SRAM. The host performs the necessary evictions, copying one or

two of the scratchpad pages from the SRAM banks to the host memory page database. One or more scratchpad pages can then be copied to the card to satisfy the request. The host completes the operation by updating the page number values in the card's SRAM and triggering a resume operation with the control register. The frame loads the new page numbers and continues processing the message that originally caused the page fault.

B.5 GRIM Function Calls

Multiple function calls have been added to the GRIM communication library to simplify the amount of effort a user must perform to utilize the computational units connected to an FPGA frame. These function calls are layered on top of existing GRIM calls and format the arguments of the message header to match what is required by the FPGA frame. Host-level functions are listed in Table B.2.

Table B.2: GRIM API for interactions with the Celoxica RC-1000 card.

grim_celoxica_send(	u32	grim_resource_id,
	u16	function,
	u8	function_sub_op,
	u8	pipeline_stage
	u8	C, A, B
	u32	write_addr,
	u32	Read0_addr
	u32	Read1_addr or constant
	u16	calculation_words
	u16	payload_words,
	u32*	payload_starting_address)
grim_celoxica_ldMem(	u32	grim_resource_id,
	u32	card_memory_address,
	u16	number_words,
	u8*	payload_starting_address)
grim_cleoxica_setPipeline(	u32	grim_resource_id,
	u8	pipeline_id,
	u8	next_pipeline_id,
	u8	next C, next A, next B,
	u32	next_write_addr,
	u32	next_read0_addr,
	u32	next_read1_addr_or_const
	u16	next_calculation_words)
grim_celoxica_killPipeline(	u32	grim_resource_id,
	u8	pipeline_id)
grim_celoxica_alu_local(	u32	grim_resource_id,
	u8	alu_op,
	u32	dest_addr,
	u32	src0_addr,
	u32	src1_addr,
	u16	calculation_words)

These API functions are briefly described as follows. The grim_celoxica_send() function is used to send a user-defined active message to a specified RC-1000 endpoint. This function allows users to set all of the frame parameters for the outgoing message and is utilized by other calls in this API. The grim_celoxica_ldMem() function is designed as a simple means of loading data from an application into the scratchpad. The grim_celoxica_setPipeline() and grim_celoxica_killPipeline() calls are used to set and clear the forwarding registers of an RC-1000 endpoint. Finally, the grim_celoxica_alu_local() function is used to transmit a message to

an RC-1000 endpoint to perform an operation using the frame's built-in ALU using scratchpad memory references.

B.6 Debugging Infrastructure

One of the more challenging aspects of working with FPGA cards is the process of debugging an application. Because an endpoint software is implemented as hardware in the FPGA, it is difficult to observe what exactly an FPGA is doing at any particular time. Therefore, it is beneficial to construct environments that can be utilized to assist circuit designers in debugging their applications.

B.6.1 Simulation Environment

Simulation is by far the quickest and most revealing method by which hardware designers can debug their FPGA applications. Modern hardware description language (HDL) packages such as Active-HDL provide a graphical environment where a design's signals can be traced in a cycle-by-cycle manner. Because these tools take design input from the same HDL files that are used for synthesis, there is a relatively high-level of confidence that fundamental design errors can be caught by designers in simulation.

In order to assist users in the debugging process, a test bench has been constructed that simulates the card-specific units of the Celoxica RC-1000. This environment allows the user to examine how an FPGA design will perform when utilized on the card. The test bench simulates the SRAM banks, SRAM arbitration signaling, clock generation, and the control/status registers of the RC-1000. The benefit of this environment is that users can load the SRAM banks with application data to determine how an FPGA will react to different settings. The common mode of operation is to store multiple GRIM formatted messages in the incoming message queues of SRAM bank 0 to examine the device's reaction.

As a means of automating the simulation debugging process, a special library was constructed to allow GRIM formatted messages to be captured directly by applications and statically passed to the simulator. This library was written as a functional replacement for the GRIM library and simply records data that a host application attempts to inject into the Celoxica card. The recorded information is then used to generate data files that can be read into the simulation environment. While this process is relatively simple and static, it has been found to be an exceptionally useful means of debugging FPGA applications.

B.6.2 Localized Debugging

After a designer has thoroughly examined an FPGA design in simulation, the next step is to examine the design with actual hardware. This process can be complicated because it is difficult to extract useful information from the live hardware. In order to assist users in this task, instrumentation code is added to the basic framework of the FPGA design. This infrastructure provides eight data signals that are routed to an LED on the RC-1000 card. These signals are commonly utilized to display both frame state information and provide a heartbeat display to indicate that the card has not locked up. Users can attach information to these signals to observe internal information. In a similar manner, users can route debug information out of the FPGA using the RC-1000's user I/O pins or the control/status registers.

Due to the complexity of monitoring FPGA signals, the most common means of debugging a non-simulated FPGA on the RC-1000 card is accomplished by simply monitoring information stored in on-card SRAM. In this approach, a user constructs a host level application that injects a series of messages into a local RC-1000 device (using a local card simplifies the communication process and reduces the number of locations that errors can be generated). After the messages are injected, a user can use multiple command line tools written for GRIM to probe the card's memory. The tool dump_celoxica utilizes information about the card's memory layout to provide useful information such as the card's front and back message queue pointers. Other

tools have been constructed to allow users to clear card memory and reset and load the FPGA with a user-specified configuration file.

B.7 Future Frame Work

The RC-1000 frame presented in this appendix represents a first generation implementation of an interface that allows an FPGA resource to be integrated into a cluster computing environment. This work is beneficial because it provides a portable API that both circuit designers and application programmers can reference. Additionally, the frame simplifies the amount of effort an end user must perform when operating with the FPGA because the frame implements a large amount of commonly required functionality. There are multiple ways in which this first-generation work can be improved.

B.7.1 Enhancements to the RC-1000 Frame

The RC-1000 implementation can be enhanced in multiple ways. The primary weakness of the current frame is that its memory interface only allows for sequential reads of vector data ports. It is often desirable for more random access mechanisms to be utilized in these interfaces, as it allows user circuits to fetch and store data as needed by the application. A simple extension for such operation would be to add a "skip" signal to each vector port. This signal could be designed to allow a circuit to instruct the memory interface to skip ahead in the vector data stream by a variable number of words. This signal only requires the addition of a small number of lines and therefore should not significantly impact the routing of the frame. From an application perspective it is more desirable for the user-defined circuit to be able to specify the offsets given to the memory units as needed. However, this approach has a number of drawbacks. Primarily, it requires the addition of a large number of signals for each vector port of each circuit. Second, it increases the amount of work each user circuit must perform in managing its vector data streams.

Finally, it is difficult to pipeline the fetching of data from memory in this approach as data can be accessed in a random fashion.

A second area of improvement for the frame is in the chaining of vector units. While the current frame allows a series of computations to take place within a single FPGA, it stores intermediate results between computational stages in a recycling buffer. This approach simplifies the frame's complexity but results in a store-and-forward form of computation. It is desirable to implement a system that is similar to the traditional chaining operations found in high-performance supercomputers. In this approach, data could be routed directly from one unit to another without significant intermediate buffering. The hardship of this approach is designing a memory interface that can satisfy all requests for data efficiently. After the first stage, each computational unit is likely to add another vector data tap to the memory interface. In this work it would be interesting to compare the design of a frame that could route data between multiple stages to a system where designers simply created custom FPGA images that statically connected a series of stages and presented the pipeline as a whole to the end user.

Finally, the FPGA frame could be enhanced to allow better concurrency in the processing of messages. In the current implementation the frame operates only on a single message at a time. As a result both user-defined circuits and parts of the frame are placed in idle states as other units complete their portion of the message processing task. A more efficient system would allow these idle units to begin processing the next message in order to create a better pipeline of operations. A simple approach would allow the frame to begin processing the next message as soon as it finishes handing off data to computational units. A more complex approach would involve scheduling multiple messages to be processed by different user-defined circuits. As with chaining, this approach requires additional constraints on the memory system that might be difficult to implement in a practical manner.

B.7.2 Future Work with Other FPGA Cards

Another important area for future work is adapting the frame to operate on different FPGA cards. Conceptually, the frame serves as a means of insulating end users from the card-specific characteristics of a commercial FPGA peripheral device. This environment allows designers to construct complex computational circuits that can easily be incorporated as processing units in the cluster architecture. Adapting the frame to operate on different FPGA cards allows circuit designers to easily migrate a computational circuit from one FPGA card to another, without having to redesign the circuit.

The main challenge in adapting the frame to operate on different FPGA platforms is addressing the architectural characteristics of each card. At a high level most FPGA cards exhibit similar architectures. In general an FPGA and PCI controller share access to a large block of on-card memory. However, there is a wide amount of variety in the manner in which each of these units are connected depending on the card. For example the Celoxica RC-1000 card provides four independent memory banks. This feature was heavily exploited in the implementation of the RC-1000. Adapting the frame to operate on an FPGA card with only a single memory bank would require the frame be modified to multiplex memory transfers onto a single memory channel. For a card with more than four memory banks, it is possible that the frame could be enhanced so that the additional banks are utilized to house extra scratchpad pages to reduce paging.

Another aspect of a card's architecture that affects the design of the frame is the hardware that provides communication with the host. The RC-1000 card uses a custom 8-bit control/status port for simple communication with the host. The frame uses these registers to notify the host of runtime events, such as a page or function fault or the presence of outgoing messages. This method of communication is inefficient and utilized only because the card cannot initiate a DMA transfer. A more desirable means of communication is for the card to DMA information into the host's memory. In general most peripheral devices allow this form of operation. Therefore, while

porting the frame to a new FPGA card will require the construction of mechanisms to support card to host communication, it is expected that these mechanisms will be more flexible than those found in the RC-1000.

In summary, adapting the FPGA frame to operate on different cards is a challenging but beneficial task. This work involves translating card-specific operations from one card to another. While the unique hardware environment of each peripheral card prevents the frame from simply being moved from one platform to the next, it is expected that the functionality from the RC-1000 implementation can serve as a guide to constructing a frame for other architectures. Performing such adaptations can be extremely beneficial because they allow a user's designs to be easily moved between platforms.

APPENDIX C

THE GRIM API

GRIM is a communication library that allows designers to easily construct applications that utilize a cluster's distributed resources. GRIM provides a relatively simple but powerful API that feature two separate programming interfaces that can be utilized concurrently by an application. The first of these interfaces is for active messages. In active messages a sender specifies a function handler that is invoked at the receiver when the message arrives. This interfaces allows users to trigger actions at remote endpoints and is particularly well suited for interactions with remote peripheral devices. The second interface provided by GRIM is one for remote memory transactions. These operations allow an endpoint to send or fetch a block of memory from a remote endpoint. Remote memory transactions are useful for efficiently moving large blocks of memory in the cluster with minimal overhead. GRIM is constructed as a linkable C library that can optionally support POSIX threads. Users can tailor GRIM's behavior through the manipulation of configuration files without having to recompile the application. This appendix describes the basic characteristics of the GRIM API.

C.1 Configuration Interface

GRIM utilizes a small number of configuration files to specify the hardware environment for the virtual parallel processing machine. There are two types of files utilized to specify the configuration. First a single application configuration file serves as a top-level means of configuring the system. This file contains basic high-level information such as the number of nodes to use for the application and the configuration of each node. It is expected that end users will commonly adjust this file to meet the needs of the application. The second category of

configuration files is for static information that does not frequently change in the cluster. These files define routing tables for the cluster as well as available hardware resources.

C.1.1 Application Configuration

From a user's perspective there is one configuration file that is central to specifying how the GRIM environment is defined. At initialization time GRIM reads a file specified by the GRIM_CONFIG environment variable to determine its configuration. In the current release this configuration file is located in grim/config/grim_config. This file contains the following variables:

- **NUM_NODES**: This specifies how many hosts are available in the system.
- **LCP_FILE**: This variable specifies a file that contains the routing information for the Myrinet network.

After these initial constraints users define the configuration for their cluster, including the resources that are to be utilized in each host. A cluster must be defined with a cluster name, a configuration file for the cluster, and a list of resources to be utilized in the cluster. For example a cluster with two hosts (paris and metz) with a Celoxica card could be listed in the configuration file as:

```
#--The French Cluster--
CLUSTER                 French_cluster
CLUSTER_RESOURCE        French_cluster_config.txt
ROUTING_MYRI            grim_routes_myri.txt

HOST paris
      USE MYRI
      USE CELOXICA

HOST metz
      USE MYRI
```

It is possible for multiple cluster configurations to be listed in the configuration file in order to allow users to easily migrate an application between different clusters without adjusting the configuration file. Before GRIM parses this file it determines the name of the host that it is operating on. It then selects the proper cluster to use from the configuration file by selecting the configuration that contains the host the program is running on.

C.1.2 Cluster Resource Configurations

GRIM utilizes multiple configuration files to specify various information for cluster resources. The application configuration file specifies the location of each of these files. GRIM parses the resource configuration files based on whether the resources are utilized by the cluster. The following files are utilized:

- **Myrinet Routing Table**: Because Myrinet uses source routing it is necessary to define all of the routing information for the cluster in advance. Routes must be selected in a deadlock free manner. Each line in the configuration specifies the source node and the paths to all other nodes in the cluster.

- **Myrinet Host Mapping File**: This file contains the listing of hosts utilized in the cluster and the physical ID of each host. The physical ID is the number that is referenced with the Myrinet routing table because it reflects the port number in which the host is physically connected to the Myrinet switches.

- **Celoxica Circuit File**: The software that manages the Celoxica card requires information describing the FPGA configuration images that can be loaded into the FPGA card. This file lists all of the available FPGA configuration files and notes which circuit is loaded in which user-defined computational slot. Incorporating additional user-defined circuits in GRIM requires that the circuits be identified in the grim_handlers.h header file.

C.2 Initialization: grim_enable()

The GRIM communication library is initialized through the `grim_enable()` procedure. This function performs a number of startup operations for the library and should only be called once by an application. This function must be called before any of the library's variables or functions can be accessed. The `grim_enable()` procedure performs the following internal operations:

- **Reset variables**: The GRIM library begins operation by allocating and resetting all variables. These variables include node information as well as various databases.
- **Parse configuration files**: Each node in GRIM must load information from configuration files to determine information about the global cluster environment. This information contains both general information (e.g., routing tables and host names) and application-specific information (e.g., the number of hosts used in application).
- **Construct a local database of resources**: Each node in the cluster utilizes configuration file information to construct a database for resources in the cluster. This database

contains both local and global information, and is referenced by end applications to determine the location of requested resources.

- **Initialize local devices**: The next step for a node in the cluster is to initialize the local set of peripheral devices that are to be utilized in the cluster environment. These initializations are device specific, performing operations such as loading a peripheral device with firmware.
- **Allocate host incoming message queues**: Next GRIM interacts with a device driver to open a block of memory that is both pinned and contiguous. The library obtains both a virtual and physical address for the memory so that it can be accessed by the user-space application and the peripheral devices. This memory is utilized to house incoming message queues.
- **Initialize and link message queues**: After all local peripheral devices and host memory is available, GRIM must establish message queues and initialize message queue pointers for the resources in the local host. The size of each message queue is based on the amount of memory available in the resource to house messages and the number of queues that must share this space. Users can request particular sizes for message queues in the configuration files, although this information is ignored if it exceeds queue capacities. After queues are allocated, the library stores pointer information in the appropriate outgoing message queue registers for each resource.
- **Handler library initialization**: Next, GRIM initializes both the local and global handler databases. Once initialized the database is loaded with a set of built-in function handlers that are available at all nodes.
- **NI synchronization**: All of the NIs for the nodes used in the cluster perform synchronization. In this procedure a NI alerts all other NIs that it is reset and waiting to hear from the other nodes. A NI must wait until it receives a reset notification from all

other NIs before it can proceed in a normal mode of operation. This process is necessary in order to reset the sequencing information used by NI pairs and to guarantee that a NI will not transmit data to a node that has not been initialized.

- **Peripheral device activation**: The final step in initialization is to notify all peripheral devices that the node is fully initialized and that operation in the cluster is to begin.

Once `grim_enable()` completes, host applications can begin utilizing the library.

C.3 Runtime Information

After initialization, users can obtain basic information about the host an application is running on. Each host in the cluster is assigned both a physical node number (PNN) and a virtual node number (VNN). The PNN is a constant number that is assigned to a particular host. It is utilized internally by the library to manage routing tables. The VNN is a number that is assigned to a host at runtime based on the cluster's configuration file. GRIM assigns VNNs linearly to hosts in the configuration file, starting with VNN 0 for the first host in the file. This approach allows users to easily specify which nodes in the cluster are utilized in the virtual machine, without having to change the files containing physical routing information. End users should always reference host nodes with the VNN in applications for portability.

Table C.1: API for obtaining cluster host information.

u32 id	=	grim_getVNNFromName(string name)
string name	=	grim_getVNNName(u32 id)
u32 id	=	grim_getMyVNN()
string name	=	grim_getMyName()
		grim_printVNNConfiguration()

Users can obtain basic information about the local host an application is running on through the commands listed in Table C.1. The first four of these functions provide either a 32-bit VNN identifier for a host or the host's name. The last command prints out information about the runtime configuration of hosts utilized in the system for an application.

C.3.1 Referencing a Cluster Resource

In addition to referencing hosts in the cluster, users must be able to reference individual communication endpoints. A reference to a communication endpoint is constructed with three pieces of information: the VNN of the endpoint's host, the type of communication device the endpoint is (e.g., host-CPU level, Celoxica card, I_2O card, etc.), and a logical channel identifier to associate transmissions to the endpoint.

Table C.2: Functions for generating a reference to a communication endpoint.

u32 =	grim_getDestID(	u32	destination node VNN,
		u8	destination device,
		u16	logical channel)
u32 =	grim_getResource(	u32	destination node VNN,
		u8	device type,
		u16	logical channel)

As listed in Table C.2, GRIM provides two functions for generating a reference to a communication endpoint. First, `grim_getDestID()` can be utilized to obtain a 32-bit reference to an endpoint if the location of the endpoint is known in advance by the user. The user must supply the VNN and device type of the endpoint, as well as the desired logical channel for communication with the endpoint. This function is useful for referencing well-known endpoints, such as the host-level endpoint that is available at every node. The second function for generating a reference to an endpoint is `grim_getResource()`. This function is designed to allow users to query the cluster's resource database in order to locate a desired resource. Users can specify a particular VNN with which to restrict the search, or specify that any VNN can be utilized.

C.4 Active Message Interface

The first of two programming interfaces provided in the GRIM communication library is for active message style interactions with endpoints. In this system the sender includes information in each message that specifies how the message is to be processed at the destination endpoint. Each endpoint contains a set of active message function handlers that are used to

process incoming messages. An endpoint must register a handler with the communication library before the handler can be utilized by other endpoints in the cluster. During registration users associate a string identifier with a handler and are returned an integer identifier that can be utilized to reference the handler in subsequent API calls. Endpoints invoke actions in other endpoints simply by sending active messages that are properly encoded. Each endpoint is responsible for periodically invoking a polling operation that extracts messages from incoming message queues and processes the messages accordingly.

C.4.1 Handler Registration

Each communication endpoint in the cluster can be equipped with a different set of active message handlers. Therefore, it is necessary to provide functions in the cluster that allow endpoints to register their own unique handlers and publish this information to the global cluster context so that other endpoints can reference and utilize the handlers. This process takes place in three phases with the functions listed in Table C.3

Table C.3: The functions utilized to register and reference active message function handlers.

	grim_register_handler(handler_call_t function, string name)
	grim_syncHandlers()
u32 =	grim_resolveHandler(string name)

The first part of handler registration takes place when an endpoint registers its function handlers locally with the `grim_register_handler()` function. With this function an endpoint updates a local table that associates a string identifier with a virtual memory pointer to a function handler. This information does not leave an endpoint until the `grim_syncHandlers()` function is executed. The `grim_syncHandlers()` function transmits the local list of function handlers to an endpoint in the cluster which is responsible for managing a global database of function handlers for the cluster. This node will merge the incoming list of handlers into the global database and transmit a copy of the global list of

handlers to the endpoint requesting the synchronization. The requesting endpoint will then update its local tables and assign a global integer identifier to every local function handler. Once equipped with this information, an endpoint can call the `grim_resolveHandler()` function to determine the global integer identifier of a function handler in the cluster. This ID can be utilized in active message transmissions with other endpoints.

C.4.2 Send and Receive Operations

As listed in Table C.4, GRIM provides a `grim_send()` operation for transmitting an active message to a destination and a `grim_poll()` function for receiving and processing incoming messages. In the `grim_send()` function the sender must provide a resource reference id for the destination endpoint, the handler id, four active message arguments, and an optional payload. The GRIM communication library will take this message, transfer it in its entirety (performing segmentation and reassembly if necessary), and execute the function at the receiving endpoint using the information provided.

Table C.4: The functions for sending and receiving active messages.

grim_send(	u32	destination resource id,
	u16	function handler id,
	s32	handler argument 0,
	s32	handler argument 1,
	s32	handler argument 2,
	s32	handler argument 3,
	u16	payload length,
	u32*	payload starting address)
grim_poll()		

The `grim_poll()` function must be performed regularly at the receiving endpoint in order to facilitate the processing of active messages. This function detects that new messages are available for processing in the endpoint's incoming message queues, extracts the message, and performs the specified computation. When POSIX threads support is enabled in GRIM, it is not necessary for end applications to execute `grim_poll()` operations. Instead a thread is dedicated to periodically performing the poll operation.

C.5 Remote Memory Interface

The second programming interface provided in the GRIM communication library is one for interacting with memory located at a remote node. Because host applications and peripheral devices operate with different address spaces (i.e., virtual and physical), the remote memory API must be specific about the types of addresses that it operates with as well as provide mechanisms for translating between address spaces. In GRIM the primary means of referencing a block of memory for end applications is a virtual memory address. Like many communication libraries that perform remote memory operations, GRIM users are only allowed to utilize virtual address spaces that are created with special allocation function calls. These calls allocate and pin the requested memory regions and supply the NI with memory translation information. The GRIM library also provides mechanisms for remote memory transactions with physical addresses. These mechanisms operate with low overhead because address translation is not necessary, but offer no protection if a user supplies the mechanisms with bad addresses.

C.5.1 Managing Memory

The first part of the remote memory interface is designed to assist users in managing memory that can be accessed by the NI. The first call listed in Table C.5 is the `grim_malloc_pinned()` function. This function interacts with GRIM's pinned memory device driver to allocate a sufficient block of memory that is guaranteed to be pinned. Internally GRIM allocates large blocks of contiguous memory and then allocates memory requests from these blocks in order to simplify the number of virtual to physical memory address translation entries loaded in the NI. The result of this function is a virtual address that an application can use as a regular pointer and pass as a reference to other endpoints in the system. When an application finishes using a block of pinned memory it can call the `grim_free_pinned_va()` call to free the allocation.

Table C.5: Function calls for managing memory used by the remote memory interface.

u8*	=	grim_malloc_pinned(	u32	number of bytes)
		grim_free_pinned_va(	u8*	pinned virtual address)
u8*	=	grim_get_pinned_pa(	u8*	pinned virtual address)
u8*	=	grim_get_pinned_va(	u8*	pinned physical address)

In addition to allocating and freeing pinned memory, the GRIM library provides mechanisms for translating addresses. The `grim_get_pinned_pa()` function translates the virtual address of a block of memory allocated by `grim_malloc_pinned()` into a physical address. This address can be used for physical memory transactions that bypass address translation in the NI. The `grim_get_pinned_va()` similarly can be utilized to translate a physical address reference of a pinned memory block into a virtual address. This function is primarily provided for completeness.

C.5.2 Remote Memory Operations

GRIM is designed to provide both send and fetch operations for interactions with a remote endpoint's memory. Both of these operations can be supplied with an optional lock to provide a simple form of notification. A lock is simply the virtual address of a 32-bit integer that is allocated from GRIM's pinned memory. In the `grim_sendMemory()` call the sender specifies the ID of the destination endpoint, the source and target virtual addresses of the block of data to transfer and the size of the block of memory. If a virtual address of a lock is supplied, the receiving NI will DMA the value specified in the call's lock value variable into the address after the entire block of memory has been transferred. The `grim_fetchMemory()` function operates in a similar manner, but transfers data from the remote endpoint to the local endpoint. Once the operation completes the NI of the node initiating the fetch operation will transfer a zero into the local endpoint's lock address, if specified.

Table C.6: Functions for transferring data between an endpoint and a remote endpoint.

grim_sendMemory(	u32	destination resource id,
	u32*	target virtual address,
	u32*	source virtual address,
	u16	number of bytes)
grim_sendPhysicalMemory(	u32	destination resource id,
	u32*	target physical address,
	u32*	source virtual address,
	u16	number of bytes)

GRIM also supports an interface for transferring data directly to a physical address in the remote endpoint's system. When the NI of the receiving endpoint receives such a message it transfers the data directly without any form of virtual memory translation, as none is needed. While this mechanism allows for the operation to take place without the overhead of translation, users must be aware that there is no memory protection employed with this function. Supply erroneous information to this function can easily cause the NI to write data into an unknown memory address in the system, which is likely to crash the host system. The physical memory interface however is useful for interacting with devices such as a video display device's frame buffer.

C.5.3 Reserving NI Memory

Additional functions are provided in the GRIM library to allow NI memory to be directly utilized by applications. While it is expected that most applications do not need such functionality, it is conceivable that some applications may need a temporary place to store data close to the wire (e.g., supporting a frame buffer in the NI). The first function listed in Table C.7 is `grim_reserveNIMemory()` and is designed to allocate a block of memory in the NI. This function **must** be called **before** the `grim_enable()` function is called. After `grim_enable()` is called the `grim_getReservedNIMemory()` function can be utilized

to gain both virtual and physical memory address pointers to the region of memory allocated in the NI.

Table C.7: Functions for reserving memory in the local NI card.

grim_reserveNIMemory(u32 size)		
grim_getReservedNIMemory(	u8**	virtual memory address,
	u8**	physical memory address)

C.6 Multicast

GRIM provides support for multicast and broadcast operations. The current implementation is tree based and performs message replication in the NI cards. A multicast tree is referenced in GRIM by a string and an integer value. Once a multicast tree is defined an endpoint can subscribe or unsubscribe from its data distribution. All endpoints in the cluster are allowed to inject messages into a multicast tree.

C.6.1 Multicast Tree Management

Table C.8 lists the API provided in GRIM for managing the multicast distribution trees. The function `grim_findMulticastTree()` is used to determine a unique integer value to reference a particular multicast tree in the cluster. If the requested tree name has not been referenced, the library constructs a new tree and assigns ownership of the tree to the endpoint that first attempted to locate the tree. After a tree has been identified an endpoint can specify that it whishes to be a part of the tree and subscribe to multicast traffic. An endpoint can use the `grim_subscribeMulticast()` function if a multicast tree has been identified or the `grim_subscribeMulticastByName()` function if the tree id is not yet known. In both cases the use must specify which NI logical channels will be subscribing to the multicast messages. The channel list is a Boolean mask. The `grim_unsubscribeMulticast()` function is utilized to remove the endpoint from the multicast distribution tree.

Table C.8: Functions for managing multicast distributions.

u32 =	grim_findMulticastTree(	string	name)
u32 =	grim_subscribeMulticase(	u32	multicast id,
		u8	channels)
u32 =	grim_subscribeMulticaseByName(		
		string	name,
		u8	channels)
	grim_unsubscribeMulticast(	u32	multicast id,
		u8	channels)

C.6.2 Sending Multicast and Broadcast Messages

Table C.9 lists the functions utilized to inject messages into the broadcast and multicast trees. These functions are identical to the normal active message send functions except that the `grim_sendMC()` function requires the specification of the multicast tree id instead of the destination endpoint, and the `grim_broadcast()` does not require the specification of a destination endpoint. All endpoints are allowed to utilize these functions, whether they subscribe to a multicast tree or not. Any active message function handler can be utilized with these functions, although users must be aware that the sending of a multicast active message may result in the invocation of the function handler at multiple nodes. Therefore, users must be cautious when designing active message function handlers that are intended for multicast operations. In the current implementation, the original sending endpoint identification information is not accurately provided to the endpoint invoking the function handler. Therefore, if such information is required it is necessary to include it in one of the active message function arguments that are passed with the message.

Table C.9: Functions for injecting multicast and broadcast messages.

grim_sendMC(	u32	multicast id,
	u16	function handler id,
	s32	handler argument 0,
	s32	handler argument 1,
	s32	handler argument 2,
	s32	handler argument 3,
	u16	payload length,
	u32*	payload starting address)
grim_broadcast(	u16	function handler id,
	s32	handler argument 0,
	s32	handler argument 1,
	s32	handler argument 2,
	s32	handler argument 3,
	u16	payload length,
	u32*	payload starting address)

C.7 Advanced API Functions: TPIL

As a means of provided accelerated performance for applications injecting data into peripheral devices a library has been constructed for x86 host endpoints named TPIL: the tunable PCI injection library. TPIL provides basic mechanisms for transferring data to a PCI device using hardware units that have evolved in the x86 architecture. When users need to add new peripheral devices to the GRIM communication library, it may be beneficial to utilize the TPIL API in order to enhance performance.

Table C.10: The TPIL API for accelerating injections of data into a peripheral device from a host CPU.

Tdev =	tpil_create(	u32	device_file_id,	
		u8*	device_mmap,	
		u32	mmap_size,	
		u32	device_ioctl	)
	tpil_h2c(	Tdev*	tpil_device,	
		u8*	destination,	
		u8*	source,	
		u32	number_bytes	)
Tcfg =	tpil_benchmark(	Tdev*	tpil_device	)
	tpil_configure(	Tdev*	tpil_device,	
		Tcfg*	tpil_configuration	)

The function `tpil_create()` is utilized to initialize TPIL and generate a reference that can be utilized in subsequent API calls. This function provides pointers to the file handler of

the device and its memory map, the size of the memory map, and a reference to the ioctl() functions that TPIL can utilize to transfer data with the assistance of a kernel driver. The ioctl() calls are optional and are a means of utilizing a card's DMA engines. The `tpil_h2c()` function is utilized to perform host-to-card transfers of data. The user must supply a pointer to where data is to be written (i.e., a pointer to somewhere in the card's memory map), a pointer to the data that is to be injected, and the size of the transfer. TPIL will utilize internal information to determine which transfer mechanism is the best option.

TPIL provides mechanisms for configuring how transfers are performed. First the `tpil_benchmark()` function can be utilized to examine the characteristics of the host machine. Generally this operation is performed offline as it performs several lengthy tests to determine how long it takes to inject data into a peripheral device. The results of these measurements can be exported for use in other programs. The `tpil_configure()` function is utilize to configure the transfer mechanisms for a particular device. Users can either utilize the information obtained from the benchmarking operations or supply custom settings. These settings are simply the cutoffs in which different transfer mechanisms are employed.

REFERENCES

[1] R. Russell, "The Cray-1 Computer System," in *Communications of the ACM*, Vol. 21 No. 1, January 1978.

[2] Meiko World Inc., "Computing Surface 2: Overview Documentation Set," 1993.

[3] R. Mendez, High Performance Computing, John Wiley and Sons Ltd., Chichester, UK, UK, 1992.

[4] N. Uchida, M. Hirai, M. Yoshida, K. Hotta, "Fujitsu VP2000 Series," in *Proceedings of Compcon*, 1990.

[5] A. Padegs, B. Moore, R. Smith, W. Buchholz, "The IBM System/370 Vector Architecture: Design Considerations," in *Transactions on Computers*, Vol. 37, No. 5, May 1988.

[6] J. Dongarra, H. Meuer, and E. Strohmaier. "TOP500 Supercomputer Sites," Presented at the *International Supercomputer Conference*, June 2002.

[7] M. Yokokawa, "Present Status of Development of the Earth Simulator," in *Proceedings of Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2001.

[8] T. Blank, "The MasPar MP-1 Architecture," in *IEEE Compcon*, Spring 1990.

[9] D. Hillis, The Connection Machine, MIT Press, Cambridge, 1985.

[10] Intel Corporation, "Paragon XP/S Product Overview," 1991.

[11] C. Leiserson, et. al., "The Network Architecture of the Connection Machine CM-5," in *Journal of Parallel and Distributed Computing*, Vol. 33, No. 2, 1996.

[12] W. Oed, "The Cray Research Massively Parallel Processor System CRAY T3D," Technical Report, Cray Research, November 1993.

[13] Beowulf Computing Website: http://www.beowulf.org

[14] D. Becker, T. Sterling, D. Savarese, J. Dorband, U. Ranawak, and C. Packer, "Beowulf: A Parallel Workstation for Scientific Computation," in *Proceedings of the International Conference on Parallel Processing*, 1995.

[15] M. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina, "Virtual Memory Mapped Network Interfaces," in *IEEE Micro*, February 1995

[16] H. Dietz, T. Chung, T. Mattox, and T. Muhammad, "Purdue's Adaptor for Parallel Execution and Rapid Synchronization: The TTL_PAPERS Design," Purdue University School of Electrical Engineering, Technical Report, January 1995.

[17] PCI-SIG, "PCI Local Bus Specification 2.2," 1998.

[18] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," in *IEEE Micro*, Vol.15, No.1, 1995.

[19] R. Horst and D. Garcia, "Servernet SAN I/O Architecture," in *Proceedings of the Hot Interconnects Symposium V*, August 1997.

[20] *Scalable Coherent Interconnect,* IEEE Standard 15961992, 1992.

[21] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology," in *IEEE Micro*, Vol. 22, No. 1 January/February 2002.

[22] R. Metcalfe and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," in *Communications of the ACM*, Vol. 19, No. 5, July 1976.

[23] IEEE 802.3ae Standard: 10Gb/s Ethernet.

[24] A. Betz and P. Grav, "Gigabit Over Copper Evaluation," draft paper available at http://www.cs.uni.edu/~gray/gig-over-copper, April 2002.

[25] IEEE Standard 896: Futurebus+.

[26] Dolphin Interconnect corporate web page, http://www.dolphinics.com June 2002.

[27] Dolphin Interconnect Performance Benchmarks web page, http://www.dolphinics.com/benchmarks.html June 2002.

[28] C. Seitz and W. Su, "A family of routing and communication chips based on the Mosaic," in *Proceedings of the Washington Symposium on Integrated Systems*, 1993.

[29] R. Felderman, D. Cohen, A. DeSchon, and G. Finn, "ATOMIC: A High-Speed Local Communication Architecture," in *Journal of High Speed Networks*, Vol. 3, No. 1, 1994.

[30] W. Dally and C. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," in *IEEE Transactions on Computing*, Vol. 36, 1987.

[31] Pittsburgh Supercomputer Center Press Release, "Terascale Computing System Installed at PSC", October 1, 2001.

[32] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[33] B. Chun, A. Mainwaring, and D. Culler, "Virtual Network Transport Protocols for Myrinet," in *Proceedings of Hot Interconnects Symposium V*, April 1997.

[34] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," in *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1995.

[35] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication," in *Proceedings of the 12th International Parallel Processing Symposium*, March 1998.

[36] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and K. Verstoep, "Performance of a High-Level Parallel Language on a High-Speed Network," in *Journal of Parallel and Distributed Computing*, Vol. 40, No. 1, February 1997.

[37] K. Yocum, J. Chase, A. Gallatin, and A. Lebeck, "Cut-through Delivery in Trapeze: An Exercise in Low Latency Messaging," in *Proceedings of IEEE Symposium on High-Performance Distributed Computing*, 1997.

[38] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li, "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication," in *Proceedings of Hot Interconnects V*, August 1997.

[39] Myricom, Inc. The GM message layer, http://www.myri.com

[40] L. Prylli and B. Tourancheau, "BIP: A New Protocol designed for High-Performance Networking on Myrinet," in *Proceedings of the 12th International Parallel Processing Symposium*, 1998.

[41] R. Martin, A. Vahdat, D. Culler, and T. Anderson, "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[42] T. von Eicken. "Active Messages: an Efficient Communication Architecture for Multiprocessors," Ph.D. Thesis, 1993.

[43] A. Birrell and B. Nelson. "Implementing Remote Procedure Calls," Technical Report CSL-83-7, Xerox Palo Alto Research Center, October 1983.

[44] R. Bhoedjang, T. Ruhl, and H. Bal, "User-Level Network Interface Protocols," in *IEEE Computer*, Vol. 31, No. 11, 1998.

[45] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," in *International Journal of Supercomputing Applications*, Vol. 8, 1994.

[46] R. Riedel, C. Faloutsos, G. Gibson, D. Nagle, "Active Disks for Large-Scale Data Processing," in *IEEE Computer*, Vol. 34 No. 6, June 2001.

[47] D. Nagle, G. Ganger, J. Butler, G. Goodson, and C. Sabol, "Network Support for Network-Attached Storage," in *Proceedings of Hot Interconnects*, 1999.

[48] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. "WireGL: A Scalable Graphics System for Clusters," in *Proceedings of SIGGRAPH*, 2001.

[49] C. Cruz-Neira, D. Sandin, and T. Defanti, "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE," in *Proceedings of SIGGRAPH*, 1993.

[50] R. Rheinheimer, J. Beiriger, H. Bivens, S. Humphreys, "The ASCI Computational Grid: Initial Deployment," Los Alamos Technical Report LA-UR-01-3602.

[51] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A Survey of Active Networks Research," in *IEEE Communications Magazine*, Vol. 35, No.1, 1997.

[52] D. Culler, K. Keeton, C. Krumbein, L. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa, "Generic Active Message Interface Specification", Technical Report, Computer Science Division, University of California at Berkeley.

[53] InfiniBand Trade Association, "InfiniBand Specification 1.0a," 2001.

[54] A. Bonhomme and P. Geoffray, "High Performance Video Server using Myrinet," in *Proceedings of First Myrinet User Group Conference*, 2000.

[55] B. Goglin and P. Geoffray, "High Bandwidth Data Transfer with OPIUM & Myrinet: Application to Remote Video," Internship with Myrinet Technical Report, 2001.

[56] A. DeHon, "Comparing Computing Machines," in *Proceedings of SPIE*, Vol. 3527, 1998.

[57] J. Villasenor and W. Mangione-Smith, "Configurable Computing," in *Scientific American*, June 1997.

[58] W. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, H. Spaanenburg, "Seeking Solutions in Configurable Computing," in *IEEE Computer*, Vol. 30 No. 12, December 1997.

[59] M. Gokhale et. al., "SPLASH: A Reconfigurable Linear Logic Array," in *Proceedings of the International Conference on Parallel Processing*, August 1990.

[60] M. Jones, L. Schard, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[61] K. Yao, "Implementing an Application Programming Interface for Distributed Adaptive Computing Systems", a Masters Thesis at the Virginia Institute of Technology, May 2000.

[62] A. Tanenbaum, Computer Networks, Third Edition, New Jersey: Prentice Hall, 1996.

[63] J. Ngai and C. Seitz, "A Framework for Adaptive Routing in Multicomputer Networks," in *Proceedings of First Symposium on Parallel Algorithms and Architectures*, 1989.

[64] Intel Corporation, "Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual," Intel document 248966-04, 2001.

[65] S. Thakker and T. Huff, "The Internet Streaming SIMD Extensions," in Intel Technology Journal, Q2, 1999.

[66] R. Bhoedjang, T. Ruhl, and H. Bal. "User-Level Network Interface Protocols," in *IEEE Computer*, Vol. 31, No. 11, 1998.

[67] American National Standard ANSI/VITA 26-1998, "Myrinet-on-VME Protocol Specification," 1998.

[68] Cyclone Microsystems website: http://www.cyclone.com.

[69] I_2O Special Interest Group, "Intelligent I/O (I_2O) Architecture Specification version 2.0," 1999.

[70] M. Fiuczynski, R. Martin, T. Owa, and B. Bershad, "SPINE – A Safe Programmable and Integrated Network Environment," in *Proceesings of the Eighth ACM SIGOPS European Workshop*, 1998.

[71] Intel Corporation, "i960 Rx I/O Microprocessor Developer's Manual," 1997.

[72] Wind River Systems, Inc., "VxWorks Programmer's Guide 5.3.1," 1997.

[73] Wind River Systems, Inc., "VxWorks Tornado User's Guide 1.0," 1995.

[74] R. West, R. Krishnamurthy, W. Norton, K. Schwan, S. Yalamanchili, M. Rosu, and V. Sarat, "QUIC: A Quality of Service Network Interface Layer for Communication in NOWs," in *Proceedings of Heterogeneous Computing Workshop*, 1999.

[75] Embedded Solutions, Ltd. "RC1000-PP Hardware Reference Manual," product data sheet, 1999.

[76] Xilinx Corporation, "Xilinx and Conexant Announce Licensing Agreement of Skyrail 3.125 Gbps Serial Transceiver Technology," Corporate Press Release, 2000.

[77] Xilinx Corporation, "Virtex-II Field-Programmable Gate Array," data sheet, 2002.

[78] Xilinx Corporation, "Virtex Field-Programmable Gate Array," data sheet, 2002.

[79] IEEE Standard 1386.1-2001.

[80] PLX Technology, Inc., "The PCI 9080 Data Book," version 1.06. January 2000.

[81] National Bureau of Standards, "Data Encryption Standard," Federal Information Processing Standards Pub. 46, 1977.

[82] D. Kessner and the Free IP Project, http://www.free-ip.com/DES/index.html

[83] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6 Block Cipher," 1998.

[84] R. Rivest. The md5 message digest algorithm, RFC 1321, 1992.

[85] Brooktree Division, Rockwell Semiconductor, Inc., "Bt848/848A/849A Single-Chip Video Capture for PCI," datasheet, 1997.

[86] Conexant Systems, Inc., "The (Almost Definitive) FOURCC Definition List," available at: http://www.fourcc.org.

[87] Video-4-Linux Documentation, available in Documentation/video4linux directory of Linux kernel.

[88] Intel Corporation, "Accelerated Graphics Port Interface Specification, Revision 2.0," 1998.

[89] A. Nye, X Protocol Reference Manual, O'Reilly & Associates 1994.

[90] X. Lin and L. NI, "Multicast Communication in Multicomputer Networks," in *Proceedings of the International Conference on Parallel Processing*, 1990.

[91] R. Sivaram, D. Panda, and C. Stunkel, "Multicasting in Irregular Networks with Cut-Through Switches using Tree-Based Multidestination Worms," in *Lecture Notes in Computer Science*, Vol. 1417, 1998.

[92] N. Mir, "A Survey of Data Multicast Techniques, Architectures, and Algorithms," in *IEEE Communications*, Vol. 39, No. 9, September 2001.

[93] M. Schroeder, B. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Sattethwaite, and C. Thacker, "Autonet: A High-Speed, Self-Configuring Local Area Network using Point-to-Point Links," DEC Technical Report SRC research report 59, 1990.

[94] R. Casafo, F. Quiles, J. Sanchez, J. Duato, "Deadlock-Free Routing in Irregular Networks with Dynamic Reconfiguration," in *Proceedings of the Third International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, 1999.

[95] R. Kesavan and D. Panda, "Efficient Multicast on Irregular Switch-Based Cut-Through Networks with Up-Down Routing," in *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 8, August 2001.

[96] F. Silla and J. Duato, "Improving the Efficiency of Adaptive Routing in Networks with Irregular Topology," in *Proceedings of the International Conference on High Performance Computing*, 1997.

[97] D. Comer, Internetworking with TCP/IP: Principles, Protocols, and Architecture, Prentice-Hall, Third Edition, 1995.

[98] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi and J. Philbin, "User-space communication: A quantitative study," in *Proceedings of the High Performance Networking and Computing Conference*, 1998.

[99] S. Rodrigues, T. Anderson, and D. Culler, "High-Performance Local Area Communication With Fast Sockets," in *Proceedings of Usenix 1997 Conference*, 1997.

[100] Alteon Networks, "Gigabit Ethernet Technology Brief," 1998.

[101] Intel Corporation, "Intel PRO/1000 MT Server Adaptor", product brief, 2002.

[102] Intel Corporation, "Intel IXP1200 Network Processor," white paper, 2000.

[103] Xilinx Corporation, "Virtex-II Pro Platform Field-Programmable Gate Array," data sheet, 2002.