The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

COMMUNICATION AND SCHEDULING IN CLUSTERS: A USER-LEVEL PERSPECTIVE

A Thesis in

Computer Science and Engineering

by

Shailabh Nagar

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2001

We approve the thesis of Shailabh Nagar.

Date of Signature

Anand Sivasubramaniam
Associate Professor of Computer Science and Engineering
Thesis Adviser, Chair of Committee

Chita R. Das
Professor of Computer Science and Engineering

Mary J. Irwin
Professor of Computer Science and Engineering

Matthias A. Blumrich
Research Staff Member,
IBM T.J. Watson Research Center
Special Member

Natarajan Gautam
Assistant Professor of Industrial Engineering

Dale A. Miller
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Abstract

Cluster computing has become the preferred model for building high performance servers today. The rapid advances and commoditization of microprocessors, network hardware and system software has made it possible for clusters to be built from off-the-shelf components. However the performance of such clusters is hampered by the software overheads of communication and scheduling at each node. The improvement of cluster communication and scheduling performance are important areas of research for modern computer systems.

This thesis explores the use of user-level networking for improving cluster communication. By using intelligent network interface cards and optimized implementations of well-designed protocols, user-level networking (ULN) can significantly reduce communication overheads. The thesis examines the design and implementation of user-level network interfaces from a software and hardware perspective. It studies the benefits of ULN in clusters and ways to improve the scalability of ULN implementations. It also investigates how quality of service can be efficiently provided in a ULN. User-level communication has been used to improve the scheduling of tasks across cluster nodes. The thesis examines communication-driven coscheduling mechanisms and their impact on parallel application performance. It is shown that communication and scheduling are closely related and overall cluster performance depends on optimizing the two subsystems together.

Table of Contents

List of Tables ix

List of Figures x

Chapter 1. Introduction 1

1.1 Efficient, scalable user-level communication 3

1.2 Communication-driven scheduling of parallel processes 6

Chapter 2. pSNOW 8

2.1 pSNOW 11

2.2 Communication Mechanisms for NOW 13

2.2.1 Network Interfaces 14

2.2.2 Communication Substrates 15

2.3 Implementation and Performance of Communication Substrates . . . 17

2.3.1 (C1,NI1) 17

2.3.2 (C1,NI2) 20

2.3.3 (C2,NI3) 21

2.3.4 (C3,NI3) 23

2.3.5 Validation 26

2.4 Microbenchmark and Application Performance 26

2.4.1 Ping_Pong and Ping_Bulk 28

2.4.2 FFT 28

2.4.3 Matmul 30
2.4.4 IS 32
2.5 Impact of Network Processor Speed 33
2.6 Summary 38

Chapter 3. MU-Net 40
3.1 Myrinet 41
3.2 MU-Net 42
3.2.1 User-level Library 43
3.2.2 MU-Net Kernel Drivers 44
3.2.3 MU-Net MCP 44
3.2.4 Details of MU-Net Operations 45
3.2.4.1 Creating an Endpoint 45
3.2.4.2 Sending a Message 48
3.2.4.3 Receiving a Message 52
3.2.4.4 Destroying an Endpoint 54
3.3 Performance Results 54
3.4 Summary 57

Chapter 4. Scalability of VIA 59
4.1 The VI Architecture 61
4.1.1 Objectives and overview of VIA 61
4.1.2 VIA Operations 63
4.2 Scalability Considerations for VIA 66

4.2.1 Memory management 67
4.2.2 Work and Completion Queues 68
4.2.3 Firmware design 71
4.2.4 NIC hardware 74
4.2.5 Application/Library 76
4.2.6 Host hardware 77
4.3 Performance Evaluation 77
4.3.1 Simulator and Workload 77
4.3.2 Messaging Sequence 81
4.3.3 Base Design: Need for a scalable solution 83
4.3.4 Size of Descriptor Payload 85
4.3.5 Effect of Completion Queues (CQ) 86
4.3.6 Pipelined/Overlapped firmware design 87
4.3.7 Hardware DMA Queues 90
4.3.8 Hardware Doorbell support 91
4.3.9 Tailgating Descriptors 94
4.3.10 Separate Send and Receive Processing 95
4.3.11 Shadow Queues 97
4.3.12 Putting it all together 98
4.4 Summary 100

Chapter 5. Quality of Service for VIA 101
5.1 Issues in providing QoS aware communication 103

5.2 Overview of ULN and NIC Operations 106
5.2.1 Doorbells 107
5.2.2 HDMA Operation 108
5.2.3 NSDMA and NRDMA operation 109
5.3 Firmware Design 110
5.3.1 Sequential VIA (SVIA) 111
5.3.2 Parallel VIA (PVIA) 111
5.3.3 QoS-aware VIA (QoSVIA) 112
5.4 Performance Results 115
5.4.1 Results from Experimental Platform 116
5.4.1.1 Raw Performance of VIA Implementations 116
5.4.1.2 Performance with CBR Channels (QoS) 120
5.4.2 Simulation Results 122
5.4.2.1 Results for Higher Loads 123
5.4.2.2 VBR QoS Channels 126
5.4.2.3 Hardware NSDMA Queue 126
5.5 Summary 128

Chapter 6. Communication-driven scheduling 130
6.1 Scheduling Strategies 133
6.1.1 Local Scheduling 136
6.1.2 Spin Block (SB) 138
6.1.3 Dynamic Coscheduling (DCS) 140

6.1.4 Dynamic Coscheduling with Spin Block (DCS-SB) 142

6.1.5 Periodic Boost (PB and PBT) 143

6.1.6 Periodic Boost with Spin Block (PB-SB) 145

6.1.7 Spin Yield (SY) 146

6.1.8 Dynamic Coscheduling with Spin Yield (DCS-SY) 147

6.1.9 Periodic Boost with Spin Yield (PB-SY) 147

6.2 Performance Results 149

6.2.1 Experimental Setup and Workloads 149

6.2.2 Comparison of Scheduling Schemes 153

6.2.3 Discussion 161

6.3 Summary 164

Chapter 7. Conclusions 167

7.1 Future Work 170

7.2 The Ideal Network Interface 172

References 174

List of Tables

2.1 Simulation Parameters 13

3.1 Comparison of Roundtrip Latencies (in μs) 55

3.2 Anatomy of a Message in μs (Effect of message size and multiple endpoints) 56

5.1 Break-up of time expended in different operations during message transfer for different message sizes. Times are in microsecs, and indicate the time when that operation is completed since the send doorbell is rung for the send side, and receive doorbell is rung for the receive side. Receive descriptors are pre-posted before the data actually comes in for PVIA and QoSVIA, and their DMA transfers do not figure in the critical path. 117

5.2 Average 1-way Message Latency in microsecs using Ping-Bulk experiment 119

6.1 Design space of scheduling strategies 134

6.2 Four Process Mixed Workloads (% of time spent in communication is given next to each application) 151

6.3 Two Process Mixed Workloads 153

6.4 Completion Time in Seconds (Workloads 1 to 5) 153

6.5 Completion Time in Seconds (Workloads 6 to 9) 156

6.6 Completion Time in Seconds (Workloads 10 to 12) 158

6.7 Individual Completion Times for Workloads 3 and 5 (in secs) 158

List of Figures

2.1 1-way Latency for (C1,NI1) 19

2.2 1-way Latency for (C1,NI2) 20

2.3 Comparison between (C1,NI1) and (C1,NI2) 22

2.4 1-way Latency for (C2,NI3) 23

2.5 1-way Latency for (C3,NI3) on a TLB hit 25

2.6 Performance of Microbenchmarks 29

2.7 Performance of FFT 30

2.8 Performance of Matmul 32

2.9 Performance of IS 33

2.10 Comparison of (C2,NI3) and (C3,NI3) 35

2.11 Impact of NP Speed 37

3.1 Mapping of Endpoint into User Address Space 46

3.2 Anatomy of a MU-Net Operation 48

3.3 Structure of a MU-Net Packet 49

4.1 Components of the VI Architecture 63

4.2 Baseline NIC 67

4.3 Sequence of Actions in Sending a Message 81

4.4 Message latency and wait timesBase (u=8, 128 bytes) 83

4.5 Message latency and wait timesBase (u=minimum(60,d), 4KB) 83

4.6 Descriptor Payload Size (Base, u=d) 85

4.7 CQ benefits (Base, u=1, 128 bytes) 87

4.8 Base+O vs. Base (4KB) 88

4.9 Base+O vs. Base (128 bytes) 88

4.10 Base+O+DMAQ vs. Base+O (4KB) 91

4.11 Base+O+DMAQ vs. Base+O (128 bytes) 92

4.12 Door+O+DMAQ vs. Base+O+DMAQ - Latency (128 bytes) 92

4.13 Door+O+DMAQ vs. Base+O+DMAQ - Wait time (128 bytes) 93

4.14 Tailgating (Door+O+DMAQ, 128 bytes) 94

4.15 Door+O+DMAQ+2CPU vs. Door+O+DMAQ (128 bytes) 96

4.16 DMA vs. MMIO using Shadow Queues (Door+O+DMAQ+2CPU, 128 bytes) 97

4.17 Door+O+DMAQ+2CPU vs. Base (128 bytes) 98

4.18 Door+O+DMAQ+2CPU vs. Base (1KB) 98

4.19 Door+O+DMAQ+2CPU vs. Base (4KB) 99

5.1 Myrinet NIC 106

5.2 Experimental Results Using 1 sender and 3 receivers with three classes (1, 2 and 3 MBytes/s) of QoS channels, with each receiver receiving one class. The θ shown is for the 2 MBytes/s class. 121

5.3 For the same experiment as Figure 5.2, this graph shows the percentage of packets experiencing jitters for PVIA and QoSVIA. 122

5.4 Simulation results for a 1-sender 4-receivers configuration with 3 QoS classes (2, 8 and 32 MBytes/s) and 1 BE channel (33 MBytes/s). The load on the BE channel, and on any QoS channel remains constant. The total injected load is varied by increasing the number of QoS channels. *All the graphs are plotted as a function of total injected load and not as a function of load on that channel class.* The line for 32 MBytes/s channel with SVIA is not shown because the system has already saturated at these loads. 123

5.5 Simulation results for a 4-senders 1-receiver configuration with 3 QoS classes (2, 8 and 32 MBytes/s) and 1 BE channel (33 MBytes/s). . . . 125

5.6 Simulation results for 2 machines sending messages back and forth on 3 QoS channel classes (2, 8 and 32 MBytes/s) and 1 BE channel (33 MBytes/s) SVIA for the 32 MBytes/s channels has already saturated at these loads. 125

5.7 Simulation results with MPEG-1 traces for the 1 sender 4 receivers configuration with 3 QoS classes (0.95, 1.9 and 3.8 MBytes/s) and 1 BE channel (16.7 MBytes/s) 127

5.8 Simulation results with Hardware NSDMA for the 1 sender 4 receivers configuration with 3 QoS classes (2, 8 and 32 MBytes/s) 127

6.1 Software components used in the implementations 135

6.2 Effect of scheduling policy on one-way latency 148

6.3 Monitoring CPU Utilization for Workload 5 159

6.4 Monitoring CPU Utilization for Workload 3 160

Chapter 1

Introduction

Cluster computing has come of age. Initially proposed as a cost-effective form of parallel computing, clusters of workstations are becoming the preferred model for building high-performance, high-availability servers which are the cornerstones of the Internet today. The rapid advances in microprocessors, high-speed networks and system and application software are enabling high-performance clusters to be assembled from commodity-off-the-shelf (COTS) components. In this thesis, the term *cluster* is used to represent a collection of commodity workstations connected by a high-speed network.

While being cost-effective, loosely coupled systems such as clusters, do suffer in terms of performance especially when compared to traditional supercomputers. The software which ties together the cluster, both at the system level (operating systems and networking libraries) and at the application level (APIs) has not been able to sufficiently exploit the offered hardware performance. In particular, the communication and scheduling subsystems of clusters provide challenging problems for system architects interested in building commercially viable platforms. The often conflicting requirements of high-performance, and the need to use unmodified system software as far as possible, provide a rich source of research issues.For instance, traditional networking stacks rely on the operating system for providing protection and buffering for messages. The

overhead of doing so limits the extent to which they can use high-speed networking hardware for fine-grain communication with other cluster nodes. COTS operating systems make scheduling decisions based on local criteria alone and do not take into account the cluster-wide context for some of the processes under their control (even if such information is available from these processes). Introducing these cluster-wide criteria for process scheduling is often achievable only at a considerable overhead.There are several other areas where the lack of efficient interfaces between the local operating system and cluster-wide resource managers makes the cluster perform more poorly than necessitated by the node hardware.

This thesis targets two major areas of clusters : (a) the communication subsystem linking the cluster nodes and (b) the scheduling subsystem controlling the processes running on each node. Both these subsystems are interrelated and need to be examined together to deliver the best performance to applications running on the cluster. The focus of the thesis is mainly on the systems software that manages the resources of the cluster - in particular, the ones which control the network interface and process scheduling. Naturally, the design choices of this low-level software depend on the hardware that it manages. Consequently, some hardware options are examined and proposed as well. There is a fundamental tradeoff between single-program performance and the utilization of cluster resources. Single-program performance is greatest when every cluster resource is immediately available for use any time the program needs it. However, from a utilization point of view, it is more important that service requests for a resource be evened out and match the service rate of the resource manager. It is the aim of this thesis to study some of these tradeoffs in greater detail.

The next two sections describe the five major contributions of this thesis : four of these pertain to the communication subsytem and one to scheduling of processes on the cluster.

1.1 Efficient, scalable user-level communication

The physical network connecting the workstations in a cluster is capable of delivering bandwidths exceeding 1 Gigabit per second, with per-hop latencies less than 1 microsecond and very low error rates. However, the delivered performance of the communication subsystem, using traditional networking protocol stacks such as TCP/IP, is much worse with latencies ranging to hundreds of microseconds. The discrepancy is caused by the high overheads of the software layers involved in messaging, both at the application level (in the form of bulky messaging libraries) and at the operating system level (due to context switches and copying).

Researchers have proposed user-level networking as one way of reducing this overhead. In this approach, the operating system kernel is eliminated from the critical path of message sends and receives. User-level processes are allowed direct access to the network interface without compromising on protection. User-level network interfaces (ULNs) such as Hamlyn [20], U-Net [93], FM [72], AM [94], Shrimp [15], PM [89], BIP [63], Trapeze [100], LFC [13] and BDM [51] have demonstrated the benefits of this approach.The computer industry has also realized the potential of ULNs and is standardizing it in the form of the Virtual Interface Architecture (VIA) specification [25].The standard is still evolving with efficient implementations, capable of meeting the needs of legacy applications, posing challenging problems.

This thesis begins, in Chapter 2, by investigating the issue of user-level communication in the context of the cluster as a whole. It looks at different kinds of network interface architectures that can be used for cluster communication. A simulation tool, called pSNOW, is used to evaluate and compare some of these options. The evaluation of some of the major network interface designs shows that the proposed benefits of user-level communication are indeed seen in a wide range of workloads. Besides, the simulation tool is a contribution in itself, allowing a uniform comparison of newly proposed designs.

Having established the merits of user-level networking through simulations, the second part of the thesis (Chapter 3) addresses the design and implementation issues on a real network interface card (NIC). Several design tradeoffs for efficient, low-latency, high-bandwidth user-level networking are examined in detail. The Myrinet NIC used in the exercise, is quite representative of the neworking hardware currently available both in terms of features and performance. The work done in this part of the thesis has resulted in a ULN implementation that performs comparably to its counterparts in other projects.

One of the first issues that any viable cluster solution has to address is that of scalability. Since the performance improvements of clustering are predicated on increasing the number of nodes, it is vital that all components of the cluster scale well with this increase. For the network architecture, scalability translates to being able to deliver the same performance to an increasing amount of network traffic (measured either in terms of messages or connections). The third part of the thesis (Chapter 4) addresses this issue for ULN's in the context of the Virtual Interface Architecture (VIA). The simulation-based

study examines the impact of several NIC design issues from the standpoint of scalability. Based on these, it proposes and examines new hardware and software mechanisms for improving the scalability of VIA.

Another aspect of networking architectures which hasn't received adequate attention in the context of clusters, is quality of service (QoS). In networking, the term refers to the fulfillment of message latency and bandwidth guarantees negotiated by an application. This is important for two reasons. First, for some applications, predictability of communication is more important than its raw performance. Such applications (notably multimedia) are becoming increasingly important for clusters. Second, it allows well-behaved applications, which consume a fair share of networking resources, to be protected from their ill-behaved counterparts.This enables better network resource management which is particularly important for the growing cluster sizes seen today. QoS issues have been extensively studied for wide-area networks (WANs) and to some extent for LANs. But for clusters and ULN, its a new and challenging area. The challenge comes from the absence of a central, trusted kernel, which is in a much better position to regulate an application's usage of networking hardware. Furthermore, the performance compromised has to be kept to a minimum so that user-level networking is still attractive vis-a-vis traditional networking. The fourth part of the thesis (Chapter 5) looks at the provision of QoS guarantees in VIA. Using a combination of simulation and measurement, it shows how communication guarantees can be met by restructuring the firmware that runs on the network interface cards.

Overall, these four parts of the thesis deal with various aspects of deploying user-level networking in clusters. The final part of the thesis addresses a closely related issue, parallel process scheduling.

1.2 Communication-driven scheduling of parallel processes

The move from closely-coupled multiprocessor machines such as the Cray T3E and Intel Paragon to cluster supercomputers built with off-the-shelf hardware and operating systems not only results in increased communication overheads for parallel programs but also requires coordinated scheduling of the parallel processes on individual nodes. A poor scheduling strategy can easily nullify any benefits gained by the use of user-level networking. Consider a request-response scenario where the sender of a message awaits a reply from the receiver. If the receiver process on the remote node is not scheduled when a message for it arrives, the sender has to wait till the receiver gets scheduled and is able to send a reply (even if we disregard application specific skew between the send and reply parts of the communicating processes).A receiving mechanism which does not take into account remote node scheduling is likely to be inefficient in terms of processor utilization and application progress.

The problem of efficient scheduling of the processes of a parallel job is not new, but it resurfaces under a different set of constraints in a cluster environment with user-level networking. Coordinating the scheduling of parallel processes, without such support inbuilt into the commodity operating systems running on each node, is an interesting problem. One approach is to use information from the communication layer to alter the

scheduling of processes. Dynamic coscheduling and implicit coscheduling are two forms of this approach that have been proposed in the past.

In addition to implementing and evaluating the benefits of communication-based scheduling strategies, the final part of the thesis (Chapter 6) proposes new strategies to enhance the performance seen by parallel programs. A comprehensive testbed implementing the Message Passing Interface (MPI) over U-Net (a user-level networking layer) has been developed on Solaris over a cluster of Sun workstations . Five new scheduling schemes have been proposed and the results show that some of these schemes offer significant performance benefits over the hitherto proposed strategies [68].

The rest of the thesis is organized as follows. Each of chapters 2 through 6 outlines one major component of the thesis, as described above. Chapter 7 summarizes the contributions of the thesis and outlines directions for future research.

Chapter 2

pSNOW

There have been numerous studies addressing efficient systems software and architectural support for parallel applications on clusters. But there is a dearth of general purpose tools to uniformly evaluate the performance of these environments and to guide the future effort in efficient architectural designs, and this chapter addresses an important void in tools for evaluating clusters.

The large overhead in communicating between the processing nodes of parallel machines is often the significant limiting factor in the performance of many applications. This problem is exacerbated when we move from tightly coupled multiprocessors to more loosely coupled cluster environments. There are three main issues to be addressed for lowering the communication overheads on such environments. First, the network connecting the workstations should be fast. Second, the interface to the network should provide an efficient way of transferring data from the memory on the workstation onto the network. Finally, the software messaging layers should add minimal overhead to the cost of moving data between two application processes running on two different workstations. Recent research has been addressing these issues. High speed networks such as ATM [32] and Myrinet [16] can potentially deliver point-to-point hardware bandwidths that are comparable to the link bandwidths of the interconnection networks in multiprocessors. To address the second issue, newer network interface designs have been proposed with

hardware enhancements to overlap communication with computation and to facilitate direct user-level interaction with the network interface [93]. Finally, low latency software messaging layers have been proposed [94, 72] making use of these network and network interface innovations.

Today, we have a wide spectrum of choices in terms of hardware for the computing engines, the networks, and the network interfaces to put together a cluster. There have been isolated studies that have focussed on developing efficient software messaging mechanisms for a particular hardware but it is difficult to generalize the results from such studies. For example, von Eicken et al. [93] develop a mechanism called Unet wherein, the processor on the Fore Systems ATM card examines the message coming in from the network and transfers it directly to a user-level buffer without the main CPU intervention. However, for the Myrinet interface, Pakin et al. [72] show that the LANai processor is not powerful enough to examine incoming packets. As a result, the delivered bandwidth goes down whenever the LANai processor is employed for additional tasks. Mukherjee et al. [67] suggest using coherent network interfaces to lower the traffic on the system memory and I/O buses, and present a taxonomy and comparison of four such interfaces. However, their study does not look at the software overheads of the messaging layers. It would be extremely useful to evaluate different network interface designs and to examine different software solutions for these interface designs. To our knowledge, the work by Krishnamurthy et al. [59] is the only study that has attempted to compare and evaluate the different network interface designs for global address-based communication. Though this study [59] provides valuable insight on the systems studied, the problem is that the comparison may not be uniform. Since the comparison is between systems

with different (in terms of the vendor, the technology, and the capabilities) computing engines, network interfaces, and networks, it is difficult to generalize the results.

The primary motivation for this part of the thesis stems from a need to have a unified framework to compare and evaluate different network interface designs and different communication software solutions for a cluster. More specifically, we would like to answer a range of questions for the system architect to design better communication mechanisms and to adapt to emerging trends/technologies:

- What fraction of the total communication overhead is spent in the hardware and what fraction is spent in software? Further, what is the breakdown of the overhead in each hardware and software component? How much of this is unavoidable and how much can be overlapped with useful computation in the application? With this information, where should future resources (in terms of effort and money) be directed to reap the maximum rewards?

- What should be done by the host CPU and what should be done by the network interface? Can we quantify this relegation of tasks based on the relative speeds of the network interface processor and the main CPU?

- If the processor on the network interface card were to become much faster, would the Unet [93] approach perform better or should we adhere to the Fast Messages [72] approach? At what network processor speed does one become better than the other?

- Recently, it has been suggested that we use a multiprocessor workstation (Symmetric Multiprocessor or SMP) and dedicate one of its CPUs for communication

purposes. What is the performance benefit with this strategy, and how efficiently are workstation resources being utilized [40] ?

- How many CPUs on an SMP can a network interface handle without significant degradation in performance? Given a finite number of CPUs, should we have a fewer number of SMPs each with a larger number of CPUs, or a larger number of SMPs each with a fewer number of CPUs [99] ?

In this part of the work, we present an execution-driven simulator, pSNOW (pronounced "snow"), that can be used to guide the design and evaluation of architecture and systems software support for a cluster.pSNOW helps us simulate the execution of applications on a cluster for a spectrum of network interface hardware and communication software alternatives, using the Active Messages (AM) layer [94] as the message passing interface.

2.1 pSNOW

pSNOW is an execution-driven discrete event simulator running on a SPARCstation that we have developed to: uniformly compare the different software and hardware communication capabilities for clusters, get detailed performance profiles on the execution, and vary architectural parameters to answer some of the questions raised in the beginning of this chapter. The input to pSNOW are applications written in C++, that use the Active Messages [94] interface for communicating between the parallel activities.

As with several recent parallel architecture simulators [81, 17, 31, 92, 75], the bulk of the application instructions are directly executed to avoid high simulation costs. Only the explicit calls that an application makes to the Active Messages interface are

simulated. The application codes are compiled to assembly, augmented with cycle counting instructions, assembled to binaries, and then linked with the rest of the simulator modules. When the application makes an Active Messages call, the simulation time is reconciled with the cycles taken for executing the application code since the previous call.

pSNOW provides a process-oriented view to implement the details of a node architecture and has been implemented on top of the CSIM [66] simulation package. It has a library of generic classes that model a CPU, memory, buses, and Network Interfaces, and default member function implementations. These generic classes can be instantiated for specific hardware artifacts such as the main CPU, the network interface processor (NP), memory on workstation, memory on the network interface (NI) card, memory bus, I/O bus etc., and the member functions overloaded/customized to model specific actions for the instance. This modularization has made it extremely easy for us to model different network interfaces and to modify hardware features (such as making the node architecture an SMP). We feel that this tool will be valuable as newer network interfaces and newer node architectures continue to emerge.

We have simulated the network interfaces (discussed in detail in the next section) on pSNOW and developed the communication software substrates for these interfaces. The Active Messages layer has been implemented on top of these substrates. Both the communication substrates and the Active Messages layer have been cycle counted to account for the software overheads. For the discussions in this chapter, we model an ATM network between the nodes of the cluster. Since our focus at this point is on understanding network interfaces, their interaction with the node architecture, and

the communication software, we have not simulated ATM switches for the performance results in this paper assuming that these switches would not add significant overheads to the network latency. However, it is easy to model switches and connectivities without altering the implementation of the current node architecture.

Table 2.1 summarizes some of the simulation parameters in host CPU cycles for the performance results in this paper, unless they are explicitly varied for a particular experiment.

Cache Access	1 cycle
Memory Access	6 cycles
NI Access	12 cycles
NP clock	2 cycles
Network Bandwidth	155 Mbits/sec
Memory Bus	80 MBytes/sec
I/O Bus	50 MBytes/sec
Page Size	4096 bytes

Table 2.1. Simulation Parameters

2.2 Communication Mechanisms for NOW

In this section, we review network interface designs, both past and present, and communication software designs for these interfaces. Krishnamurthy et al. [59] give an overview of the network interfaces found on five hardware platforms, namely the CM-5 [90], the Cray T3D [30], the Meiko CS2 [10], the Intel Paragon [55], and the Berkeley NOW which is a cluster of UltraSPARCs connected by Myrinet. As mentioned earlier, we would like to refrain from discussing the details of any specific machine/vendor, since

we are interested in a uniform comparison of different designs. Instead, we present three different categories of network interface cards (the interfaces found on the above machines may fall in one of these categories) plugged into the I/O bus of a workstation, and discuss possible communication software designs for these three categories.

2.2.1 Network Interfaces

The three categories of network interfaces (NIs) that we consider are: a network card with no intelligence (no NP) and no DMA support (NI1); a network card with no intelligence (no NP) but provides DMA capabilities (NI2); and a network card with a NP and with DMA capabilities (NI3). The capabilities of the network interface card thus progressively improves as we move from NI1 to NI3. There are newer designs which have proposed special hardware-based virtual memory mapped network interfaces [14] and coherent network interfaces [67], but we are not considering such interfaces in this study. NI1 is representative of older interfaces, such as the one found on the CM-5 and the older Fore Systems TCA-100 ATM adapter [43]. On these interfaces, the main CPU has to explicitly read/write data from/to the receive/send FIFOs. This overhead will particularly dominate for large data transfers. NI2 is representative of the interfaces on a slew of parallel machines such as the nCUBE/2 [70] as well as on many LAN interface cards such as Ethernet. The advantage over NI1 is that the CPU can use the additional hardware (the DMA) to pick/deposit the data to send/receive from/to the memory while it can do useful work. However, DMA controllers normally work only with physical addresses. To prevent the DMA send/receive buffers from being paged out, usually these buffers are created in kernel space and pinned in physical memory.

Additional copying needs to be done to/from these buffers. Finally, NI3 is representative of the newer network interfaces such as the Fore Systems SBA-200 ATM interface [44], the Myrinet interface [16], and the interfaces found on some parallel machines such as the Intel Paragon [55]. On these interfaces, apart from DMA capability, there is also a network processor (NP) that can be programmed. This helps move some of the processing to the NP while the main CPU can indulge in useful work. As we shall soon observe, this NP also provides the ability to perform data transfer to/from the user address space directly without involving the operating system on the main CPU, thus lowering context switches, interrupts and copying costs.

2.2.2 Communication Substrates

On these three NIs, we consider three low-level communication substrates (C1, C2, C3). These communication substrates support a reliable basic send/receive mechanism, incorporate packetization for long messages, and provide protection between multiple user processes. Higher-level programming interfaces such as Active Messages [94] will be provided on top of these substrates for the applications.

The first substrate (C1) is a kernel-level messaging layer ie. the user program has to invoke the send/receive mechanism of the kernel for all data transfers to/from the network. Consequently, the communication will involve context switches to the kernel, and additional copying of data between the kernel and user address spaces. The advantage with C1 is that protection between user processes is fairly straightforward to implement. C1 is thus similar to the normal socket mechanism found on UNIX systems

in LAN environments. We consider C1 for NI1 (where the kernel will directly operate the send/receive FIFOs) and NI2 (where the kernel will program the DMA).

The second substrate (C2) that we consider is a user-level messaging layer, where a user-level library implements the basic send/receive mechanism. Note that, we cannot implement C2 on NI1 and NI2 since that would compromise on protection. Potentially, a user process could examine an incoming message of another process if the network interface were mapped into the user space. Hence we do not consider this option. However, as the Unet [93] approach showed, we can implement a user-level layer for NI3. For each user process, we can reserve a communication segment in its own address space. This segment is pinned in physical memory. For an outgoing packet, the NP (running kernel mode code) can examine communication segments of user processes and send them out into the network. Similarly, for an incoming message, the NP can examine the header, figure out which user process needs to receive it, and can DMA it in to the communication segment of this user process.

One of the problems with C2 is that communication segments of user processes need to be pinned in physical memory which can lead to inefficient use of the physical memory. Further, the data to be sent/received needs to be copied to/from the communication segment from/to the eventual destination address in the user space. It would be extremely useful if the network interface could work with virtual addresses instead of the physical addresses (as is done with C2) to avoid these problems. The network interface processor (NP) could look up the operating system page table to perform the virtual to physical mapping before performing the data transfer. In fact, recently looked up page table entries can be cached (similar to how the TLB caches page table entries

for the main CPU) on the NP's local memory that is provided on most of the recent NI3 type cards [44, 16]. These entries can be updated/invalidated when the operating system on the main CPU evicts pages from physical memory. We refer to this communication substrate as C3, and this can be implemented on only the NI3 interfaces.

To summarize, in this exercise, we focus on three kinds of network interfaces (NI1, NI2, NI3) and three low-level communication substrates (C1, C2, C3). The implementation details and performance comparisons for the four combinations that we consider (C1,NI1), (C1,NI2), (C2,NI3), and (C3,NI3) are given in Section 2.3.

2.3 Implementation and Performance of Communication Substrates

In this section, we describe the implementation of the different communication substrates (C1, C2, C3) on the three network interfaces (NI1, NI2, NI3) discussed in Section 2.2, and the performance of these substrates. As we mentioned, we present results from four combinations (C1,NI1), (C1,NI2), (C2,NI3), and (C3,NI3). These combinations are compared using a simple experiment to ping-pong messages between 2 nodes using the send/receive mechanisms provided by the substrates, assuming a 100 MHz host CPU.

2.3.1 (C1,NI1)

This combination is studied as a worst case example since the network interface (NI1) is the most primitive of the chosen three, and the communication software is also the most inefficient. The only two capabilities provided by the network interface are pushing data out from a send FIFO onto the network, and assembling incoming messages into a receive FIFO and raising an interrupt when the message is received.

The main CPU on the workstation has to copy data from/to the host memory to/from the send/receive FIFO of the network interface card. Since the CPU directly manipulates the network FIFOs which are potentially shared between multiple application processes at a node, the send/receive mechanisms for (C1,NI1) necessitate execution in the kernel mode.

Figure 2.1 shows a detailed profile of the execution path of a message from the time an application process invokes the send call until the time the message is received by the application process at the other end for the (C1,NI1) combination. On a send system call, the CPU pushes the data to be sent from the host memory onto the send FIFO if the NI is free. If not, it queues this request and will be subsequently informed (interrupted) by the NI when the device becomes free. On the arrival of a message, the NI interrupts the main CPU, which would incur the cost of switching to an interrupt service routine (ISR) that runs in kernel mode. The ISR copies the message from the receive FIFO on the card to the system's message queue. On a receive call by an application process, the corresponding message is dequeued from the system's message queue and returned to the requesting process. Figure 2.1 shows that the one-way latency (from the time the application process invokes the send system call until the time the receiver application process returns with the data from the receive system call) for this combination is around 450 microseconds. Of this, only a small fraction is due to the hardware transmission time on the wire. Most of the cost, which is incurred by the main CPU, is due to crossing protection boundaries, apart from software costs of checksumming, buffer and queue management, and driver and ATM processing overheads.

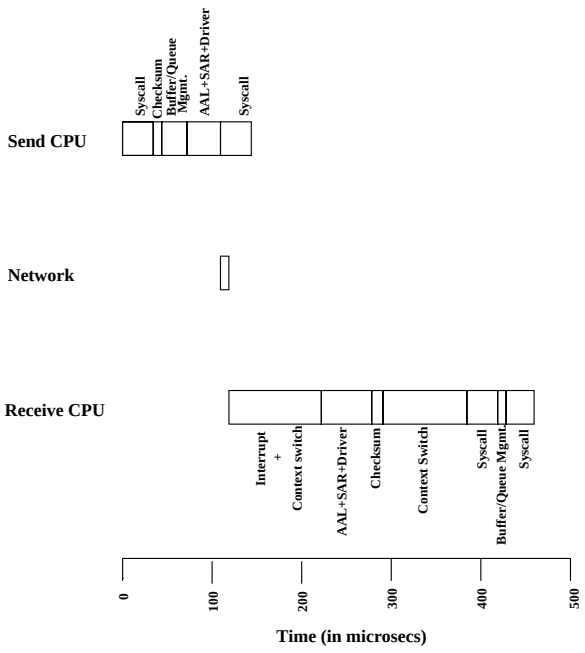


Fig. 2.1. 1-way Latency for (C1,NI1)

2.3.2 (C1,NI2)

The improvement in NI2 over NI1 is the additional DMA engine. As a result, this relieves the CPU from the burden of transferring data to/from the network interface from/to the host memory. However, the CPU still has to program the DMA to perform these operations, and we cannot allow user processes to directly program the DMA since protection will be compromised. Hence, the send/receive mechanisms have to be implemented in the kernel. Also, DMA controllers normally work with physical addresses (not virtual addresses). Once the DMA controller is programmed with the physical address of the buffer in the host memory, this buffer has to be pinned (not paged out) till the data transfer is completed.

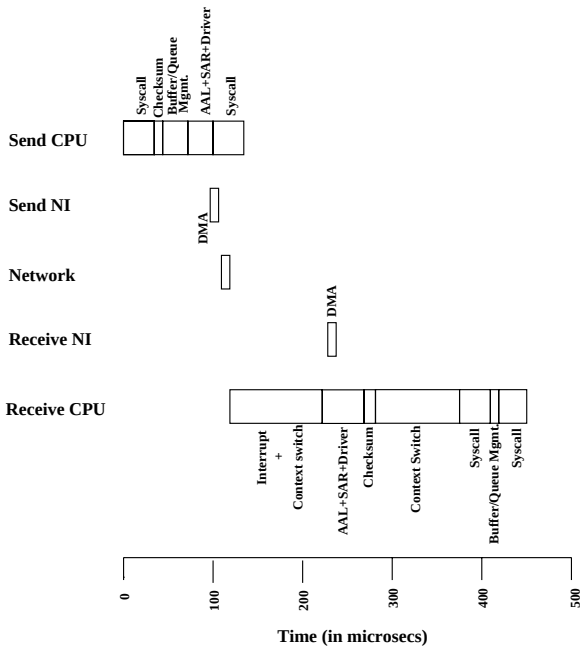


Fig. 2.2. 1-way Latency for (C1,NI2)

Figure 2.2 shows the one-way latency for a message with the (C1,NI2) combination. Comparing Figures 2.2 and 2.1, we can see that the one-way latencies for (C1,NI1) and (C1,NI2) are very similar. This is because, the main CPU still does most of the other work involved which constitutes the bulk of the one-way latency. Further, for the short messages, such as the one used for this experiment, the cost of programming the DMA is comparable to the cost of directly transferring data to/from the network FIFOs. We expect (C1,NI2) to outperform (C1,NI1) only for longer messages. To illustrate this, we have varied the size of the messages exchanged in our ping-pong experiment, and we show the performance results (Round trip time) in Figure 2.3. As we increase the message size, we find the latencies growing linearly with the message size, and the gap between (C1,NI2) and (C1,NI1) increases. However, in these experiments, each CPU is sending the data and waiting for the other CPU to respond. Consequently, the true rewards with a DMA are not really reaped with (C1,NI2) since the CPU is not performing any other work while the DMA is moving data between the host memory and network FIFOs. This time could potentially be overlapped with useful work in an application and we will see this effect with the application workloads in Section 2.4.

2.3.3 (C2,NI3)

The NI3 card has been modeled after the Fore SBA 200 [44] card, which has an i960 processor (NP), on-card memory, DMA engine and hardware checksumming support. The i960 can access the host memory only through the DMA and the host CPU can access the card memory directly. C2 implements a software messaging layer similar to the Unet mechanism [93]. The queue structures for outgoing messages are

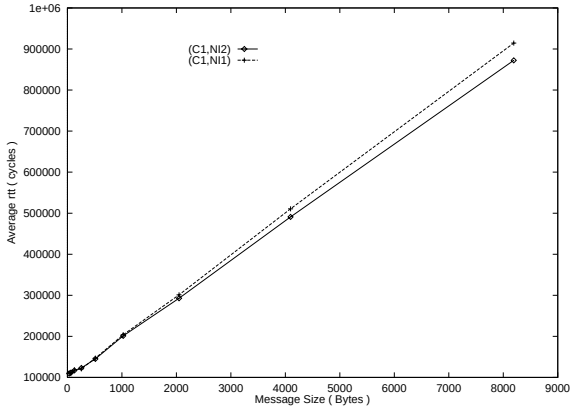


Fig. 2.3. Comparison between (C1,NI1) and (C1,NI2)

located in the card memory, and the queues for incoming messages are located in the host memory. Each user process has an endpoint which serves as the application's handle into the network. The endpoint contains a communication segment storing the messages and the incoming (receive) queue structure, which is pinned in the host memory. The NP polls the network, examines incoming messages to identify the destination, and deposits them into the corresponding endpoint while updating the receive queue structure. The NP also polls all the endpoints in the host to check if there are any messages to be sent, and pushes them out into the network from the send queue. Since the send and receive calls work only with the data structures in the user's own address space and need not access the network hardware directly, these calls can be implemented as a user level library.

As seen in Figure 2.4, the 1-way latency of 43 microseconds for (C2,NI3) is much lower than the 450 microseconds obtained for the earlier two cases (Figures 2.1 and 2.2). The cost of making kernel calls as well as switching contexts to process interrupts (which

are the significant components of the message latencies in the earlier combinations) are altogether avoided. Further, the NP takes over some of the queue/buffer management duties in sending/receiving messages. Finally, though not significant, the card provides a hardware checksumming capability to lower the software costs.

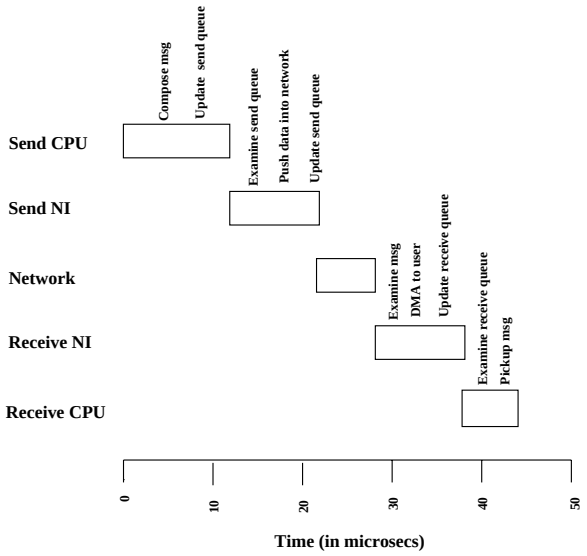


Fig. 2.4. 1-way Latency for (C2,NI3)

2.3.4 (C3,NI3)

A drawback with the previous (C2,NI3) scheme is that the communication segments of application processes have to be pinned in physical memory. This can lead to an inefficient usage of the physical memory in the system. Also, to send/receive, data has to be copied from/to the data structures in the application to/from its communication segment. These drawbacks can be addressed in C3 by making the network interface

work with virtual addresses instead of physical addresses. There are different ways to realize this, and we discuss here one such implementation for C3.

In addition to the queues that are maintained with C2, C3 maintains a software TLB on the card memory. This TLB caches the virtual to physical address translations of the applications' communication segments from the entries in the page table of the host operating system. Initially, this TLB is empty, and as messages arrive, the NP finds that it does not have an entry in the TLB (lookup is implemented using a hashing function) and interrupts the main CPU. The host CPU looks up the corresponding page table entry and records this in the card TLB. The NP then arranges for the DMA to transfer the data using this physical address. However, in the more common case (with good locality behavior), we can expect the NP to find the mapping in the software TLB (that is obtained via a hashing function) and it can directly program the DMA with the physical address without interrupting the host CPU. The host operating system has to keep track of the page table entries cached on the card whenever it is interrupted by the NP for an entry. When a page in physical memory has to be replaced, the host operating system has to check if this entry is cached on the card, and if so, it has to invalidate this entry after making sure there is no pending DMA transfer to/from this page.

The reader should note that C3 provides the capability of transferring data from/to the actual data structures of the application provided the NI is supplied with the virtual address of the data structures. It is up to the higher layers to exploit this capability. For instance, in a shared virtual memory system, the messages carry the virtual addresses which can be looked up by the NI. We do not exploit this feature at the receiver end in our Active Messages implementation since the handlers associated

with the messages are executed on the host CPU and not on the network CPU. At the sender end, the NP picks up the data to send from the user data structures directly without requiring the host CPU to copy it to the communication segment.

There are 3 different cases under which the performance of (C3,NI3) should be considered. The first scenario is where the NI finds the virtual address to physical address mapping in the card TLB. In this case, we find the performance to be slightly higher than the (C2,NI3) combination with the additional cost of TLB lookups (see Figure 2.5). The second scenario is where the NI does not find the mapping in its TLB and has to interrupt the main CPU instead. In this case, the cost of the interrupt will dominate the software costs which will result in performance as in the C1 substrates. The final scenario is where the corresponding page is not present in the host memory and has to brought in from the disk by the host operating system. In this case, we find that the cost of the disk I/O overshadows the rest of the overheads.

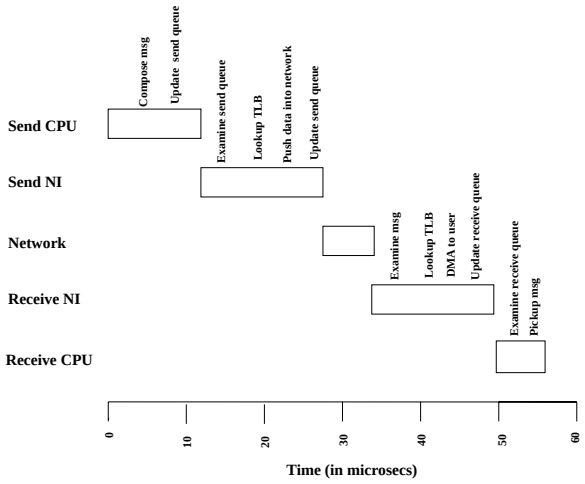


Fig. 2.5. 1-way Latency for (C3,NI3) on a TLB hit

2.3.5 Validation

Before we proceed with the rest of the discussion, we would like to point out that the communication performance presented here agree with results on actual experimental platforms. As we mentioned in Section 2.1, for the (C1,NI1) and (C1,NI2) combinations, we have not written the software and we have instead used our earlier measurements [80] to model the costs. But we have written the software layers for (C2,NI3) and (C3,NI3) ourselves (which are cycle counted by pSNOW), and the resulting performance is within 10-15% of the results presented in related studies [93].

2.4 Microbenchmark and Application Performance

The communication substrates (C1,C2,C3) discussed until now provide a basic send/receive interface. Usually a higher level programming interface is implemented for applications on top of these substrates. For instance, we can implement shared virtual memory on top to provide a shared memory programming environment, or we can provide messaging environments such as PVM [87], MPI [65], or Active Messages [94]. For this study, we confine ourselves to message passing environments, particularly the Active Messages (AM) programming paradigm which has been proven as an efficient way for integrating computation and communication in applications [94].

The AM environment provides a single resource view of the cluster and provides globally unique processes. Our AM layer conforms to the Generic Active Messages Specifications (GAM Version 1.1) providing the basic operations: am_request, am_reply, am_store, am_get, and am_poll. am_request is used to send short messages (that can fit in the message descriptor) to a remote node, and upon receipt a handler is invoked at

the remote node. am_reply is used within the request handler to send a reply message which also fits within the message descriptor. am_store and am_get are used to send and receive data that may not fit within the message descriptor. am_poll polls the network for all arriving messages. This function may either be called by the application to check for message arrival or within any of the other AM functions. In general, we expect applications to spend more time on am_poll compared to the other functions.

For the C1 substrate, these operations invoke the kernel level send/receive mechanisms while they are user-level libraries for C2 and C3. Note that the handlers are always executed at the user-level in the host CPU, regardless of whether we have a NP or not. Consequently, for the (C3,NI3) implementation, even though a NP can potentially be given a virtual address (given in the incoming message) to deposit the message directly in the user process data structures, we cannot exploit this feature with the current AM implementation for this substrate. The NP has to DMA the message into a designated user communication segment and invoke the handler which may in turn have to copy into user data structures. However, at the sender end, the NP can instruct the DMA to pick up the data directly from the user data structures instead of requiring the host CPU to copy it to a communication segment.

In the following discussion, we present performance results for 2 microbenchmarks (Ping_Pong and Ping_Bulk) and 2 applications (FFT and Matmul) written using the AM interface for the different communication substrates.

2.4.1 Ping_Pong and Ping_Bulk

These are microbenchmarks for comparing communication software performance between two nodes and are part of the Active Messages distribution codes. One node sends a message to another and waits for a reply. The handler at the destination sends back an am_reply. In Ping_Pong, the message is short and can fit in the descriptor itself, and uses the am_request call. In Ping_Bulk, the message is longer (1024 bytes) and the am_store call is used. These actions are performed over a number of iterations (128 in this study) and the roundtrip time (RTT) per iteration is presented in Figures 2.6 (a) and (b). Note that the performance results include the costs of the communication substrates, the costs of the AM implementation, and the frequency/dynamics of the invocation of the poll routine. As expected, the C1 mechanism (on NI1 and NI2) performs poorly for both microbenchmarks because of crossing protection boundaries. In the Ping_Pong case where the messages are small, the cost of programming the DMA offsets any savings in the data transfer time, resulting in poorer performance of NI2 over NI1. However, for the 1024 byte messages in Ping_Bulk, NI2 shows better performance. Since the same data items are being exercised across iterations for both benchmarks, the cost of interrupting the host CPU to fill the software TLB on the card gets amortized for (C3,NI3), giving performance that is close to (C2,NI3).

2.4.2 FFT

FFT is a 1 dimensional *Fast Fourier Transform* application which we have translated to an AM implementation from our earlier scalability studies [81, 82]. The application goes through three phases. In the first and third phases, the processes perform the

(a) Ping_Pong (b) Ping_Bulk

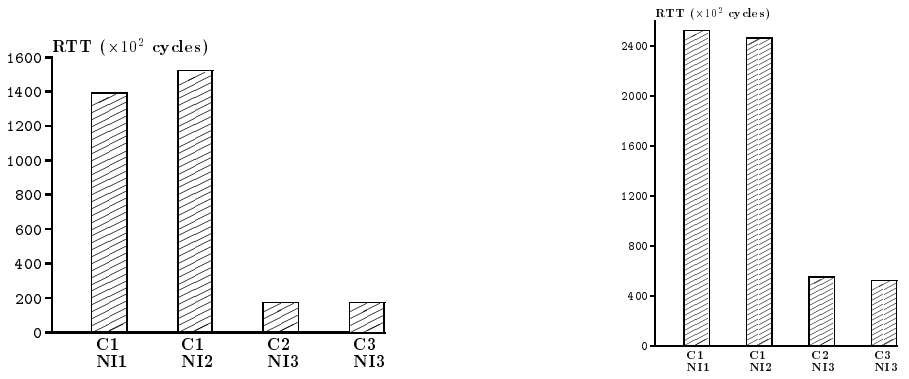


Fig. 2.6. Performance of Microbenchmarks

radix-2 Butterfly computation, which is an entirely local operation. The middle phase is the only phase involving communication in which the cyclic layout of data is changed to a blocked layout using an all-to-all communication step. This communication is implemented using am_get and am_poll functions. Each node runs only one application process, and thus when we vary the application processes, we are also increasing the number of workstations accordingly.

Figure 2.7 (a) shows the performance for a 64K problem with 4, 8, and 16 processes. The times are presented for the local computation performed by a host CPU as well as the time spent by the CPU in each of the AM functions. We find the overall execution time going down as we increase the number of processes. But, while the computation time (useful work) done by a process goes down, the time in the AM calls increases suggesting a point of diminishing returns. Of all the AM calls, we find the maximum time spent in am_poll, with the other calls being overshadowed (and not showing up in the graphs). In Figure 2.7 (a), we do not find significant differences between

(a) Varying processes for FFT(64K problem) (b) Varying problem size for FFT(16 procs)

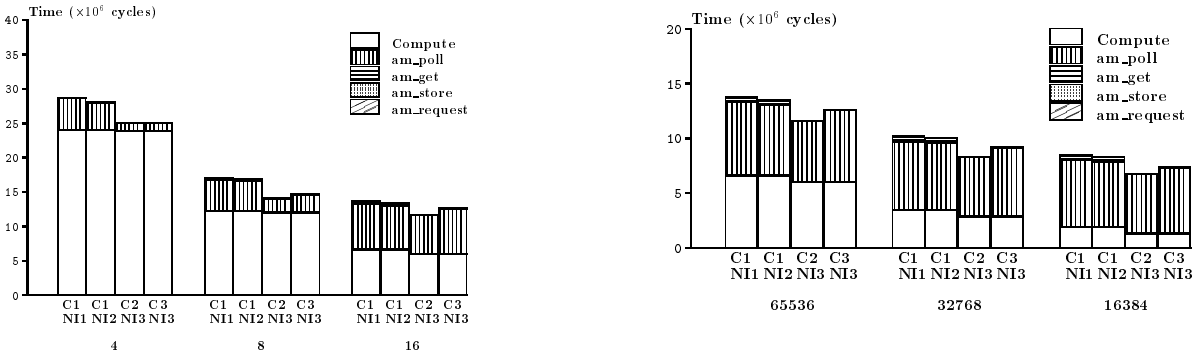


Fig. 2.7. Performance of FFT

(C1,NI1) and (C1,NI2). However, the differences between the two (ie. the advantage of the DMA) becomes more apparent when the problem size is increased (see Figure 2.7 (b)). In general, (C2,NI3) and (C3,NI3) do better than the former two. Between these, (C2,NI3) does marginally better than (C3,NI3), with the difference becoming more apparent for larger problem sizes (where the NP is likely to perform more TLB lookups and interrupt the host more often) and larger number of nodes.

In the following two application execution time graphs, we are not showing the performance of (C1,NI1) and (C1,NI2) since these perform much worse than the other two (to help us focus in on the benefits of (C2,NI3) and (C3,NI3)).

2.4.3 Matmul

This is a *Matrix Multiplication* application that uses a systolic algorithm given in [45] to multiply two dense matrices of size $N \times N$ (480×480 in this exercise). The input matrices are decomposed in two dimensions, and the submatrices are allotted to each process. Each step of the algorithm involves three stages: a submatrix is broadcast to

other processes of the same row; local computation is performed; and the submatrices are rotated upwards within each column. These communications involve am_store and am_poll functions.

Here, we have used a different scaling strategy from FFT in varying the number of processes to show how multiple processes executing on a single node can affect performance. The performance results shown in Figure 2.8 are for 3 workstations each running 3 processes (3×3), and 4 workstations each running 4 processes (4×4) for multiplying two matrices of size 480 * 480. The time reflects the virtual time that the host CPU spends on executing each application process. Thus, as we move from 3×3 to 4×4, while the local compute time for a process would go down, the overheads would increase because of higher communication as well as the higher contention for the shared resources on a workstation.

(C1,NI1) and (C1,NI2), as expected, perform much worse than the other two with (C1,NI2) doing slightly better because of the longer messages and the multiple application processes running on the host CPU that benefit from the DMA transfer. Of (C2,NI3) and (C3,NI3), the latter does better than the former (see Figure 2.8). This application uses large data transfers and there is a saving in the memory copying costs for (C3,NI3) since the host CPU has to copy the data to be sent to pinned DMA buffers for (C2,NI3). Further, the high cost of TLB fills by the NP is amortized over numerous message exchanges.

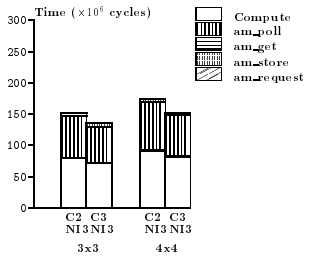


Fig. 2.8. Performance of Matmul

2.4.4 IS

IS is an *Integer Sort* application drawn from the NAS suite [7] that uses bucket sort to rank a list of integers. We have coded an AM version of this application drawn from the implementation suggested in [74]. The input data (the list to be ranked) is equally partitioned among the processes. Each process maintains local buckets for the chunk of the input list that is allocated to it. Hence, updates to the local buckets are entirely local operations. Next, these local buckets are merged into a global version of the buckets using a ring type of communication between the processes. For each bucket, each process is then assigned a set of ranks by atomically subtracting that processes' local bucket information from the global bucket information. This involves an all-to-all communication between the processes, but the communication is skewed to minimize hotspots. Finally, with all this local information, each process ranks the elements of the list assigned to it.

Figure 2.9 shows the performance results for IS with 4 and 8 workstations each running one application process. Compared to Matmul, the performance results for IS are reversed. For the chosen problem size in IS, the cost of TLB fills does not get amortized

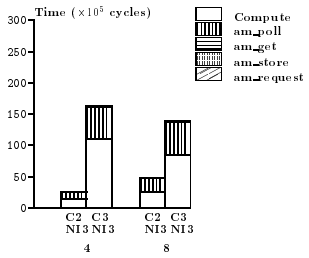


Fig. 2.9. Performance of IS

enough to be offset by savings in memory copies of the host CPU. Consequently, (C2,NI3) does better than (C3,NI3) for this application. We would like to point out to the reader that the "compute" time lumps together all the time spent in the application code (not in any of the AM calls). This time can again depend indirectly on communication costs because of any synchronization and other dynamics in the program execution. Since the communication costs of (C2,NI3) are lower, it also has an indirect consequence of a lower compute time.

2.5 Impact of Network Processor Speed

A simulation tool such as pSNOW not only allows us to study and understand the performance and scalability of specific application-architecture pairs (as shown in the previous section), but also provides us the flexibility of varying architectural parameters that would not be possible on an actual hardware platform. To illustrate the benefit of such a tool, we study the impact of the network processor (NP) speed on the communication performance and use the results to project the requirements from the network processor. Given infinite resources (in terms of technology and dollars), we

would like to have the fastest host CPU and the fastest NP for our system. However, with physical/economic constraints, it would be useful to find out the rate at which the NP should be speeded up as the main CPU on the workstation keeps getting faster. This information can help pick a network interface card for a particular workstation and, more importantly, guide future network interface designs.

Note that these studies are relevant for only the (C2,NI3) and (C3,NI3) combinations. For both these combinations, the code executing on the NP is responsible for moving data from/to the user address space to/from the network. In addition, for (C3,NI3), the NP needs to look up the software TLB to find the virtual to physical address mapping for the user buffer, and may need to interrupt the host operating system if it does not find the information in the software TLB. An improvement in the NP clock, would speed up this code helping lower the communication latency.

The (C3,NI3) combination exercises the NP more than the (C2,NI3) combination. Our first set of experiments brings out this effect and studies the impact of the speed of the NP on the difference between these two combinations. Figure 2.10 shows the performance results from these experiments for the Ping_Pong benchmark. We have shown the performance of the two combinations for 2 instances of the Ping_Pong benchmark (one for 16 iterations and the other for 256 iterations), over a range of NP clock speeds with the host CPU set to 200 MHz. As expected, (C3,NI3) has a lower roundtrip latency than (C2,NI3) because it requires the NP to look up the software TLB before initiating the data transfer. However, for the very first message, the look up in the TLB would fail, resulting in interrupting the main CPU. This cost is significant particularly if the number of iterations is low (see graph for 16 iterations). However, once the TLB

is filled, subsequent messages will not incur this penalty, and the cost of the interrupt can be amortized over a large number of iterations (see graph for 256 iterations). For larger number of iterations, after the NP gets at least one-third as fast as the host CPU, the performance of (C3,NI3) and (C3,NI2) are comparable. But for lower number of iterations, however fast the NP executes, the cost of the interrupt (which is executed on the host CPU and will not decrease with a fast NP) there is always a gap between (C3,NI3) and (C3,NI2).

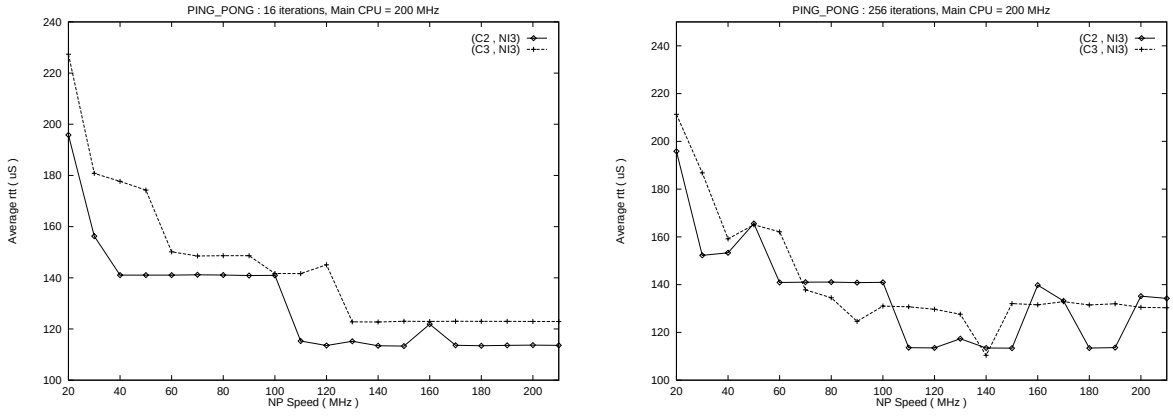


Fig. 2.10. Comparison of (C2,NI3) and (C3,NI3)

These results suggest that the software TLB lookup is by itself not a significant problem with a fast NP. On the other hand, it is more important to lower the cost of the TLB misses which results in a host interrupt. One possible way of lowering this cost is by exposing the host operating system page table data structures to the NP. The NP can directly look up the page table instead of interrupting the host. On a multiple CPU

workstation where we can use one of the CPUs as a NP, this will be fairly straightforward to implement since the same OS runs on both CPUs and the page tables reside in shared memory.

In the next set of experiments, we vary the host CPU clock as well as the NP clock, and study the performance of the (C2,NI3) and (C3,NI3) combinations for the Ping_Bulk benchmark. The number of iterations used for the benchmark is 16 (where the interrupt cost for (C3,NI3) is not amortized) and the message size is fixed at 32 bytes. Figure 2.11 shows the performance results for these experiments, where we plot the average round-trip time for the benchmark over a range of NP clock speeds for each chosen host CPU clock. From these results, we can make the following observations:

- The average roundtrip latency uniformly improves as the NP becomes faster. However, for a chosen host CPU speed, there is a point of diminishing returns (the *knee*) beyond which it does not help to make the NP any faster.

- The knee depends on the host CPU speed. For the (C2,NI3) combination, the knee occurs at a NP speed around one-third that of the main CPU, and for the (C3,NI3) combination, it occurs at a NP speed around half that of the main CPU.

Though the results are not fully conclusive, these observations suggest that we should have NPs that are at least half as fast as the host CPU. The communication to computation ratio in Ping_Pong/Ping_Bulk is much higher than what we can find in typical applications with good scalability behaviors. We qualify applications as scalable (in an intuitive sense) if there is a good mix of communication and local computation and the generated communication traffic is fairly uniform (there are no hotspots). We

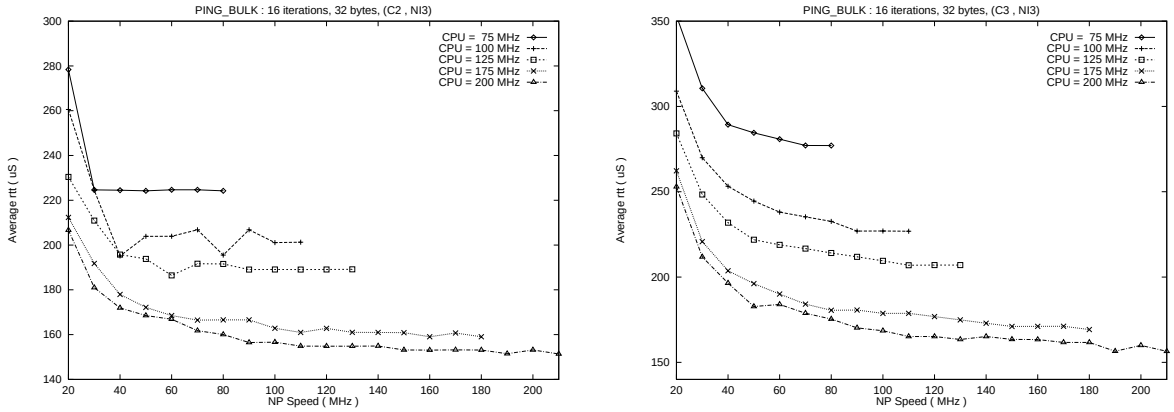


Fig. 2.11. Impact of NP Speed

have conducted a similar set of experiments for a scalable execution of FFT, and the results (not shown here) do tend to confirm that a NP should be at least half as fast as the host CPU. However, a more comprehensive study with a wider scope of applications and a detailed sensitivity analysis to study the relationship with other parameters (such as bus bandwidths, network bandwidth etc.) is needed before we can make a strong pronouncement. Still, the NP speed results from this study can represent a good operating point for many scalable applications.

The result presented here has other important consequences. Given a host CPU and a NP which exceeds the requirements specified by our result, it would make sense for the host CPU to hand over extra work to the NP. Thus, in a Active Messages environment where a handler is invoked upon message arrival, it would probably be more efficient to execute the handler on the NP (instead of the host CPU) in such scenarios. However, if the NP was slower, then it would be better to go for a (C2,NI3) communication substrate instead of (C3,NI3). The requirement for the NP speed can also be used to calculate

the number of host CPUs that a single network interface card can support efficiently on a SMP (symmetric multiprocessor) workstation. Also, there has been recent interest [40] in dedicating one of the CPUs on a SMP workstation for communication purposes (performing the tasks of a NP). Our results confirm the observations in [40] that the CPU dedicated for communication would be under-utilized if there was only one other CPU on the SMP.

2.6 Summary

Performance evaluation is an integral part of any system design process. Evaluation tools should clearly identify, isolate and quantify the bottlenecks in the execution to help restructure the application for better performance, as well as suggest enhancements to the existing design. While there has been significant recent progress in novel network interface designs and system software solutions to lower the communication overheads on emerging clusters, performance evaluation tools for these environments have not kept pace with this progress. Previous performance evaluation studies have focussed on specific hardware/software designs, or have compared designs across different (in terms of the vendor, capabilities and technology) hardware platforms making it difficult to generalize the results. Recognizing the need for a tool that can help us uniformly compare existing solutions and evaluate newer designs, we have presented an execution-driven simulator called pSNOW in this part of the work. This tool gives us a unified framework to model different designs, and evaluate these designs using real applications.

While pSNOW can be used to study a wide variety of cluster architecture issues, we have used it to evaluate communication support for clusters. As a result, pSNOW

provides an object-oriented view to model a range of network interface designs, and different software solutions.Using pSNOW we have modeled three network interface designs, and three different communication software substrates. On top of these substrates, we have implemented the Active Messages interface for running parallel applications. Using two microbenchmarks and two applications, we have argued the scalability of the implementations, shown detailed profiles to isolate hardware and software bottlenecks, and illustrated the relative merits/demerits of the different messaging layers,

A simulation platform like pSNOW also helps us vary hardware parameters and study their impact on performance. We have illustrated this by studying the impact of the network processor (NP) speed on communication performance. Our results show that a NP, that is at least half as fast as the host CPU, can meet the requirements for implementing efficient user-level messaging (without going through the operating system). These results hold as long as the applications exhibit good scalability behaviors (see Section 2.5), the NP is not asked to perform any additional work (such as running message handlers) other than user-level data transfer, and we are confined to single CPU workstations (not SMPs).

Chapter 3

MU-Net

As was shown in the previous chapter, software costs incurred by involving the kernel in the critical send/receive path is a significant detriment to performance. At the same time, there is a great need to have multiple applications executing at each node of the cluster, and sharing the cluster concurrently. Addressing these concerns, this part of the thesis presents the design, implementation and performance of MU-Net, a user-level messaging platform for Myrinet that allows protected multi-user access to the network. Our design of MU-Net has drawn from ideas of other user-level platforms [93, 72] but there are some key differences. Unlike the original Fast Messages implementation [72], MU-Net allows multiple applications on a node to concurrently use the network in a protected manner. Further, MU-Net provides additional mechanisms for transferring longer messages in cases where the host CPU has other useful work to do, while Fast Messages always employs the host CPU to packetize/reassemble longer messages. Our performance results on a SUN Ultra Enterprise 1 platform running Solaris 2.5 show that despite being able to support multiple application processes, MU-Net performs comparably to Fast Messages 2.0 (the version which supports only one application process per node).

3.1 Myrinet

Before we discuss the MU-Net implementation, it is important to understand the Myrinet hardware on which it is implemented.

Myrinet [16] is a high-speed switch-based network which allows variable length packets. A typical Myrinet network consists of point-to-point links that connect hosts and switches. A network link can deliver 1.28 Gbits/sec full duplex bandwidth. Source routing is used for packets on the Myrinet network. The sender appends a series of routing bytes onto the head of each packet. When a packet arrives at a switch, the leading byte is stripped and used to determine the outgoing port. Myrinet does not impose any restrictions on packet sizes and leaves the choice to the software. Myrinet does not guarantee reliable delivery, however cyclic-redundancy-checking (CRC) hardware is provided in both the network interfaces and switches. This allows for error detection, but reliable delivery is left for higher level software layers to implement. However, the network itself guarantees in-order delivery of messages.

The Myrinet network interface card used to implement MU-Net sits on the workstation's I/O bus (SBUS on the SPARCstations) and consists of a 256KB SRAM and a 37.5 MHz 32-bit custom-built processor called the LANai 4. In addition, there are 3 DMA engines on the card. Two of these engines are responsible for sending(receiving) packets to(from) the external network from(to) SRAM. The third is responsible for moving data between SRAM and host (workstation) memory. All three DMA engines can operate independently. Note that the LANai processor cannot communicate directly with host memory, and therefore must rely on DMA operations for reading and writing

host memory. The LANai processor executes a MU-Net Control Program (MCP) that manages and coordinates activities on the interface card and interfaces with programs running on the host CPU. Though it shares the same initials as the Myrinet Control Program supplied by Myricom, it differs significantly in functionality.

3.2 MU-Net

In developing MU-Net, our design draws ideas from the implementation of U-Net [93] on ATM, and Fast Messages [72] on Myrinet. Hence, the name *MU-Net* standing for Myrinet U-Net. The design goals for MU-net are summarized below:

- provide a low-latency, high bandwidth user-level messaging layer (to avoid costs of crossing protection boundaries),
- support multiple processes running on a workstation to concurrently use the network without compromising on protection and without significantly degrading communication performance,
- implement optimizations for shorter messages, while lowering the cost of packetization/reassembly for larger messages,
- allow alternate send mechanisms that can be used to overlap useful CPU computation with communication wherever needed,

The inclusion of these features in the MU-Net design requires detailed development of user level libraries (API), kernel level drivers, and software for the LANai. We

have implemented these components for Solaris 2.5 running on a range of SPARCstations. The following subsections describe the design and responsibilities of each of these components.

3.2.1 User-level Library

As in [93], MU-Net supports the notion of an *endpoint* that is intended to give user processes a handle into the network. The endpoint is visible to the user as a structure which contains state information about pending messages (to be sent or received) on that endpoint. A user process can only access those endpoints that it creates. In essence, an endpoint virtualizes the network for each user process, and uses the traditional virtual memory system to enforce protection between these processes.

The MU-Net API provides user applications with an interface for creating an endpoint in the user's address space, destroying an endpoint, sending to and receiving messages from a destination endpoint. Creation and destruction of endpoints involve the kernel driver and are expensive. These are done only in the initialization and termination phases and hence do not impact the latency of the critical path.

There are two kinds of send operations both of which are nonblocking. The first is meant for processes which want to minimize latency at the cost of host CPU cycles. In this kind of send, the host CPU is used to transfer data to the network interface card instead of the DMA engine, similar to the send mechanism in Fast Messages [72]. However, for long messages, the time spent in the send call (proportional to the size of the message) may become significant especially if the CPU has other work to do. Hence, in our API, we provide a second mechanism called `send_DMA()`, which uses the DMA for

large data transfers to the card. If the length of the data to be transferred is smaller than 128 bytes, send_DMA defaults to the normal send. We expect applications to use this send when they have work that can be overlapped with the operation, and when they can tolerate a slightly higher latency. If the applications cannot proceed with useful computation, and the message to be sent is larger than 128 bytes, then they can use the normal send() mechanism which uses the host CPU to packetize the data and transfer it to the network interface.

There is only one receive call which is used to receive both kinds of messages.

3.2.2 MU-Net Kernel Drivers

MU-Net uses a two driver design in the operating system kernel similar to [16]. The MU-Net *myri* driver is used to attach to the Myrinet device, and map the LANai registers and SRAM into kernel memory. The second MU-Net driver, called the *mlanai*, is a *pseudo* driver, and therefore does not actually drive any physical device. Rather, the *mlanai* driver provides ioctls for creating and destroying endpoints. It also maps a communication segment into user space. The details of the communication segment are described later.

3.2.3 MU-Net MCP

The MU-Net Control Program (MCP) is the program that runs on the LANai processor which is downloaded into the card SRAM by the host during initialization.

The MCP does not directly interact with the host. It detects send requests by polling the SRAM. It multiplexes these send requests and sends them out into the network. Incoming messages are first buffered on the SRAM, demultiplexed and then

delivered to the destination endpoint. An examination of an incoming message is required to determine the destination endpoint and a DMA operation is used to perform the delivery.

The maximum number of endpoints supported by the card is statically determined. However, the number of active endpoints is dynamic and communicated to the MCP by the host driver. This ensures that the MCP does not spend time polling for send requests from inactive endpoints.

3.2.4 Details of MU-Net Operations

The MU-Net operations are described in the same logical order they would most likely occur in a typical user application. The implementation of these operations is also discussed along with other design alternatives.

3.2.4.1 Creating an Endpoint

Messages are exchanged between endpoints and not processes. One process could create multiple endpoints though it is not necessary since endpoint communication is connectionless and the same endpoint can be used for sending/receiving from multiple remote nodes. An endpoint consists of three regions, two of which reside on the host memory and the third on the card SRAM. All three areas are mapped into the user processes' address space. The first two regions are the Host Send Buffer and the Host Receive Buffer. The Host Send Buffer is used to store long messages in host memory for a subsequent LANai-initiated DMA to SRAM (send_DMA() operation). The Host Receive Buffer is the target of LANai-initiated DMA's for incoming short and long messages. The third region is the Send Descriptor area located on the SRAM. It is written by the

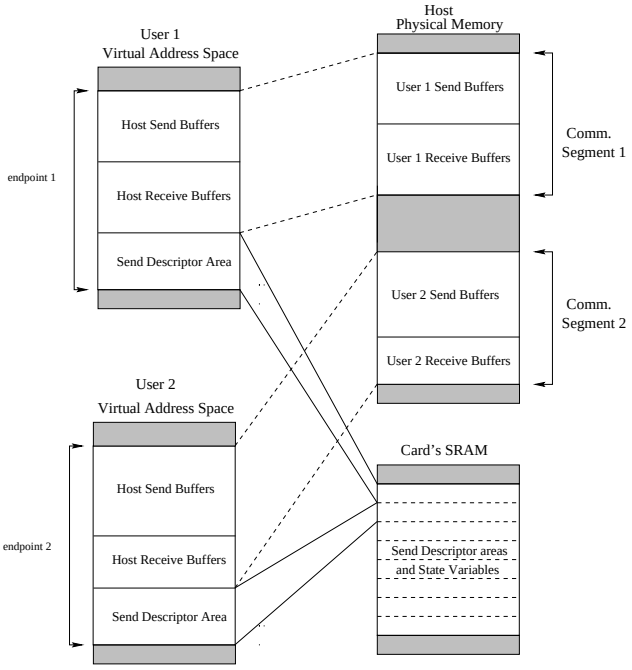


Fig. 3.1. Mapping of Endpoint into User Address Space

host CPU during a send call. Figure 3.1 shows two user processes' endpoints and their mappings. The two host buffers together constitute the *communication segment* of the endpoint.

MU-Net allows a user program to dynamically create endpoints. To create an endpoint, the user program calls the user-level MU-Net API and provides the desired sizes of the Host Send Buffer and Host Receive Buffer. The size of the Send Descriptor area cannot be specified by individual user processes. The communication segment needs to be allocated and pinned in physical memory and mapped into the DMA space of the DMA engine located on the card. Since this memory mapping can only be done in kernel

mode, the MU-Net API makes an ioctl call to the MU-Net *mlanai* driver to perform these operations.

The communication segment memory is provided by the kernel and is not shared amongst endpoints. The card SRAM, which is a shared resource, is also protected by the selective mapping of only one Send Descriptor area into user space. Moreover, since the MU-Net API library can only use virtual addresses, it can only access those portions of the SRAM mapped in by the driver. Thus, MU-Net uses the operating system's virtual memory system to implement protected user-level communication similar to [93]. The driver, being part of the kernel, is assumed to be secure while the user can potentially use a different library with the same API.

The major design issue relevant to endpoint creation is the partitioning of the communication segment. In MU-Net, the partitioning of the communication segment into Send and Receive areas is done before communication begins and cannot be changed in the middle. If the number/size of one type of message (send or receive) is much lesser than the other, the corresponding area is underutilized and may even affect the latency/bandwidth of the other type (when the latter operates at peak buffer capacity). An alternative would be to divide up the entire communication segment into fixed size buffers, which could be dynamically assigned for a send or a receive (by maintaining pointers to these buffers on the host and on the card). This approach, used in [93], has the disadvantage of limiting the size of a single data transfer between the host and the card (in either direction). The fixed size of the buffers would have to be large to avoid multiple data transfers in the average case. However larger sized buffers would also lead to greater internal fragmentation. Hence the rigid partitioning approach was chosen. If

the user process has a priori knowledge of traffic patterns, it can choose to have one area larger than the other. With the communication segment being pinned in physical memory, a process needs to exercise restraint in the sizes requested since it can impact overall host memory performance. This restraint can also be enforced by the driver on creation of an endpoint.

3.2.4.2 Sending a Message

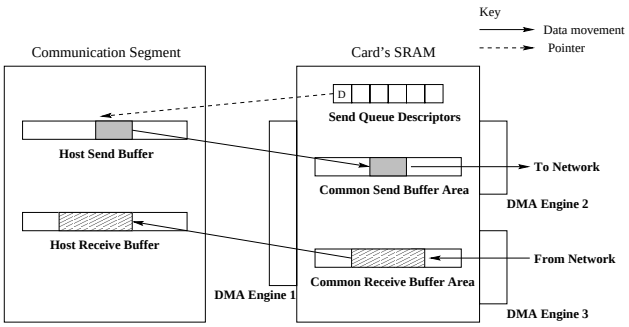


Fig. 3.2. Anatomy of a MU-Net Operation

From the user's viewpoint, once an endpoint has been created, the process of sending a message is straightforward. The user calls the MU-Net API with a pointer to message data, length of message and destination endpoint.

For a message longer than 128 bytes (henceforth called a long message), the API copies the message data into the Host Send Buffer, creates a Send Descriptor and appends it to the Send Descriptor queue of the endpoint in the SRAM. Within the Descriptor is a pointer to the data in the Host Send Buffer (which is part of the DMA space of

the card). The MCP polls the Send Descriptor queue to detect messages to be sent. Upon detection of the newly added descriptor, it uses the pointer within it to initiate a DMA of the message data into a Common Send Buffer area in SRAM (Figure 3.2). The MCP also creates a message header that includes routing information and a 2-byte tag (1 byte for the message type and the other for the destination endpoint number). After the DMA from host completes, the packet (header + data) is DMA'ed from the SRAM onto the network. Figure 3.3 shows the layout of a MU-Net packet.

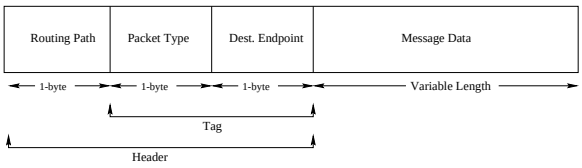


Fig. 3.3. Structure of a MU-Net Packet

The above send_DMA process works well for situations where the application can tolerate slightly higher latencies because it has other work to do. As observed in [72], it is more expensive to transfer data into the SRAM using DMA than having the host CPU directly write it. So for the normal send mechanism, we have an optimization wherein the descriptor entry itself contains the data (and not a pointer to it). A 128 byte size buffer is part of the Send Descriptor. For messages up to 128 bytes, the API copies the data into the Send Descriptor created. The Host Send Buffer is not accessed. More importantly, the MCP does not have to perform a DMA to get the data from the host and can proceed with the other steps of a message send directly. When the application

wants to send messages longer than 128 bytes, the API packetizes the message and places them as separate entries in the descriptor queue. This packetization is transparent to the MCP.

We have found that the performance of the send operation depends heavily on the implementation details. The following are some key issues :

- **Descriptor detection**: Instead of polling head and tail pointers to the descriptor queue (which we have found to be highly inefficient), the MCP can poll a designated location within the next expected descriptor entry to find out if data is ready to be sent. This entry needs to be the last field in the descriptor filled by the host to avoid a race condition.

- **Queue management**: Any queue that is accessed by both the host CPU and the MCP needs to be managed carefully for correctness and efficiency. In MU-Net , there is one producer and one consumer for items in a queue and these are on different sides of the I/O bus. For the Send Descriptor queue, the host CPU is the producer and the MCP the consumer. On the receive side, the situation is reversed (as we shall see later). The host CPU can access the SRAM directly (through memory mapped I/O) but the MCP can only access host memory through DMA operations. DMA is not suitable for frequent accesses to individual variables. So all queue maintenance information is kept on the SRAM in the Send Descriptor Area.

  All queues are circular and both producer and consumer have to wrap around when they reach the end of a queue. Since wrap around occurs relatively infrequently, the

cost for checking it need not be incurred on each queue access. Another cost saving measure is to reduce the number of updates required to queue variables by the producer and consumer. Typically the producer increments the tail after putting an item on the queue and the consumer increments the head after consuming an item. Queue full and empty checks require both variables to be read by the producer and consumer respectively.

In MU-Net, the user process and the MCP are decoupled as far as possible for purposes of queue maintenance. The user process keeps a local copy of the Send Descriptor head and tail pointers. The portion between the head and tail represent messages queued on the SRAM but not yet sent on the network. Additions to the queue (after a send), result in an update of the local tail alone. The updation of the head kept on the SRAM is done by the MCP after it consumes a new message in the queue. The local head is important only when the local tail becomes equal to it (signifying a queue full condition). At this point that the user needs to reconcile the local head pointer with the value kept on the SRAM. As long as both MCP and user maintain FIFO order of the queue, this decoupling improves performance without compromising correctness.

- **Overlap of steps**: The MCP tasks for a send operation need to be carefully examined to maximize overlap of operations without affecting correctness. After initiating a DMA from the host for long messages, the MCP has to wait for it to complete before going on with the next step of the send operation. There is potential for doing useful work here (unrelated to the send) instead of busy waiting

which is possible because of the separation of all three DMA engine operations. This issue is examined further in the next chapter.

3.2.4.3 Receiving a Message

The MCP detects incoming packets from the network by checking flag bits in a LANai Interrupt Status Register (ISR). The MCP then initiates a DMA from the network into a Common Receive Buffer Area on the SRAM. After this DMA completes, the length of the message and the destination endpoint are read off the message header. A DMA to that endpoint's Host Receive Buffer is initiated. Upon completion of the DMA, the appropriate queue variables are updated for that endpoint.

We would like to point out that there is an optimization possible here. Instead of DMAing off the net and then examining the header for the length and the endpoint, there is a way to examine just the first bytes of a message and program the host DMA with these parameters in parallel with the DMA of the packet onto the SRAM. Note that the host DMA cannot actually begin before the entire packet is in the SRAM.

The user process calls the MU-Net API to receive a message. The API checks for a pending message by polling the queue state variables. It then copies the message data to the location provided by the the user process and updates the queue state variables to reflect consumption of the message.

Implementation details for the receive operation are discussed below:

- **Short message optimization**: Unlike the send operation, no distinction is made between long and short messages and consequently no optimization is possible for short messages.

In [93], the receive operation is done differently for long and short messages. For long messages, the message data is first DMA'ed to the host. This could require multiple DMA's to be initiated since the host buffer sizes are fixed. After it completes, a *receive descriptor* is DMA'ed to a queue residing in host memory. Besides message length, the receive descriptor contains pointers to the host buffers used as destination for the first DMA. The user detects a message by polling on the receive descriptor. For shorter messages, the data is copied to the receive descriptor buffer and only one DMA is done.

We are not using this strategy because we feel that the LANai processor, which is already quite slow [72], would be taxed more than needed. Also, we are not using the pool of fixed buffers strategy. Instead, we allow the Host Receive Buffer to accommodate variable sized messages that are placed adjacent to each other in the order received. Hence all messages, short and long, need only one DMA to the host.

- **Message detection**: The only way the MCP can communicate with the host is via the DMA. But, we would like to limit the number of DMA operations to one (for transferring the data to the host memory and implicitly informing the host of the arrival of the message). To ensure correctness, notification should occur only after the whole message data is in host memory. The MCP can determine that this has occurred by checking for completion of the DMA it initiates. The user can determine it by polling for the arrival of the last byte or word of an expected message. The latter is preferable since notification is automatic on completion of

DMA of the data. In this case, the MCP does not need to wait for completion of the DMA it initiates. However, polling for the last word of an expected message has its own difficulties. The location of the last word has to be known, so the MCP has to ensure that message length is sent within the first word (effectively creating a header). Also, the memory at the location of the last word must be cleared before the DMA begins otherwise the host can't detect the end of DMA. Since the message length is not known apriori, it means that the entire receive buffer must be kept clear. While this can be done (by zeroing out the buffers of consumed messages), its an expensive operation. To avoid these problems, we use an intelligent probe, by the host, of the state variables on the SRAM.

3.2.4.4 Destroying an Endpoint

Once a user application no longer needs to access the network, a call to the MU-Net API can be made to *teardown* an endpoint. Destroying an endpoint involves deallocating the associated host memory. We also need to inform the MCP that it need not spend time polling for messages sent using this endpoint. The MU-Net *mlanai* driver must be called to complete this process since these tasks cannot be done at the user level.

3.3 Performance Results

To evaluate the performance of our MU-Net implementation, we have exercised this software over a couple of SUN Ultra 1 Enterprise servers connected by Myrinet (through an 8-port switch) and present performance results using a simple microbenchmark that ping-pongs packets between 2 UltraSPARC workstations.

	Message Size (in bytes)							
	8	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
MU-Net	40	50	57	71	104	162	272	495
Fast Messages	36	38	40	45	55	74	110	188
Myricom API	211	216	227	243	271	329	447	681

Table 3.1. Comparison of Roundtrip Latencies (in μs)

Table 3.1 shows the roundtrip latencies for messages using this microbenchmark as a function of the message size for MU-Net, Fast Messages [72] and Myricom's API [16]. For a fair comparison, we run MU-Net using only 1 endpoint though the code for multiple endpoints is in place. The Send_DMA() call is used on the send side which as we mentioned defaults to the normal send (the CPU explicitly copies the entire message to the descriptor on the interface card) when the message size is less than or equal to 128 bytes, and uses the DMA for data transfer to the card (the CPU is free to do useful work) when the message size is larger than 128 bytes. The results indicate that while MU-Net compares favorably with Fast Messages for short messages, its performance for long messages is poorer. This is because of FM's use of the host CPU (rather than the DMA engine) to perform data transfer to the NIC. MU-Net does much better than the Myricom API for all message sizes. The Myricom API latencies are considerably higher primarily because the API is a general purpose messaging layer supporting TCP/IP and automatic network remapping. The cost of the additional code is especially noticeable for small messages. For larger messages, the differences between the three layers diminish as data transfer dominates the critical path.

	8 bytes				**1024 bytes**	**4096 bytes**
	1 endpt	2 endpts	4 endpts	8 endpts	1 endpt	1 endpt
Detected in send queue	6.5	7.0	7.5	9.0	6.5	13.0
Header sent on network	7.0	7.5	8.0	9.5	12.5	45.0
Data sent on network	8.0	8.0	9.0	10.0	23.5	73.0
Data in SRAM	9.5	10.0	11.5	14.0	47.0	152.0
DMA to host	18.0	18.5	19.0	23.5	70.5	219.5
Message received by host	19.5	20.5	22.5	23.5	81.0	247.0

Table 3.2. Anatomy of a Message in μs (Effect of message size and multiple endpoints)

Table 3.2 shows the anatomy of the different operations performed for short and long messages in MU-Net with different number of endpoints (though the remaining endpoints are not exercised, the LANai still has to multiplex/demultiplex messages between these endpoints). The times for each row are cumulative for the operations that occur from the time the user makes an API send call. The first row gives the time taken by the MCP to detect a message. As expected, this time grows with the number of endpoints being polled by the MCP (even though only one of them is exercised). For messages longer than 128 bytes, there is an additional memory copy within the send call. The cost of this copy, however, shows up only beyond 1K bytes.

The second row gives the time for the MCP to send the header out over the network. The difference with the previous row is insignificant for the small message since it is already on the card SRAM. For the longer messages, the difference is a measure of the cost of a DMA to get the data from host memory.

The third row marks the time for the message to be completely sent out over the network. The difference with the previous row accounts for the cost of network DMA setup and transfer. Again, this is significant only for long messages.

The remaining rows denote operations at the receive end. The fourth row is cumulative until the time the receiver detects the incoming message and DMAs it into the SRAM completely. The reader should note that the receiver LANai is not just waiting for an incoming message, but is also polling its own endpoints for outgoing messages. As a result, an increase in the number of endpoints affects the performance of this operation slightly.

The fifth row shows the time taken for completion of DMA of message data to host memory. Since both short and long messages require a DMA at this step, both show a significant increase from the previous row.

The final row indicates the time when the host has completely received the message in the application. This includes the time to detect the message, copy it to the program specified buffer and update the state variables. The differences with the previous row are almost the same for multiple endpoints since this part of the receive operation is unaffected by the number of active endpoints. For larger messages, the difference with the previous row grows with message size due to the cost of the memory copy within the receive call.

3.4 Summary

This chapter has presented the implementation details of MU-Net, a user-level messaging platform for Myrinet that allows protected multi-user access to the network.

Our design of MU-Net has drawn from ideas of other user-level platforms [93, 72]. It uses the idea of virtualizing the network from U-Net [93] to provide protected multi-user access. The implementation of MU-Net has also benefited from the experiences of [72] in working with the Myrinet hardware. MU-Net has been implemented on the SUN Solaris 2.5 operating system, and performance results on a Ultra Enterprise 1 platform show that despite being able to support multiple application processes, MU-Net's performance is comparable to other messaging substrates which do not allow protected multi-user access.

Chapter 4

Scalability of VIA

With several research endeavours having demonstrated the benefits of user-level networking [20, 93, 72, 94, 15, 89, 63, 100, 13, 51], industry has taken note of its potential, and has attempted to standardize the interface between the application and network hardware in the form of the Virtual Interface Architecture (VIA) [25] specification. Such a standardization can accelerate the adoption of ULN in the mainstream - in clusters and as a base for developing conventional protocol stacks [23, 26, 33]. Hardware [48] and software [12, 19, 29, 6, 11] implementations of VIA have been or are currently being developed, together with studies examining the performance of this platform.

However, previous studies and the published performance results of ULN implementations and VIA [12, 19, 86], including MU-Net presented in the last chapter, have largely ignored a crucial aspect, namely their *scalability*. VIA provides the notion of a *channel* between pairs of communicating processes. While most of the focus in current VIA implementations is on maximizing performance on a single channel, there is the more important consideration of how performance on a channel degrades as the number of channels/connections goes up, and we refer to this issue as the scalability of VIA. With higher multiprogramming levels, the number of activities concurrently involved in network activity goes up, requiring more channels (one channel is needed per protection

domain at the very least). Further, software environments on clusters typically keep several open channels with the other nodes to lower demultiplexing and setup/tear-down costs. These factors can lead to a degradation of VIA performance, if scalability is not taken into consideration during its design. While the VIA specification exhorts implementors and designers to be especially concerned with scalability considerations, there has not been an indepth study examining the different issues underpinning its scalability.

Having concentrated on issues affecting the raw communication performance on a single channel of a ULN in the previous chapters, we now move on to examine issues affecting the scalability of a ULN in the context of VIA. Specifically, we examine the performance issues with the increase in the number of channels and the load experienced on them. This aspect of its performance is critical to ensure development, deployment and acceptability of communication products based on VIA for the cluster as well as the LAN/WAN environments.

Several factors affect the scalability of a VIA implementation. These include the network interface hardware features that are available (such as the number of DMA engines, capabilities of these DMA engines, processing power, memory size on the network interface etc.), hardware speeds of these devices (bus bandwidths, memory latencies, DMA transfer rates, processor clock speed etc.), and the software that manages/coordinates the hardware. Of these, the focus of this chapter is primarily on the first and third issues. The second issue is usually dictated by costs, technology and engineering constraints, though there are cost-performance trade-offs that one needs to address there as well. Specifically, this work starts with a baseline network interface,

similar to the Myrinet M2F-PCI32C [52] interface which offers DMAs and processing capabilities, and examines the different ways of improving this hardware (hardware DMA queues, doorbell support, multiple processing engines), and the different ways in which the software can use this hardware keeping scalability in mind.

4.1 The VI Architecture

We first provide a brief overview of the VI architecture as specified in the standard [25]. This introduction is focussed on explaining some of the terminology and providing the background for the various operations that are part of a VIA implementation. A detailed description of the architecture may be found in [25], as further clarified by [28].

4.1.1 Objectives and overview of VIA

The Virtual Interface Architecture seeks to establish standards for user-level networking (ULN) in a System Area Network (SAN). Several research ULNs have already shown the benefits of eliminating the operating system kernel from the critical path of communication at a cluster node. These ULNs [93, 64, 72, 89, 63, 37, 51, 20, 13] make different assumptions about the network interface whether it be the Network Interface Card (NIC) hardware, firmware running on the NIC, messaging libraries or device drivers. VIA is an attempt by Intel, Microsoft and Compaq, to unify the basic principles of these systems into a standard which can be used to design the network interface. The current version of the standard, v1.0, is rather ambiguous in specifying what parts are hard requirements and what are suggested implementations. Additional documents such as the Intel Developers Guide [28] and User Conformance Suite [27] help clarify some of the design objectives but their status as a specification versus recommendation is not

clear. This is perhaps a deliberate attempt to let good implementations evolve and to let some of the design tradeoffs become more clear before setting down the standards. As such, since our goal in this work is to examine the scalability issues of VIA, we assume only the actual specification [25] to be a hard requirement.

In VIA, each process gets a direct, protected access to the NIC through what is called the virtual interface (VI). A VI is opened by the process by going through the kernel and thereafter provides user-level communication in conjunction with the trusted firmware running on the NIC. Protection amongst processes relies on the selective memory mapping, and use of trusted components such as the kernel for setup, and NIC firmware for communication.

The VIA protocol is point-to-point and connection oriented. Three modes of operation, Unreliable Delivery, Reliable Delivery and Reliable Reception can be used with different assumptions about the reliability and corresponding change in performance. We assume the Unreliable Delivery mode throughout, as it is the only mode which is required to be supported and can provide the highest performance.

Figure 4.1 shows the important components of a VIA implementation. Each VI comprises a pair of Work Queues (WorkQs) - one for send (SendQ) and the other for receive (ReceiveQ). It also has a pair of **doorbells**, one for each WorkQ. The elements of the WorkQs are called Descriptors and they contain all the information needed to initiate a send or a receive operation. The *posting* of Descriptors to a WorkQ is followed by the *ringing* of the corresponding doorbell by the application to notify the NIC of pending work. Also associated with a VI are data buffers allocated in pinned (non-pageable) host memory, henceforth called registered memory. WorkQs are typically also allocated in

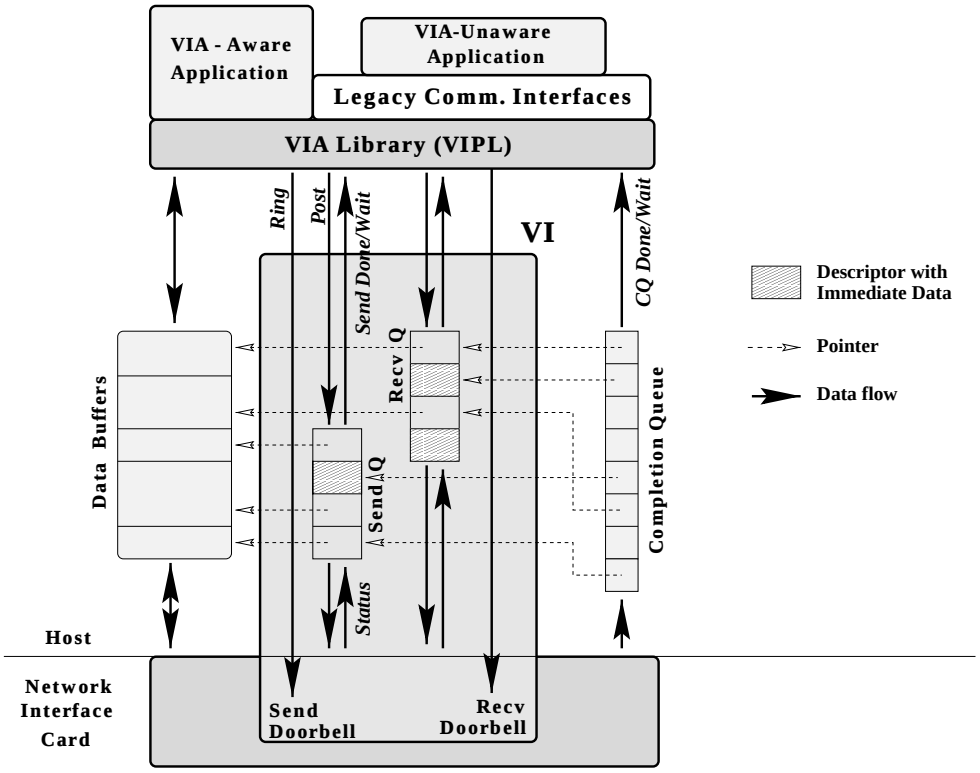


Fig. 4.1. Components of the VI Architecture

registered memory (we later discuss alternatives to this) and the NIC informed of their location during VI setup.The doorbells are allocated on the NIC and memory-mapped to the process' virtual address space allowing the process to ring a doorbell using simple writes.

4.1.2 VIA Operations

A brief description of the pertinent parts of VIA is provided below. The terminology of the VIA API, also called the VI Provider Library (VIPL) in [25], is used for consistency.

VI setup: This phase of operations is done with support from the operating system kernel for protection reasons. Typically, an application first allocates its pinned memory requirements (for WorkQs and data buffers) by calling VipRegisterMemory. A subsequent call to VipCreateVi sets up the data structures (on the NIC and the host) and initializes the VI. An initialized VI is ready for use after it is connected to another VI (typically on a remote node) using calls like VipConnectRequest and VipConnectAccept. Corresponding calls are available to disconnect a VI, deallocate its resources and close it.

Send: Logically there are three operations that an application (through VIPL calls) has to go through to send a message. First, it must allocate a data buffer in registered memory and copy the contents of the message to it. The VIPL library call assumes this is done before it is invoked. Note that registering the memory region for data buffers is typically done before the send begins as it involves a costly system call. Next, the application posts a Send Descriptor in the SendQ. The Descriptor contains the address of the data buffer, length of the message and other control and status fields. Finally, the application rings the Send Doorbell to notify the NIC of the newly posted Descriptor. Control of the Descriptor is now transferred to the NIC.

There are two ways in which the application can then wait for notification of completion of the send. One is to poll (VipSendDone) or block (VipSendWait) on an update to the status field of the Descriptor on the WorkQ. The other is to poll (VipCQDone) or block (VipCQWait) on a Completion Queue (CQ) that has been explicitly

associated with the SendQ during the latter's creation. A single Completion Queue can be associated with several WorkQs and serves to coalesce notifications from all of them.

Meanwhile, the NIC, after processing the send, updates the status fields of the Descriptor and optionally puts an entry into the associated CQ. When the application is aware of this completion notification, the send operation is over.

Receive: The actions on a receive are very similar to that of a send with the direction of data flow being the major difference. First, the application allocates an empty data buffer in registered memory. Then it creates and posts a Descriptor for it on the ReceiveQ followed by a ringing of the Receive Doorbell. When a message arrives for the VI, the NIC transfers the message contents to the corresponding buffer. The semantics of VIA require that a Descriptor pointing to an adequate sized buffer be available *before* the message arrives on the NIC. Else it is dropped.

Completion notification is done in exactly the same way as the Send. Status fields of the Receive Descriptor are updated and CQ entries inserted if necessary. The application polls or blocks on either of these as before (if a CQ is associated with a WorkQ, it is mandatory to use it for notification).

Other operations: In addition to two-sided communication calls like send and receive, VIA supports one-sided calls like Remote DMA (RDMA) Read and Write. These calls allow an application process to access the data in the registered memory regions of another application (at the other end of a VI) without requiring any explicit action by the latter. Security against arbitrary access is provided by the same mechanisms (involving memory tags) that protect one VI from another on the same node. The VIA specification

only mandates support for RDMA Writes. We do not look at RDMA operations explicitly in this work since the differences between them and the Send/Receive calls are not expected to be significant from the scalability viewpoint.

4.2 Scalability Considerations for VIA

Having looked at the overall structure of VIA, we now focus on those features that could affect its scalability. Some of these features are inherent in the protocol/specifications while others are implementation issues. There is some overlap between the following subsections as they look at different implications of the same design choice.

To make the remaining discussion more concrete we assume a Baseline NIC similar to the Myrinet M2F-PCI32B card [52]. It has a custom processor, memory that can be mapped into a process' address space, one host side DMA engine for data transfers to/from registered host memory, and two DMA engines (one outgoing, one incoming) as part of the packet interface to the bidirectional wire link to the network fabric. Figure 4.2 shows a simplified view of the NIC that is used in the discussion to follow.

For the purposes of this study, *scalability is a qualitative/relative notion capturing the number of VI channels that can be supported without significantly degrading the performance on an individual channel.* Typically, the number of channels that are active simultaneously is a subset of those that have been opened. But since the NIC/firmware has to be prepared to process messages on any of the open VIs, scalability is defined in terms of VIs opened (declared) rather than being used.

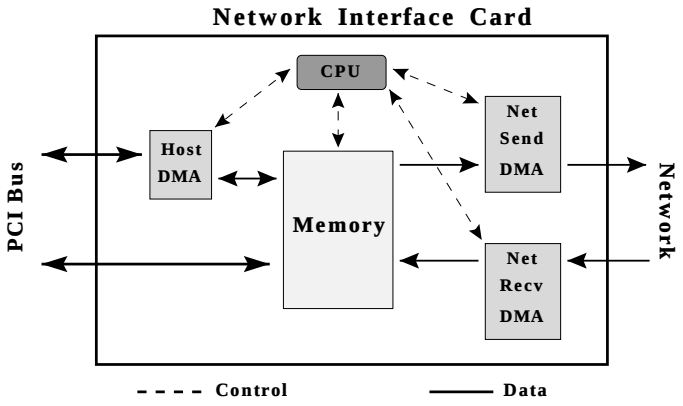


Fig. 4.2. Baseline NIC

4.2.1 Memory management

VIA, like all ULNs, relies on memory-mapping to provide protected access to the network interface. This, coupled with the need for address translation on the NIC, makes memory management an important part of the communication protocol.

Physical memory: The amount of physical memory on the NIC can determine the placement of data structures required for processing messages. If the NIC has limited physical memory, most of the data and control structures would need to be placed in host memory. Having to get them down to the NIC within the critical path would affect latency seen by a VI and the overall scalability. Also, the host memory locations used to store the data and control structures would need to be in registered memory, which is a precious resource even when host physical memory is relatively plentiful.

Buffer management: Buffer management is needed at each level of the protocol. The messaging libraries (such as VIPL) manage the registered memory buffers that are

pointed to by the Descriptors. Then there are the buffers used internally by the NIC to stage messages both from and to the network. Sometimes, having more buffers allows concurrency of operations at the expense of extra management. Not all of the buffer management needs to be in the critical path of a messaging operation, and the tradeoffs are thus not always obvious.

Address Translation: The use of virtual addresses by a Descriptor makes it necessary for the NIC to maintain address translation tables. These affect scalability in two ways. The size of these tables imposes restrictions on the number of registered memory regions Also, since the access to these tables is a necessary part of descriptor processing, their implementation on the NIC can affect message latency. However, there is scope for overlapping this access with message transfer.

4.2.2 Work and Completion Queues

Descriptors and the queues used to manage them are amongst the most critical aspects of the VI Architecture. The location of the queues, size of Descriptors and the way that event notifications are sent can all affect the latency and concurrency of messaging operations.

Location: The VIA specification appears to mandate the placement of WorkQs in host registered memory. However, there are obvious benefits to having these on the NIC itself. One DMA, in the critical path, is avoided and this is a significant saving in overhead for small messages. This has been shown for ULNs in general [93, 72, 64] and VIA in particular [19] where they are called Active Doorbells. The concept of Active Doorbells

has been used in the implementation of MU-Net as well. Less obvious is the benefit of eliminating "small" length DMAs (such as ones for Descriptors) from the scalability perspective, as we describe in our results later.

On the flip side, the status fields of a Descriptor are used for message completion notification. If the application chooses to poll on these, keeping the WorkQs on the NIC alone might lead to severe degradation of performance since the I/O bus is crossed on each poll.

If the WorkQs have to necessarily reside in host memory, we propose the use of a **shadow queue** on the NIC. After posting a descriptor in a WorkQ in host registered memory, the VIPL *additionally* writes the control segment part of the descriptor to a per-VI shadow queue maintained on the NIC. This allows the NIC to proceed with data transfer immediately after it notices the doorbell. Completion status is, however, still sent to the host resident WorkQ. Polling on it remains efficient and the semantics of completion calls such as VipSendDone, VipSendWait, VipRecvDone and VipRecvWait are maintained *including* the return of a pointer to the posted Descriptor. The NIC can reuse a shadow queue entry independent of the lifetime of the corresponding Descriptor in host memory. So the shadow queue does not necessarily consume much memory on the NIC, and does not need to hold all the fields of a descriptor (can be an abridged version). The benefits of using shadow queues is shown in Section 4.3.

Size of Descriptor payload: Independent of the above design option, the payload of a descriptor (in the immediate data segment) can be increased beyond the 4 bytes suggested by the example implementation in the VIA specification document [25]. For a

small message this would eliminate the set up costs associated with a DMA operation. This proposal to increase the Descriptor payload was also made in the Fast Descriptors idea of [19] where a payload of 16-32 bytes was selected instead of 4. However, their recommendation has not been backed by a quantitative analysis. The issue is not new - almost all ULNs have looked at such "small" message optimizations to determine the cutoff point at which the latency benefits of MMIO are not enough to outweigh the increased CPU utilization. However, these findings have always focussed only on the latency aspect. Since the effect on scalability comes from both reduction in latency and reduced contention for the DMA engine, we revisit the issue in the context of VIA.

Use of Completion Queues: The use of Completion Queues (CQ) reduces the host CPU time spent on polling for notifications. This not only improves the CPU utilization (for doing useful work) but also allows it to do the host-end processing for more messages and indirectly allows more VIs to be active concurrently. On the flip side, the firmware on the NIC has to perform an additional DMA operation for each completion notification (it has to update the relevant Work Queue and its associated CQ). Hence, it is not intuitively clear how CQ usage affects scalability overall.

One design option is to avoid placing the CQ in the host and keep it in a memory-mapped location on the NIC. This would naturally place a heavy burden on the I/O bus if the application/library chooses to poll on the CQ (instead of blocking). Since one of the reasons for using a CQ is to be able to poll for notification more cheaply, we do not consider this any further.

Another alternative is to optimize the updates to a CQ by the NIC firmware. Since the CQ is associated with multiple VIs, it could be updated after several entries accumulate. Updation of the CQ could also be deferred to a time when the NIC has fewer/no events to process. The effect of delaying notification to the application must, however, be kept in mind while attempting such optimizations.

4.2.3 Firmware design

Perhaps the most critical aspect of the performance of a VIA implementation is the design of the firmware that runs on the NIC. The elimination of the kernel from the critical path offloads much of the protocol processing load onto the NIC. This is because it is the only trusted component left in the path and also because of a desire to offload work from the host CPU which has the potential to do other useful work. However, the NIC typically cannot match the processing power or memory resources of the host. Hence, the firmware running on the NIC has to be carefully designed for optimizing performance with limited resources. Overly complex code, which is either memory or compute intensive, rarely fits the bill.

Doorbell support: The doorbell mechanism is used to alert the NIC that a descriptor has been posted and needs processing. The most efficient way of doing this is to map a portion of the NIC's memory into the host process' address space and allow it to do a memory write into a specific location which is polled by the NIC processor. Ringing the doorbell is then reduced to a memory write.

However, this approach has its disadvantages, as also outlined in [19]. The minimum memory size which can be mapped is governed by the host memory page size

(typically 4-8 KB). This severely limits the number of such mappings, given that NIC memory is limited, and maintaining protection requires that this mapped area not be used for anything unrelated to the VI or owner process. Since the protection domain is a process and not a VI, there is some scope for optimizing the space wastage by collating all doorbells for VIs owned by the same process onto a single/fewer mapped page(s).

But the greater disadvantage of not having hardware support comes from the demand placed on the NIC while it polls each of these doorbell locations. As the number of declared/open VIs increases, the number of locations that must be polled increases, regardless of their usage or collation onto a single page. And the penalty for the extra polling is paid by all the VIs (existing and newly added). This could cause a severe degradation in performance since doorbell detection is always on the critical path of a message.

The problem can be alleviated to some extent by carefully choosing the order in which the firmware polls the different doorbell locations. If load is light with only one or a few VIs sending messages, the firmware could favor these active VIs; an extreme being the case where the firmware does not move on to another VI as long as the current one is ringing the doorbell. Such an extreme case is rather tempting when the performance criteria is the latency seen in a regular ping-pong type of test where only one VI is used. For more evenly distributed loads, a round-robin approach may be more suitable. In all cases, the complexity of determining the order must justify the delays seen by the target workload.

DMA operations and event ordering: The host side DMA engine is an important element in the critical path as all NIC-initiated data transfers to and from host memory have to use this resource. For a WorkQ associated with a Completion Queue, as many as 3 host DMA operations unrelated to the message data have to be done (as shown in Figure 4.3 and explained in detail later). The host DMA for message data is of variable length and could be quite long.

When the firmware, during the course of message processing, finds the host DMA engine busy, it has to wait. How it spends this wait time is very important for scalability. The simplest design would involve a busy wait. This could be very expensive for the long messages as it directly adds to the wait for service time seen by the messages on the same or on different VIs. Slightly better is to poll for doorbells so that the average wait for detection of a rung doorbell is reduced. However, the first successful doorbell poll ends this overlap, unless a software queue of rung doorbells is maintained. One could also start processing an incoming message (from the wire) but might have to wait for the same host side DMA engine again (hardware support to alleviate this is discussed in the next subsection).

In either case, it is apparent that it might be advantageous to break up the processing of a message into well-defined stages of a pipeline. Such a breakup allows the NIC CPU to freely go from any stage of one message to any other stage of another. This requires that the state of a stage be maintained efficiently, allowing the suspension and resumption of state be determined by asynchronous events rather than a predetermined firmware order. The asynchronous events are those whose occurrence is beyond the firmware's control such as ringing of a doorbell, arrival of a message over the wire and

the completion of a DMA. One way of achieving the pipeline mentioned is to have a task queue for each DMA engine. Details of such a design and performance results for the same are presented in Section 4.3.

Pipelining as an idea is not new even in the context of network interfaces. Trapeze [100] and PM [89] both try to overlap the operation of the host and network side DMA engines. However, they do so in an attempt to maximize the bandwidth seen by one (in Trapeze) or as few as 4 (in PM) communication channels. [95] provides interesting insights into minimizing the latency of a message by resizing the fragments which need to be processed by the store-and-forward stages of a communication pipeline. While our design assumes a similar communication pipeline, it optimizes latency *across* messages, thus allowing the processing of several hundreds of VIs. This changes the kind of state maintenance required and its associated costs so that a different kind of optimization needs to be done.

4.2.4 NIC hardware

Adding hardware support for various operations that have been identified as potential scalability bottlenecks is an interesting exercise. This support can be classified into two types: those which improve the performance of the NIC in general and hence allow more VIs to be processed in the same time, and those which are aimed at the VIA protocol in particular.

Hardware doorbell support: The description of the "doorbell region" in the newer Myrinet M2L-PCI64A NIC [53] reveals one way in which hardware support for doorbells could be provided. A certain (large) portion of the I/O address space is managed by

the NIC in such a way that all writes by the host into the entire space appear in FIFO order as seen by the NIC. This serves to collate the doorbell entries into one or more queues, much like how CQs collate completions across all the VIs associated with it. The problem of large memory requirement is also side-stepped since it is not necessary for the entire doorbell I/O space to map to the same sized physical NIC memory. The hardware traps the memory references and the physical memory required is only that of the expected/maximum length of the doorbell queues.

DMA engines: There are a couple of ways by which the concurrency of DMA usage could be improved, to cut down the overhead of waiting for a DMA engine to finish (whether the firmware is waiting to use it for the next transfer or is waiting to confirm completion of the current transfer).

- Concurrent DMA transfers: A DMA engine could be programmed to initiate more than one DMA operation simultaneously. For instance, while interacting with the PCI bus, a host memory read operation necessarily causes a retry while the relevant data is fetched from host memory. This time can be used to initiate another DMA burst for a non-conflicting host address. PCI buses are designed to allow multiple DMA operations (since they could be coming from different adapter cards) and depending on the configuration, the NIC DMA engine could take advantage of this concurrency. The Myrinet card described in [53] permits upto 4 DMA channels to operate concurrently. The resulting reduced contention for the DMA engine in the message processing chain of events can improve the scalability.

- Multiple DMA entries: The DMA engine could read the parameters of the next DMA transfer request from a queue instead of requiring the NIC processor to program it each time around. A queue of DMA entries would eliminate the need for the processor to poll for DMA completion before re-programming it (it would still need to poll for completion). This feature is especially useful when a scatter-gather list is used within a Descriptor. Even otherwise, it provides hardware support for one stage of a pipelined firmware design and hence could improve scalability. [53] provides an insight into one way in which these entries could be used.

Processor-memory subsystem: As with the host node, the performance of the processor-memory subsystem on the NIC affects each message processing stage. Slow processors and memories determine the extent to which the firmware can be redesigned for scalability. If the cost of the messaging operations is too high on account of poor performance of this subsystem, the scalability could suffer.

4.2.5 Application/Library

Two important considerations in the host software (application/user-level libraries) are identified below:

Polling: For completion notifications, the application could either use the non-blocking primitives (such as VipSendDone, VipCQDone) in a polling loop, which maximizes overlap with application dependent useful work. Or it could rely on the blocking primitives. Scalability is affected by the wastage of host resources (especially the CPU) that might otherwise have been used to process another VI.

VI usage: Applications could be structured in different ways that call for the creation of a different number of VIs. It could choose to open only one VI to each distinct destination, multiplexing the messages on this VI if it needs any sub-channels. Or it could open more than one VI to a given peer process for purposes of programming ease, prioritization of messages etc. The order in which it exercises the VIs it has opened, or rather the way this order interacts with the NIC processing order, can become a problem as shown in the performance results.

4.2.6 Host hardware

Finally, the host hardware characteristics such as processor speed, memory access latencies, and cache performance can affect scalability indirectly. Investigation of the impact of these issues on VIA scalability is beyond the scope of this thesis.

4.3 Performance Evaluation

4.3.1 Simulator and Workload

To examine some of the above issues, a simulator for a cluster node (4-way symmetric multiprocessor) with a NIC (similar to a Myrinet interface) capable of supporting VIA has been developed. The network fabric is assumed to be one containing a small number of high-degree switches, and the latency and contention on the network are assumed to be negligible. The other important assumed characteristics are negligible error rates and in-order delivery of messages. All these assumptions are applicable to the Myrinet switches, and deviations from these assumptions are not expected to significantly change the results and conclusions drawn from this study. All software and hardware elements of the host and the NIC are modeled. In particular, the PCI bus

(64-bit 66 MHz has been chosen here), long identified as the primary bottleneck on the host side for ULNs, is modeled in great detail, including the PCI bridge and DMA engines. Similarly, the NIC firmware and hardware were modeled in detail, based on our extensive experience in programming and using different Myrinet cards, so as to allow a good performance evaluation of the chosen issues from the scalability perspective. The simulator development has been undertaken in an industrial setting, incorporating a lot of proprietary hardware information, and these models have been used in product design exercises as well.

Microbenchmark workloads have been chosen to highlight each issue under investigation. Though one would like to exercise the system with real applications, there are some hurdles in this approach. The first is the limited availability of cluster workloads that can be readily used. Workloads currently in use (scientific applications written for MPI, web servers etc.) on clusters do not use many connections, more so because of the fear of degrading performance (a chicken and egg problem). As a result, they are affected more by the bandwidth per connection than the scalability issue. In contrast, a scalable VIA implementation, may actually warrant the use of VIA capabilities in a different fashion than how they are being exercised in current applications/environments. Subjecting several VIs to a large load (as is done in a few VIs in these applications) results in saturating the system. Further, a single workload gives only one data point for analysis. On the other hand, a carefully chosen spectrum of microbenchmarks that stress the system with different loads, can give more revealing insights from the scalability perspective. Since our goal here is more to examine the scalability of the design alternatives, rather

than on raw performance (as measured by single VI latency/bandwidth), microbenchmarks are a reasonable choice. With different loads, microbenchmarks can give results for a wide range of operating conditions, which can then be used to make a choice for a given application/environment. The microbenchmarks are composed as follows.

Each 4-way SMP node runs 4 identical processes, one on each CPU. A process establishes a separate VI channel with one other process on every remote node. The number of VIs at each node thus increases as $4*(N-1)$ where N is the number of nodes in the cluster. Considering cluster sizes (N) of 2,8,16,32,64 and 128 nodes, we have d=4,28,60,124,252 and 508 VIs created (declared, though not necessarily used). Each process on a node "owns" $(N-1)$ VI channels, which is henceforth referred to as the *subrange* for that process. This model closely resembles the recommended usage in the VIA specification [25].

An *active* VI is one that is actually being used to transfer messages (has not been idle after declaration). The number of active VIs across all processes on a node (u) at a given time is obviously a subset of those that have been declared. This distinction is made to highlight the fact that some NIC designs have to do a certain amount of work for each declared VI regardless of its usage. Logically, an application process iteratively performs a send followed by a receive on each active VI, though the separation between the send and the receive may change depending on the workload. Based on the number of active VIs and their usage pattern, four types of workloads are defined :

- **Type I** : One VI, randomly chosen from the subrange, is active. A send on this VI is immediately followed by a receive before moving on to the next iteration.

- **Type II** : All VIs in the subrange are active. A send is performed on all these active VIs before doing receives on all of them. Hence, the message transfer on any one VI can enjoy the benefit of overlap with transfers on other VIs of its subrange.

- **Type III** : A fixed (small) number of VIs in a subrange are active. These are used in the same pattern as Type II workloads.

- **Type IV** : A Type III workload in which more than one message operation (send or receive) is performed on each VI at a time. The intention is to have several outgoing/incoming messages on a VI, to capture situations where an application may be exchanging relatively long messages that are packetized into multiple shorter messages seen by the underlying VI layer, or is sending messages in close succession on a channel.

Each workload uses either immediate messages (which fit within the payload of a descriptor) or long messages (where the data has to be retrieved separately from the descriptor). *System load* is defined as the number of VI channels which are used actively out of the ones that have been declared open by a process. The load on any single active channel itself remains constant as we increase the number of active/declared VIs in an experiment.

The metric used for evaluation is the round trip latency incurred by messages on a node averaged across all active VIs. Various design options have been studied and representative results are presented here. The scalability of a particular option is seen primarily in the *rate of increase* of latency with VIs (declared or active). However, the absolute latencies themselves are important, as will be clear from the discussion of results.

We have also collected detailed statistics on utilization of buses, processors, and DMA engines, together with delays through different stages of the message transfer process. We do not present all these results, and briefly allude to them only when necessary.

4.3.2 Messaging Sequence

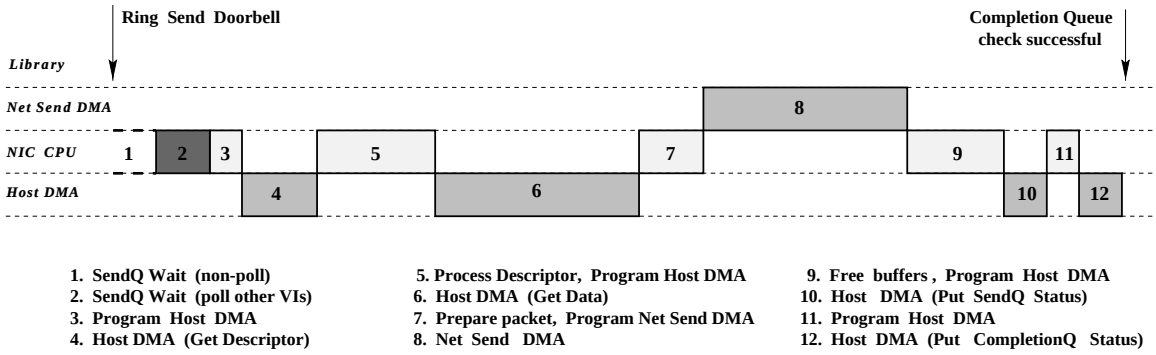


Fig. 4.3. Sequence of Actions in Sending a Message

It is important to understand the messaging sequence for the rest of this discussion. Figure 4.3 shows the different actions/events in a message send, at the user-level library, processor on the NIC, and the DMA engines (the send interface to the network and the interface to host memory). This sequence, which is similar to the implementation in [19], assumes that a long message is sent, descriptors are maintained in host memory, and completion queues are enabled. This NIC is assumed to not have any other hardware support for message processing, and is referred to as the Baseline NIC that has been shown earlier in Figure 4.2.

After a Send Descriptor is posted by the host library and the Send Doorbell has been rung, it takes a while for the NIC to detect it. Part of this SendQ Wait time is due to the processing of messages on other VIs (Stage 1 of Figure 4.3) and partly due to the unsuccessful poll of other VI doorbells by the NIC (Stage 2), since it does not have hardware doorbell support. After detecting the doorbell, the NIC programs the Host DMA engine to bring down the descriptor from host memory (Stage 3) and busy waits for its completion (Stage 4). Subsequently, it moves on to processing the descriptor (Stage 5). This involves various checks and the programming of the Host DMA engine to get the data part of the message (if it is not an immediate message). After busy-waiting for the data to be brought in (Stage 6), the NIC programs the Net Send DMA (Stage 7), waits for it to complete (Stage 8) and moves on to freeing internal buffers and preparing the SendQ Status information (dequeue/remove the element). It programs the Host DMA to put the Send Queue Status (Stage 9) and after the DMA completes (Stage 10), optionally repeats the process for a Completion Queue update (Stage 11 and 12) if a CQ is associated with the SendQ.

A similar chain of events occurs for a message receive as well, and is not explicitly shown/described here.

In the following discussion, we take such a Base NIC design (no hardware support, and sequential execution of the messaging stages) and progressively refine it with both firmware and hardware enhancements to understand their impact on scalability.

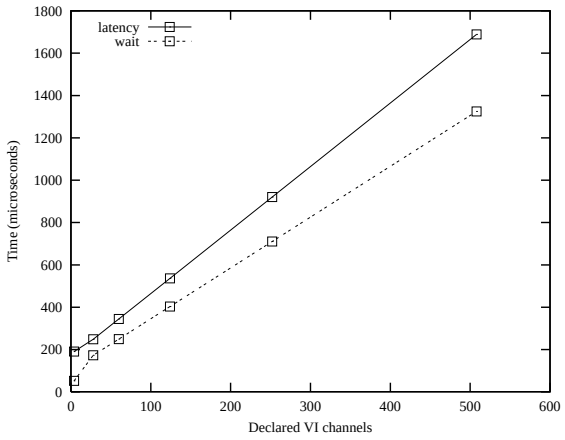


Fig. 4.4. Message latency and wait timesBase (u=8, 128 bytes)

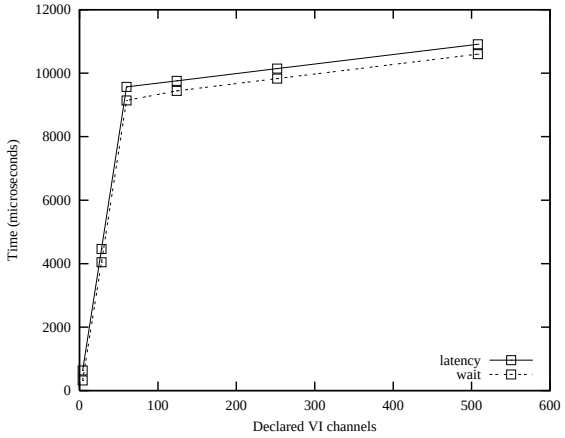


Fig. 4.5. Message latency and wait timesBase (u=minimum(60,d), 4KB)

4.3.3 Base Design: Need for a scalable solution

To motivate the need for looking at scalability, the Base NIC is used. Here, the NIC processor polls the doorbell queues for each declared VI in round robin order.

Figure 4.4 shows the effect of declared (unused) VIs on a Type III workload with 8 active VIs (u=8) at a node, and 128 byte immediate (data fits in the descriptor) messages. The latency increases almost linearly. This follows directly from the time spent by each message waiting for service as seen by the second curve in the figure (labeled *wait*) which almost parallels the latency curve. The serial processing of messages means that after a doorbell has been rung, a message may have to wait in the send queue while the NIC finishes processing another message (Stage 1 in Figure 4.3). In addition, it has to wait for a period that is roughly half the time taken by the NIC to poll the doorbell queues of all other VIs. This is seen as the length of Stage 2 in Figure 4.3.

Figure 4.5 shows the same effect for a Type III workload where a maximum of 60 VIs are active (u is set to minimum(d,60)). For $d = 4, 28, 60$, this translates to a Type II workload (all declared VIs are active), and for the rest it represents a constant application-offered load. So the same graph shows the effects of different offered loads as we increase the number of declared VIs. A sharp increase is seen in the Type II part of the curve (the application load is increasing), while the Type III part shows a reduced slope (the application load is constant). The Type II part shows the waiting time increase caused mainly by the processing of messages in other VI queues, while the slope in the Type III part captures the effect of the delay caused by unsuccessful NIC polls.

Both graphs underscore the effect of declared VIs on the latency of a message. Further, the latency seen by messages on one VI grows with the time spent by the NIC in processing messages on other VI channels. It is thus important to not only lower the overheads in the constant offered load part of the above curves, but also reduce the

latency of messages in the initial increasing load portion, to enhance the scalability of the system. The following subsections investigate a range of issues towards this goal.

4.3.4 Size of Descriptor Payload

It is well-known [72] that there are trade-offs in supporting large payload sizes within descriptors. A larger size can help avoid additional DMA operations when messages can fit within this size. Based on experiments, previous work has looked at ideal payload sizes for a given set of hardware/application parameters. We would like to point out that there is one more (scalability) factor that needs to be taken into consideration in determining the payload size when implementing VIA on a Base NIC.

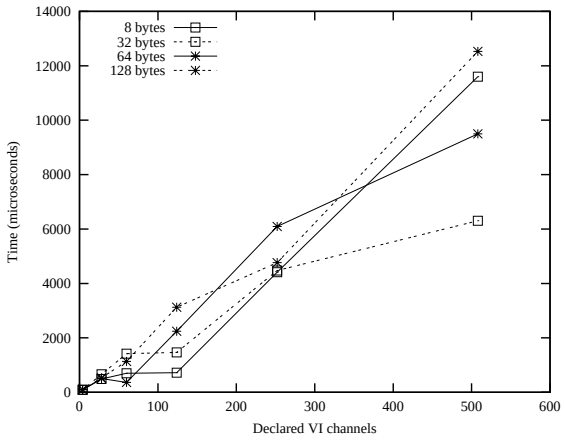


Fig. 4.6. Descriptor Payload Size (Base, u=d)

To demonstrate this issue, a Type II workload with immediate messages of various sizes is used. To further isolate the effect of NIC event processing overheads, the shadow queue described in Section 4.2.2 is used so that the NIC does not need to DMA the

descriptor down after it detects a message waiting to be sent on a VI. Results are shown in Figure 4.6 for 8, 32, 64 and 128 byte messages. In each case, the data fits within the descriptor, and so the number of DMA operations (at the receive end) incurred is the same. The results indicate that a 32 or 64 byte message can incur a lower latency than a message of 8 bytes for 508 declared channels. This non-intuitive result can be explained by examining what happens on the NIC. Since it polls each queue in round robin order, if the card finishes processing one message fast enough, it can miss the posting of the send on the next VI (that has to wait a full cycle of doorbell checks). With a little longer message, this does not happen as the card takes longer to finish processing the send (it also takes longer for the host to finish posting its send but that is not enough to compensate for the other factor).

While this result is a special case which may not occur in practice, it does serve to illustrate that descriptor payload sizes can not only affect raw latency, but can also result in the above discrepancies when scalability is considered. Hence workload specific setting of the size may be considered if the NIC memory and host permit larger sized descriptors.

4.3.5 Effect of Completion Queues (CQ)

The workload used here is of Type I where the application process picks a VI at random to send a message, and then waits (polls) to receive a message on any of its declared VIs. Such a functionality, similar to the UNIX *select* call, could be useful to several applications. The host receive operation is the counter-part of the operation where the NIC had to check for posted doorbells. Without a CQ, the host has to poll all the VIs it owns for an incoming message, and the resulting cost goes up linearly with

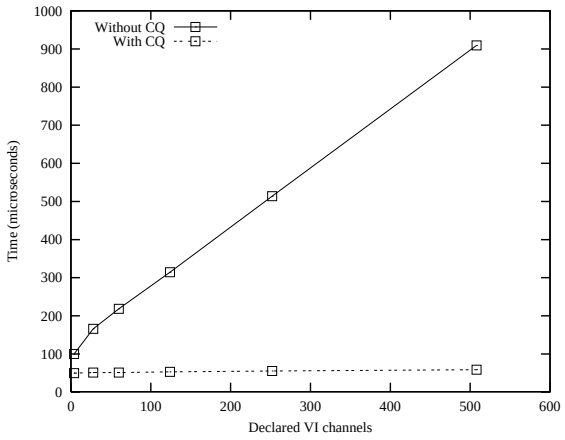


Fig. 4.7. CQ benefits (Base, u=1, 128 bytes)

the number of declared VIs (half these VIs may need to be polled on the average). With a CQ, the host has to poll just this queue, making the cost relatively independent of the declared VIs. The use of CQs is not without its cost as is seen by Stages 11 & 12 in Figure 4.3. So the benefits to the application would need to overcome the cost of the additional DMA.

Figure 4.7 illustrates these points. Again, while the specific workload brings out the contrast sharply, the results stress the importance of having an efficient implementation of completion queues.

4.3.6 Pipelined/Overlapped firmware design

The wait for completion of DMA operations wastes the CPU resource on the Base NIC (Stages 4, 6, and 8 of the pipeline in Figure 4.3). Overlapping DMA operations with the processing of another VI can improve the throughput and scalability of an implementation. To study this, we model a NIC firmware which maintains a software

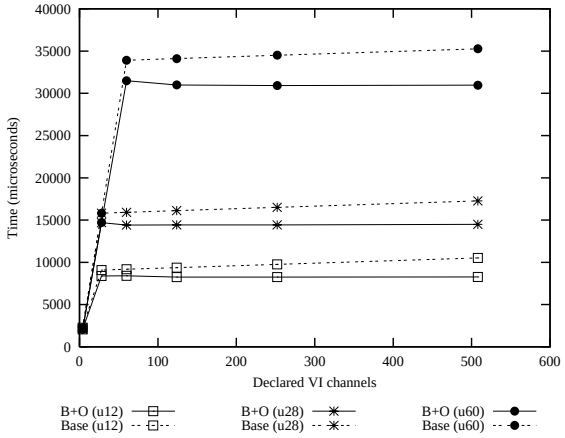


Fig. 4.8. Base+O vs. Base (4KB)

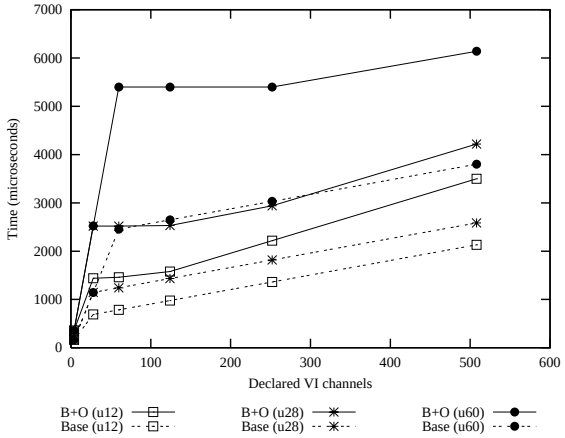


Fig. 4.9. Base+O vs. Base (128 bytes)

task queue for each of the DMA engines. An event (such as detection of a doorbell) causes an entry to be inserted into the task queue of the appropriate DMA engine (if it is not free). Its DMA operation will be initiated whenever it reaches the head of the queue. The firmware is thus reduced to an event processing loop which schedules work

to and from each of these queues, and this model is referred to as Base+O. It is to be noted that the queue is maintained in software, and the hardware option is explored in the next subsection.

Figure 4.8 compares the latencies of Base+O (abbreviated to B+O in the graph) with Base. The workload is of Type III where 12, 28 and 60 VIs are active/used, labeled u12, u28 and u60. The benefits of pipelining are evident in all the workloads, with the benefit becoming more pronounced at higher loads (larger number of active VIs). A larger message size of 4 KB is used in these experiments, which allows a larger scope for overlap. The pipelined design used here allows the DMA engine transfers to overlap for either the same message (if scatter-gather is used), or different messages from the same or different VIs. Only the latter effect of overlapping messages from different VIs has been shown here.

To observe the effect of the length of the DMA operation, the same workload is exercised with smaller (128 byte) messages in Figure 4.9. Here, the results are reversed, with the latencies increasing (more for higher loads) for a pipelined firmware. The length of the DMA operation is not sufficient to overlap with processing, and the overhead of queue/state maintenance actually degrades performance. The benefits of a software pipeline would depend on the length of Stages 6 and 8 in Figure 4.3. The lengths of the DMA transfers in Stage 4, 10 and 12 are independent of the message size for non-immediate messages. As such they offer little scope for overlap.

4.3.7 Hardware DMA Queues

There are a couple of disadvantages with the pipelined firmware operation in the previous section. First is the cost associated with software queue maintenance. Second is the lag/gap between the completion of a DMA operation and the detection of that condition by the firmware to initiate the next operation (as opposed to the situation in Figure 4.3 where Stages 5, 7, and 11 can immediately succeed Stages 4, 6, and 10 respectively). While one could use an interrupt on DMA completion, interrupt processing can itself add to the overheads.

A solution to ameliorate these problems is to have hardware support for queuing of DMA entries, such as in [53]. Not only does this lower queuing costs, the DMA can directly proceed to the next operation when the previous operation gets done. We refer to this model as Base+O+DMAQ. In this model, while the Host DMA and Network Send DMA engines are permitted to queue a large number of entries, the Network Receive DMA is restricted to a queue length of 2. The underlying rationale is that since incoming message lengths are unknown, each entry put in the Net Receive DMA queue must have a buffer of network transfer unit size associated with it. To put restrictions on the buffer space availability for incoming messages, we restrict the hardware queue length.

We compare the benefits of a firmware designed with this hardware support to Base+O. Figure 4.10 shows that this hardware support (abbreviated as B+O+D) does not help for larger messages (4 KB). We observed earlier that the software solution was itself doing fairly well in this case (Figure 4.8) because the length of the DMA operation masks the effect of any overheads. On the other hand, Figure 4.11 shows the

benefit of such support for short messages (128 bytes or lower) with the effect being more pronounced for higher loads. However, these latencies are still worse than a non-overlapped design (compare to Figure 4.9).

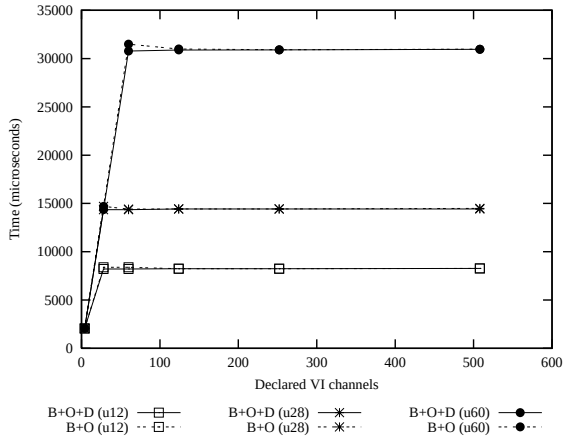


Fig. 4.10. Base+O+DMAQ vs. Base+O (4KB)

4.3.8 Hardware Doorbell support

The provision of hardware support for doorbells enables the NIC to avoid polling all declared VI channels. In terms of Figure 4.3, Stage 2 can be eliminated completely. Figure 4.12 demonstrates the significant improvement in latencies with this feature (Door+O+DMAQ) (abbreviated as D+O+D) for a Type III workload using a varying number of active VIs (12, 28 and 60) and 128 byte messages, compared to Base+O+DMAQ (the only difference between the designs being compared is the hardware doorbell support). We find that the design with hardware doorbell support keeps

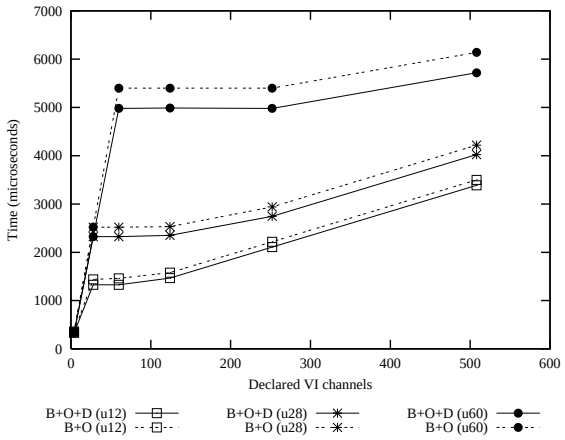


Fig. 4.11. Base+O+DMAQ vs. Base+O (128 bytes)

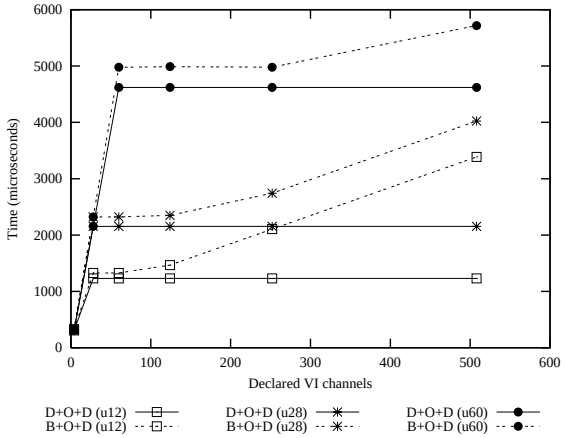


Fig. 4.12. Door+O+DMAQ vs. Base+O+DMAQ - Latency (128 bytes)

the latency constant for an offered application load, while the latency grows with the number of declared VIs (even when the offered load remains constant) for the cases without hardware support. This directly results from the wait time experienced by a message before it is noticed by the NIC, as can be seen in Figure 4.13.

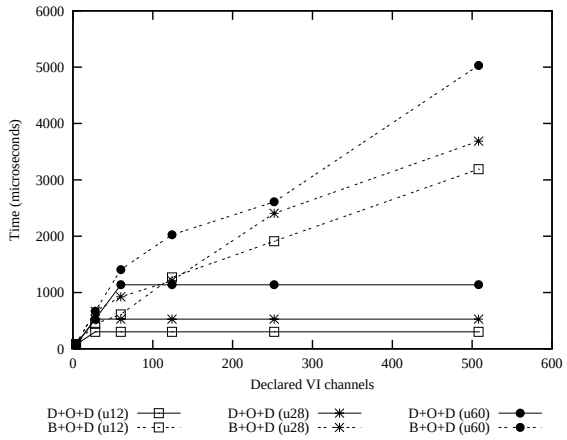


Fig. 4.13. Door+O+DMAQ vs. Base+O+DMAQ - Wait time (128 bytes)

Figure 4.12 gives further interesting insights. Focusing on the curves for the designs without hardware doorbell support, we see an initial steep increase as was the case in Figure 4.11 due to the increasing load on the system. When the declared VIs increase with no further increase in the offered load, the latency at first increases very slowly. This relatively flat (modest increase) part of the curve is due to the overlap of the doorbell polling by the NIC processor with the DMA operation. With a larger load, higher is the possibility of this overlap, which is the reason why this region is longer for the higher loads. After a point, the number of declared VIs becomes large enough that the redundant polling simply outgrows the overlap with DMA operation, causing the latency to go up again.

The benefits of hardware doorbell support are not really felt by long messages (the graphs are not explicitly shown here). From Figure 4.10, it is seen that the curves automatically flatten out for a certain offered load with long messages, because of the

sufficient overlap. Hence hardware doorbell is not really needed in such cases, and the overlap can achieve the same effect.

4.3.9 Tailgating Descriptors

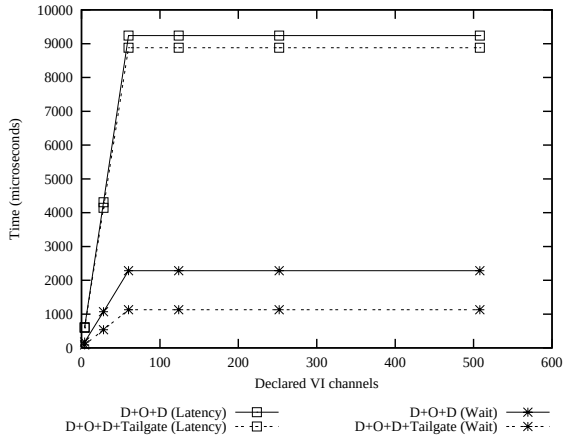


Fig. 4.14. Tailgating (Door+O+DMAQ, 128 bytes)

We can see from the message sending sequence that 2 host DMA operations are incurred - one for the descriptor and the other for the actual data. While this may appear inefficient, the advantage is that it places little storage overheads on the card memory. This approach is used in [19], while U-Net [93] uses the other approach of keeping the descriptor queue entirely on the card. A variation of this option is evaluated later in Section 4.3.11. One could use a hybrid approach between these two of keeping a certain number of descriptors on the card, and then chaining on to a descriptor queue maintained in host memory only when the queue on the card overflows. We propose an alternative solution to this problem, where a descriptor is appended/chained to the data

part of the previous message on that VI if the data part has not already been DMA-ed down to the NIC. This way, while the DMA operation may itself take a little longer, the costs associated with initiating and transferring the descriptor separately are avoided i.e. stages 3 & 4 of Figure 4.3 are avoided. This mechanism will, however, incur two DMA transfers whenever there is no previous data part to append the descriptor with. Further, there are some race conditions between the host and NIC to deal with.

We augment the design in the previous subsection with this new enhancement and subject it to a Type IV workload (using two successive sends/receives on a VI). Figure 4.14 shows both the latencies and wait times in the send queues (for 128 bytes messages) with and without this enhancement. There is a reasonable improvement with this enhancement, though it should be noted that in this workload nearly half the messages are likely to avoid stages 3 & 4 of the send pipeline. In general, this fraction will be workload specific, getting larger if the application sends long or frequent messages on a VI, and becoming smaller if successive messages on a VI are spread out in time. The difference between wait times on the SendQ with and without tailgating is larger than the corresponding difference for latencies because the saving in wait times is somewhat offset by the contention for the NIC resources.

4.3.10 Separate Send and Receive Processing

We can stretch the hardware support even further by providing two relatively independent pipelines for processing sends and receives respectively, so that these logically separate operations interfere with each other minimally. This can be achieved by providing two processors and two hardware doorbell queues, one each devoted to the

send and receive pipelines. A hardware doorbell queue is not strictly needed for receive doorbells since they need be answered only when a message comes in for that VI, and the message destination can be used to directly get to the corresponding doorbell. However, memory management of NIC buffers is easier with a unified way of answering send and receive doorbells. The separate Send and Receive DMA engines to/from the network (present even in the Base NIC) ensures that the only point of contention between the two pipelines is the Host DMA. While this could be alleviated to some extent by providing multiple DMA channels within the same DMA engine, we have not modeled it.

Figure 4.15 shows the large improvements in latencies seen with this enhancement. The improvement is more pronounced with higher operating loads.

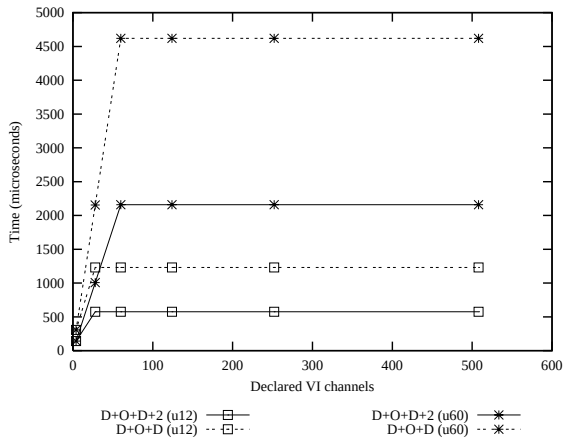


Fig. 4.15. Door+O+DMAQ+2CPU vs. Door+O+DMAQ (128 bytes)

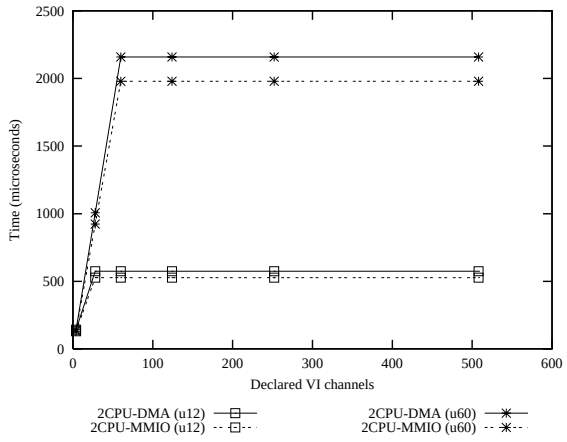


Fig. 4.16. DMA vs. MMIO using Shadow Queues (Door+O+DMAQ+2CPU, 128 bytes)

4.3.11 Shadow Queues

Further improvements for a hardware supported pipelined firmware are possible by balancing the pipeline segments to improve throughput and, consequently, latency seen across VIs. Using the shadow queues (with MMIO) that has been proposed in Section 4.2.2 eliminates the small sized DMAs for descriptors from the critical path, giving the opportunity for further overlaps in processing. This achieves the same goal as the Tailgating descriptor scheme described earlier, though the DMA for the descriptor is always avoided in this case (not just when there are messages ahead of it in the VIA that have not yet been DMA-ed). The reduction in latency seen as a result is shown in Figure 4.16. The workload used here is of Type III with 128 byte messages. As mentioned earlier, a shadow queue element does not need to store all descriptor fields, and can have a shorter lifetime than the corresponding descriptor.

4.3.12 Putting it all together

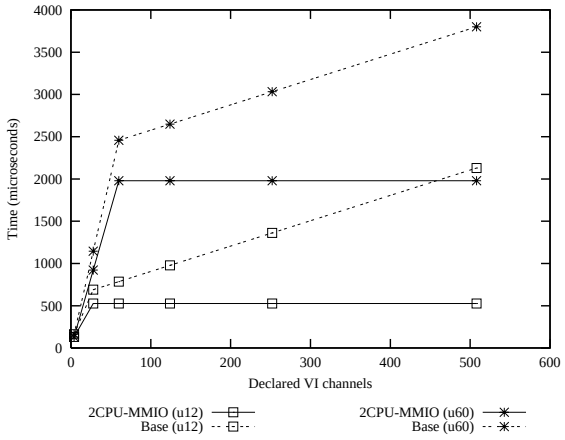


Fig. 4.17. Door+O+DMAQ+2CPU vs. Base (128 bytes)

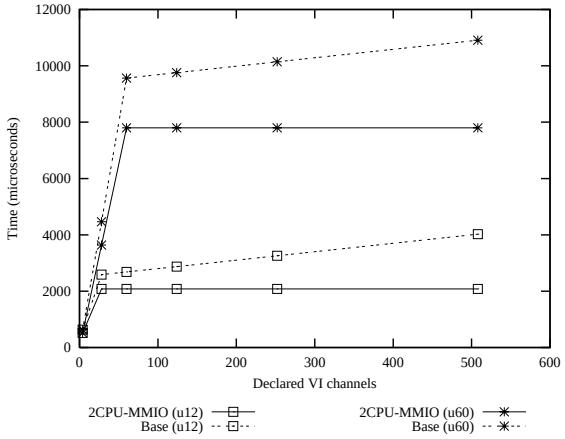


Fig. 4.18. Door+O+DMAQ+2CPU vs. Base (1KB)

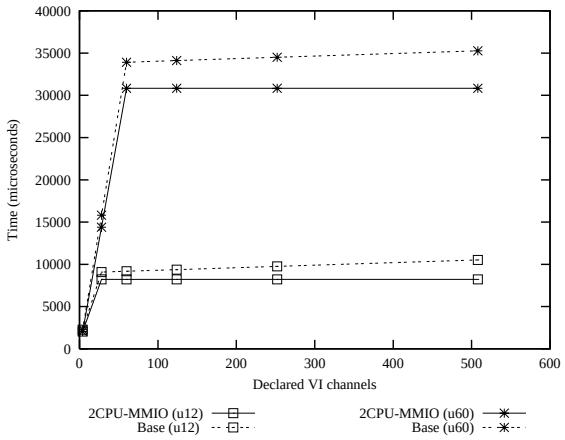


Fig. 4.19. Door+O+DMAQ+2CPU vs. Base (4KB)

Through this section, we have progressed from a Base NIC which provides minimal hardware support (a single host DMA engine, two network DMAs, some memory, and a programmable processor) to a sophisticated NIC (Door+O+DMAQ+2CPU) supporting DMAs with hardware queues, hardware doorbells, and separate processors to handle sends and receives respectively. Correspondingly, the firmware has also moved from a sequentially mode of operation, to one which does a sophisticated overlap of operations together with shadow queues to even eliminate some parts of the messaging pipeline. We have incrementally made this progression, showing the benefit of each enhancement along the way. We can put the overall benefits of these enhancements in perspective by comparing the sophisticated design with the baseline. Figures 4.17, 4.18 and 4.19 show the improvements with the enhancements consistently for the different message sizes (128 bytes, 1KB and 4KB) respectively. We find the improvements materializing not just in the increasing load region of the curves (the initial steep part), but also continuing in

the constant load region. The improvements are seen for both low (12 active VIs) and high (60 active VIs) offered loads.

4.4 Summary

By standardizing the ULN interface exported to the applications, VIA has potential for bringing user-level networking to the mainstream. VIA can also serve as the underlying platform to develop conventional protocol stacks leading to its adoption in LAN and WAN environments.

In addition to low latency and high bandwidth on a single channel, scalability is an important consideration, especially as multiprogramming levels continue to increase and each executing application keeps several open communication channels to avoid setup and tear-down costs.

In this chapter, we have taken a simple baseline NIC and firmware and have progressively refined it, one enhancement at a time, to study the impact on scalability. This is, perhaps, the first study to examine different hardware and software design alternatives within a unified framework. Each enhancement exercise has given interesting insights that can be useful to hardware designers for incorporating scalability-aware features, and to software developers for adapting/exploiting the underlying hardware based on application characteristics, towards a scalable VIA implementation.

Chapter 5

Quality of Service for VIA

Clusters are gaining acceptance not just in traditional scientific applications that have needed supercomputing power, but also in domains such as databases, web service and multimedia. Many of these emerging applications/environments place diverse Quality-of-Service (QoS) requirements on the underlying system, over and beyond the higher throughput and/or lower response time that is usually desirable. However, existing hardware and systems software for clusters are not adequately tuned to handle these diverse demands, and a careful design at several levels of the system architecture is necessary for the successful deployment of clusters in these different (and emerging) application domains. This part of the thesis takes an important step towards this goal by examining issues in developing QoS-aware communication for clusters. Having looked at ULNs from a performance and scalability perspectives earlier, it is a natural progression to look at the issue of providing guaranteed service.

The issue of guaranteed service in user-level networking is largely unexplored in research literature. In this work, QoS is a broad term that is used to describe the desirable performance criteria for a given communication channel. This could include (a) low latency, (b) high bandwidth/throughput, (c) (deterministic or statistical) guaranteed latency (bound on the interval between injection at one node and ejection at another), and (d) (deterministic or statistical) guaranteed bandwidth (over a period of time).

The first two criteria are the typical metrics that others have tried to optimize for a particular channel and which have been the focus of earlier chapters. We refer to network traffic which is concerned with these metrics alone as *Best-Effort*. Guaranteed latency is needed for mission-critical applications, and we refer to the traffic generated by them as *Real-Time*. Finally, guaranteed bandwidth is a desired feature for many multimedia applications where messages are injected/ejected at periodic intervals. We refer to the corresponding traffic as *CBR* (Constant Bit Rate) or *VBR* (Variable Bit Rate) traffic. In this work, we assume that a communication channel is used exclusively for either one of these three kinds of traffic and refer to them as Best-Effort (BE), CBR and VBR channels. The term QoS channel is broadly used to refer to the latter two.

There are several applications on cluster environments that can benefit from QoS-aware communication. A common example is multimedia streaming (Video on Demand, instructional videos, etc.), where the cluster is used to hold multimedia data [88], which is then streamed to one or more clients on demand through the network. There could be a bridge/gateway between the cluster network and the LAN/WAN - such boundaries between different networks are expected to gradually fade with emerging technologies such as Infiniband [54]. Another example is multimedia databases where the data is not necessarily streamed to a client but may be sent to one or more machines in the cluster itself. The searching and processing of information in such multimedia databases can benefit from QoS channels. Finally, a large class of vision applications that process real-time video can benefit significantly from the high performance capabilities of a cluster. For instance, detection of obstacles in an aircraft's flight path requires computationally intense algorithms, and assigning different tasks of these algorithms to the cluster nodes

and streaming the video frames through these piplelined tasks has been shown to meet the real-time processing needs that is currently not possible on a single workstation [98]. Clusters could thus be deployed as servers to run these different applications in a multiprogrammed manner, making it critical to ensure that the QoS demands of each application are met. While the primary motivation for this work stems from the need for QoS aware communication on clusters, it should be noted that ULNs are being employed to layer protocol stacks for LAN/WAN environments as well, and the results from this research can be useful there.

5.1 Issues in providing QoS aware communication

Developing QoS-aware communication for a cluster requires a substantial design effort at several levels of the system architecture. At one end, the network fabric linking the nodes of a cluster needs to recognize the QoS demands of the communication channels at a global level (across all nodes), and should cater to each of them. While there is a large body of literature on providing QoS support for multiple traffic types in LAN/WAN networks, it is only recently that this issue has been under investigation for cluster networks [57, 58, 39, 39, 76, 36, 21, 8, 62, 85, 47, 24, 56, 101]. Since cluster networks are invariably routed by circuit-switching or cut-through mechanisms (such as wormhole routing) to lower latencies, the solutions have been somewhat different from those for LAN/WAN environments which are typically packet switched. QoS-based arbitration and scheduling (and, perhaps, pre-emption) of contending message flits on output links has been shown to work quite well with circuit-switching [36, 21, 8] and wormhole routing mechanisms [62, 85, 47, 56, 101]. Another technique that had been used in

an off-the-shelf wormhole routed network (Myrinet [16]), without requiring any hardware modifications, has been to periodically synchronize the entire network (using the firmware on the network interface cards) and schedule packet movements appropriately [24].

Another level of system architecture that is important for providing QoS to applications is the operating system at each node. It needs to be aware of the QoS requirements of the processes corresponding to the applications and allocate the CPU(s) and buffers so that messages can be injected/ejected at a desired rate. Some work has been done on QoS-based CPU scheduling for single node systems [49]and a vast body of literature exists for real-time scheduling for parallel systems. However, there hasn't been much work done in developing distributed QoS-aware schedulers for a cluster.

The scope of this work is the network interface and communication software at each node of the cluster which lie in between the two levels mentioned. Very few prior studies [2, 60, 96] have examined this issue, but none in the context of VIA/ULN. For the purposes of this discussion, we assume that the network fabric is already QoS-aware and can meet the QoS requirements for all channels using it. We also do not consider the issue of the operating system being QoS-aware. It is assumed that the relevant processes can be scheduled and inject/eject messages as needed. For the purposes of experimentation and measurement under these assumptions, it is not essential to provide a QoS-aware network or OS. It is sufficient to provide enough resources (network bandwith, CPU time and buffers) at these levels that they do not become the constraining factor in the flow of messages from one application/process to another.

The absence of the OS kernel in the critical path of a message send/receive, a defining feature of a ULN, complicates the task of making the communication software QoS-aware. The only trusted entity left on the critical path is the firmware running on the network interface card (NIC). Now it has to perform the additional task of regulating the outgoing traffic so that it can throttle channels that are exceeding their negotiated usage of communication resources. At the same time, the firmware has to carefully schedule the processing of messages on various channels so that it can meet their varying QoS demands. Even without QoS requirements, the firmware needs to coordinate a number of activities on the NIC such as host DMA transfers, outgoing network transfers, incoming network messages, buffering and some protocol processing. Incorporating the QoS needs of channels into this coordination increases its burden considerably. Typically, the firmware is executed on much less powerful hardware than the host (typically a single processor, slower, smaller memory etc.) So it needs to be designed very carefully to perform additional tasks while maintaining acceptable levels of raw channel performance.

Compared to the firmware, the kernel driver is simpler. Its job, as with Best-Effort channels, is to establish channels and convey the QoS requirements to its NIC/firmware. Before doing this, it needs to ensure that its local NIC can meet the desired QoS requirement without degrading the service for the channels already established. It also needs to communicate with the NIC/firmware of the destination channel to ensure that it can handle the requirement without degradation of service. If both ends are able to handle the load, then the channel is setup and any necessary information (period/rate, deadlines etc.) is conveyed to the firmware. Otherwise, channel setup is denied. The principal task of the device driver is to do admission control.

5.2 Overview of ULN and NIC Operations

We describe the ULN and NIC operations by discussing the implementation of VIA, which is also the basis for the evolving Infiniband architecture on a Myrinet interface. Our choice of conducting this exercise using Myrinet rather than a hardware VIA capable NIC [48, 46] is due to the following reasons. Myrinet provides high hardware transmission rates and is quite popular for deploying clusters. While this interface provides several hardware features to make VIA implementations more efficient, it is fairly generic (representative of evolving interfaces) and is not overly specific to VIA. Providing QoS aware scheduling on NIC operations may also be easier and flexible (to change on the fly) with a firmware-based approach than a hardware implementation.

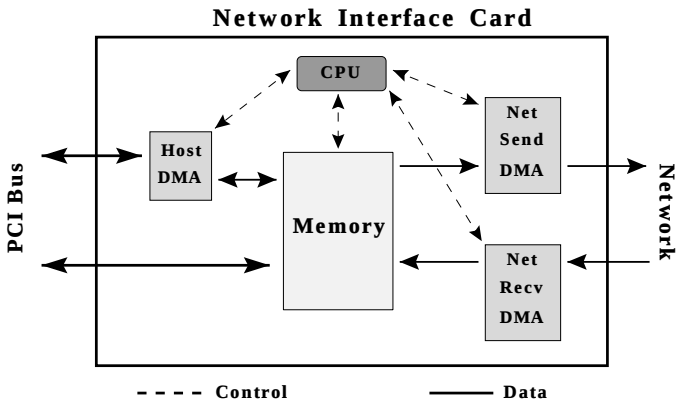


Fig. 5.1. Myrinet NIC

Figure 5.1 shows the network interface card for Myrinet again. In this chapter, the NIC processor is called the LANai and the Host DMA engine , which is used to

transfer data between host memory and card buffer, is referred to as *HDMA*. The Net Send DMA engine, which sends data out of the SRAM onto the network (wire), is called *NSDMA* and the corresponding Net Receive DMA engine is referred to as *NRDMA*.

VIA, as described in Chapter 4.1, serves as the interface between the hardware named above and the application.

The following actions are important considerations when implementing a ULN and making it QoS aware:

- How does an application process notify the NIC about a message that it wants to send, or how does the NIC detect the presence of an outgoing message? Similarly, how does it inform the NIC about a message that it wants to receive?
- How and when should the NIC bring down the message from the host memory down onto its local buffer? Subsequently, when should this message be sent out on the network?
- How and when should the NIC pick up an incoming message from the network? Subsequently, when should this is transferred up to the host memory?

The hardware features on the Myrinet which help us implement these actions are discussed next.

5.2.1 Doorbells

Event (send or receive) notification to the NIC by application processes is done in VIA through the *doorbell* mechanism. Each VIA channel has two sets of doorbells, one each for send and receive. When an application wants to send or receive a message, it

creates a header for it (called a *descriptor*), makes this accessible to the NIC, and then rings a doorbell for that channel. An important consideration is how should the doorbell information be conveyed to the NIC. In the earlier Myrinet interface (LANai 4), the only option is to have a software queue (of doorbells) associated with each channel (on the NIC SRAM), and have the LANai poll the queue. Each channel needs to have a separate queue due to protection constraints. However, with this approach, the performance degrades with the number of channels since the LANai has that many queues to poll, even if there are no doorbells on them, as shown in Chapter 4. The newer interface (LANai 7) provides a hardware queuing mechanism whereby (doorbell) writes to different channels can be merged (by hardware) into a single queue, which can then be polled by the LANai. This avoids polling queues of VI channels without any posted doorbells. However, one of the problems with this mechanism is that doorbells are dequeued in FCFS order, and can lead to additional overheads if the implementation needs to service messages in a different order (based on QoS requirements).

5.2.2 HDMA Operation

Once a doorbell (send or receive) is detected, the LANai first needs to get hold of the descriptor before it can get to the corresponding data buffer. It is more efficient to keep the descriptors on the NIC SRAM so that it can be easily accessed by the LANai. However, the NIC SRAM is a precious resource, and it is not desirable to devote too much space for descriptors (this also removes any assumption about the hardware resource availability in the VIA specification). As a result, some of the VIA implementations [19] keep the descriptors on the host memory, and this needs to be DMA-ed (using HDMA)

down to the card SRAM. The LANai then examines the descriptor before it DMAs (again using HDMA) down (for a send, with the reverse direction used for a receive) the data. Host memory locations for descriptors and buffers in VIA are specified as virtual addresses. Address translation tables need to be maintained (our implementations keep these tables on NIC SRAM) to translate them to physical addresses before programming the HDMA.

If the HDMA is busy, the LANai needs to wait (or do some other useful work in the meantime) before programming it. This can lead to inefficiencies in its operation. As a result, the newer interface (LANai 7) offers a hardware queuing mechanism for DMA operations as well, wherein the LANai can simply insert a DMA request in a queue, and can in its own time come back and check the status for completion. It should be noted that the same HDMA is used for: (a) bringing the send descriptor down onto the SRAM, (b) bringing the data to be sent down onto the SRAM, (c) bringing the receive descriptor down onto the SRAM, and (d) sending the incoming data up to the host memory.

5.2.3 NSDMA and NRDMA operation

These DMAs on the Myrinet NIC do not provide hardware queuing features. If the LANai needs to use the NSDMA when it is busy, it has to maintain its own queue (in software), and come back periodically to check for availability. The outgoing transfer rates are very much dependent on the network fabric speeds and the contention on the network. An incoming message needs to be DMAed by the LANai on to the SRAM (using the NRDMA), the channel id is extracted from the header and the receive descriptors are checked for a match. If there is a matching descriptor, then the data transfer up to

the host can be initiated right away using the HDMA. Otherwise, a descriptor needs to be brought down before the data can be transferred.

5.3 Firmware Design

The firmware running on a NIC has been established to be a key component for providing QoS support to user level networking. This section revisits firmware design in view of the new objectives. Henceforth, the firmware running on the NIC is called the LCP (LANai Control Program).

The sequence of steps performed on a message send has already been described in detail in Section 4.3.2. We briefly restate these as the operations that an LCP performs for messaging alone :

- Poll doorbell queue(s) for an application's send/receive notification.
- Transfer the descriptor associated with the doorbell from the host memory down onto the SRAM using HDMA.
- Transfer the data associated with a send descriptor from host memory to SRAM using HDMA.
- Transfer the packet out onto the network using NSDMA.
- Pick up packet from network using NRDMA.
- Transfer data from SRAM to the host memory using HDMA.
- Transfer completion information (of send/receive) to host memory using HDMA.

From the design point of view, we look at three ways of structuring these operations to meet application QoS demands :

5.3.1 Sequential VIA (SVIA)

The simplest way of structuring the LCP is to perform all the operations mentioned in a fixed sequence. This corresponds to the Base design of Section 4.3.3. The advantage of such a design from a QoS standpoint is its simplicity. Since there is no overhead for making explicit QoS-related decisions, the firmware can offer better raw performance. This increased performance could be sufficient to meet QoS demands for some loads.

5.3.2 Parallel VIA (PVIA)

Another way of reordering LCP operations to meet QoS needs is to make it completely event driven.In this approach, called PVIA, the LCP processes events from the three DMA engines and the doorbell queue without regard to the ordering imposed by a single send or receive. It is possible, for instance, to initiate the network send DMA for one message followed by the host DMA receive for another's descriptor followed by the network receive DMA for a third message. PVIA consists of the design described earlier in Section 4.3.6 with the slight change that the HDMA incorporates the hardware DMA queue described in Section 4.3.7. The advantage of PVIA is the increased throughput of message processing which improves the NIC's ability to meet bandwidth related QoS specifications.

5.3.3 QoS-aware VIA (QoSVIA)

The key to providing differentiated service to channels is being able to preempt the processing of one message in favor of another which has a greater QoS demand. Studies for QoS provision in multi-hop networks rely on this reordering of message processing at each hop to ensure that end-to-end guarantees are met. The various stages of the message processing pipeline within a NIC can be similarly viewed to be *preemption points* at which a higher priority message can be processed preferentially, regardless of the order in which the processing ended at the previous stage. Both SVIA and PVIA do not address this important aspect directly, though PVIA is a step in that direction since it permits the processing stages of different messages to be interleaved. Once interleaving or reordering is possible, different QoS requirements can be met by appropriate scheduling of operations of different message streams (represented by channels) at each of the four service points in the pipeline (Doorbell queue, HDMA, NSDMA and NRDMA).

In QoSVIA, we take the PVIA implementation and augment it with some scheduling algorithms to determine channel priorities. VirtualClock [104] is a well-known rate-based scheduling algorithm that can be used here to regulate the allocation of resources to different channels based on bandwidth requirements. The idea of this algorithm is to emulate a Time Division Multiplexing (TDM) system, and its adaptation to QoSVIA is outlined below.

When a VI channel is opened (this is a driver call), the application also specifies the desirable bandwidth if it needs to use it for CBR/VBR traffic (in the VipCreateVI

call). The driver then needs to communicate with both its NIC as well as the destination NIC to find out whether they can handle the load (without degrading service of other channels). If they cannot, then an error is returned to the application, which can then try to renegotiate. In our current implementation, we do not implement this admission control mechanism (incorporating this is part of our future work), and find admissible loads for the NIC through trial and error. It may be noted that the lack of admission control does not impact performance results since it is not in the critical path of communication.

The driver translates this bandwidth information into a period, called `Vtick`, which essentially specifies the ideal inter-arrival/service time between successive packets for a pre-determined packet size. A smaller `Vtick` specifies a higher bandwidth requirment. For Best-Effort channels, `Vtick` is set to infinity. The `Vtick` for the channel is then conveyed to the LCP, which keeps this information together with the other channel specific data. In addition, the LCP maintains another variable, `auxVC` for each channel, and calculates this value when it notices a doorbell for that channel (i) as follows:

$$auxVC_i \leftarrow \max(\text{current time}, auxVC_i)$$

$$auxVC_i \leftarrow auxVC_i + Vtick_i$$

$$\text{timestamp the doorbell with } auxVC_i$$

The VirtualClock algorithm specifies that packets be served in increasing timestamp order to allocate bandwidth for resources proportional to their specified requirements.

Once the time stamp is determined, the next issue is to figure out how to allocate the resources, based on these timestamps. The most important resources are the HDMA and the NSDMA since they are both entirely controlled by the firmware. The Doorbell queue and NRDMA are each influenced partially by agents outside the firmware's control (the application and a network wire/switch respectively) so it is limited in its ability to manage them. It should be noted that the firmware can exert backpressure on the application and the network fabric by the rate at which it services doorbells/packets placed in the respective queues.

When the doorbell on a channel is posted, it is timestamped using the VirtualClock algorithm, and the corresponding descriptor is inserted in the hardware HDMA queue in timestamp order. Similarly, packets in the software NSDMA queue are maintained in time stamp order. One could hypothesize that maintaining the NSDMA queue in FCFS order is sufficient to maintain the ordering imposed by the previous pipeline stage. However, hardware constraints and the need to avoid race conditions prevent the preemption of a message which is already receiving service at the HDMA or even a few that immediately follow it. It could so happen that a doorbell for a message with a more critical deadline (higher priority) arrives later than a message currently being serviced (or already serviced) by the HDMA. Keeping the NSDMA queue in timestamp order helps us to correct the inaccuracies in the ordering of messages at the HDMA queue. This is also in line with the philosophy of maintaining multiple preemption points in the message processing pipeline.

The basic difference between PVIA and QoSVIA is only in the time-stamping of events using the VirtualClock algorithm and the possible insertions into the middle of

the HDMA/NSDMA queues (PVIA inserts only at the tail). There is a cost that is paid in performing these operations and our experimental evaluations compare the benefit gains with the incurred costs.

It is well understood [103] that a rate-based service discipline, such as VirtualClock, can potentially starve best-effort channels (as long as there is demand from a CBR/VBR channel). There are alternate scheduling mechanisms, such as hierarchical schedulers [49], that one could use to avoid some of these problems. Conversely, there is a need to limit the scheduling overhead on the LANai. In this work, we have used a simple scheme like VirtualClock, only to demonstrate the feasibility of supporting QoS-based traffic in the NIC. Even with Virtual Clock, there are operating points for the offered load where best-effort channels can get reasonable service. Our results will show how the presence of CBR/VBR channels can affect the performance for best-effort channels.

5.4 Performance Results

PVIA and QoSVIA have been implemented and evaluated on both an experimental platform as well as using simulations, and compared with SVIA. For SVIA, we use a publicly distributed version [19] which is representative of several other implementations optimized for low latency on a channel. The evaluation platform consists of a Pentium/Linux cluster connected by a Myrinet switch. Each node of the cluster has dual 400 MHz Pentium Pros with 256MB memory and a 66 MHz 64 bit PCI bus.

The metrics of concern are the (a) *jitter fraction* (denoted as θ) for the QoS (CBR or VBR) channels, and (b) *1-way latency* for messages on Best Effort (BE) channels. Jitter fraction is defined as follows. Let us say a QoS channel has negotiated a certain

bandwidth (e.g. an inter-arrival time for a specified message size) with the underlying system. Assuming that the workload generator can operate at the negotiated rate, when messages are pumped into this channel, the application expects the messages to be separated by at most the negotiated inter-arrival time at the receiver end. Else, it incurs a jitter for that message, with a *jitter duration* equal to the difference between the inter-receipt time of the messages and the negotiated inter-arrival time. Jitter fraction is a normalized version of this metric, where the mean jitter duration (over all the messages) is divided by the negotiated inter-arrival time. While we have obtained the mean jitter duration and the number of messages incurring jitters in the experiment, the results shown below are specifically for θ which we feel is more useful from the application's viewpoint (for example, a 1 ms jitter duration on a 1 MByte/sec channel is not as significant as a 1 ms jitter duration on a 10 MByte/sec channel, and θ captures this difference). The number (or fraction) of packets incurring jitters is also an issue that we consider in some cases.

5.4.1 Results from Experimental Platform

5.4.1.1 Raw Performance of VIA Implementations

Before we examine the QoS capabilities of the three VIA implementations, we first examine the raw performance (typically Optimized in ULNs), to see how much overhead is added by PVIA and QoSVIA on BE channels without any QoS traffic. We investigate this issue by first giving the breakup of the 1-way latency (generated by using a simple ping-pong experiment on 1 channel between two machines) for SVIA, PVIA and QoSVIA.

NIC Operation	SVIA				PVIA				QoSVIA			
	4	1K	4K	16K	4	1K	4K	16K	4	1K	4K	16K
Send												
HDMA Prog. (Desc.)	3	3	3	3	7	7	7	7	9	9	9	9
Desc. Down	6	6	6	6	11	10	11	10	13	13	13	13
HDMA Prog. (Data)	9	9	9	11	19	19	23	34	24	24	30	49
Data Down	11	18	43	141	23	28	52	148	27	33	57	152
NSDMA Done	12	25	69	244	28	39	82	254	32	43	86	261
HDMA Prog. (Status)	13	27	71	245	33	45	87	260	37	49	91	266
Status Done	14	28	72	247	38	49	92	264	41	53	95	270
Receive												
HDMA Prog. (Desc.)	3	3	3	3	0	0	0	0	0	0	0	0
Desc Down	5	5	5	5	0	0	0	0	0	0	0	0
HDMA Prog. (Data)	8	8	9	11	10	10	14	24	12	12	17	35
Data Up	10	17	42	138	14	19	42	138	15	21	44	139
HDMA Prog. (Status)	11	18	43	140	19	24	47	143	20	25	48	143
Status Done	12	20	44	141	23	28	51	146	24	29	52	147
1-way Latency	30	53	121	385	69	90	153	423	73	90	157	426

Table 5.1. Break-up of time expended in different operations during message transfer for different message sizes. Times are in microsecs, and indicate the time when that operation is completed since the send doorbell is rung for the send side, and receive doorbell is rung for the receive side. Receive descriptors are pre-posted before the data actually comes in for PVIA and QoSVIA, and their DMA transfers do not figure in the critical path.

Table 5.1 gives the time when the different send operations complete, relative to the time when the send doorbell is rung. The operations are HDMA Programming (to bring down the send descriptor), Descriptor Down (indicating that the send descriptor is on the NIC), HDMA Programming (to bring down the data portion), Data Down (the data is on the NIC), NSDMA Done (the data has been sent out on the wire), HDMA Programming (to move up the status), and Status Done (indicating that the main memory has status completion.) Similarly, the receive side gives the time when the receive operations complete, relative to the deposit of the message on the NIC by

the NRDMA engine.These receive side operations are HDMA Programming (to bring down receive descriptor), Descriptor Down (indicating that the descriptor is on the NIC), HDMA Programming (to send the data up to the host memory), Data Up (the data is now in host memory), HDMA Programming (to move up the status), and Status Done (indicating that the main memory has status completion.) Also shown are the observed 1-way latencies. The Status done times on the send and receive sides are measured on different NIC clocks and do not add up to the 1-way latencies. The breakup is still a useful way of seeing the time spent in different parts of the message pipeline.

It is quite apparent that the restructuring of the firmware in PVIA and QoSVIA add to the overheads of these operations (state and queue maintenance) compared to SVIA, which is to be expected. However, these overheads are relatively independent of message size (and only dependent on the efficiency of the LCP). As a result, the differences between these implementations become less important as we move to larger message sizes, even with a ping-pong experiment which does not allow any scope for parallelism/pipelining in the NIC operations. Further, the overheads for QoSVIA compared to PVIA, due to time-stamping and priority queue orderings, are not very significant (the priority queue is not really stressed in this case since the ping-pong experiment keeps at most one message in the queue at any time).

To study a more realistic situation where there is scope for parallelism, we run a ping-bulk experiment between two machines (sender pumps in several messages and waits for several messages, with the receiver doing the reverse) using two uni-directional (one for send and another for receive) channels between two machines. It should be noted that the window size (number of messages to send before switching directions)

can have an effect on the LCP performance. Small window sizes would limit the scope of parallelism, and large window sizes may create flow control problems. VIA does not not mandate a flow control mechanism and hence it is absent from our implementations. By varying the window size, we have found the point that provides the lowest average 1-way latency per message in each experiment, and the corresponding results are shown in Table 5.2 as a function of the message size for the three LCP designs.

Msg. Size	SVIA	PVIA	QoSVIA
4	8	26	32
1K	14	28	34
4K	48	36	46
16K	152	86	

Table 5.2. Average 1-way Message Latency in microsecs using Ping-Bulk experiment

The results in this table confirm our arguments about the necessity for a parallel/pipelined firmware with larger loads on a channel. The overheads of the optimizations are overshadowed by the performance enhancements at higher loads. For instance, both PVIA and QoSVIA outperform SVIA after message sizes of 4K. With small messages (1K or less), the overheads of the optimized LCP are not able to overlap with the diminished operation costs. With larger messages, the benefits of the overlap materialize. Though not explicitly shown here, we can expect similar benefits from PVIA/QoSVIA with multiple channels (even if the load on a channel does not increase), since there is literally no change in the behavior of the firmware with increase in BE channels (the doorbells on different channels are merged by the hardware into a single queue).

5.4.1.2 Performance with CBR Channels (QoS)

In the next experiment, we evaluate the three LCPs using CBR traffic. The experiment is composed as follows. We employ one sender machine that uses three classes of QoS channels (requiring 1 MByte/sec, 2 MBytes/sec and 3 MBytes/sec respectively) to send out messages to three receiver machines (each receiver receives one class of channels). The load is increased by increasing the number of channels in each class as 1,2,3,4 and 5 channels, giving overall injected loads of 6, 12, 18, 24 and 30 MBytes/sec respectively. Carefully designed workload generators were used to inject messages in the different channels at the specified rates, and the time difference between successive receipt of messages at the receiver is used to measure jitter duration and θ. The resulting jitter fraction (θ) with the three LCPs is shown in Figure 5.2. We can observe that SVIA has slightly higher θ (around 10% in many cases) than the other two schemes. Between PVIA and QoSVIA, there is very marginal differences (the latter is a little better though the difference is hardly noticeable in the graph). The differences between PVIA and QoSVIA are a little more noticeable when one observes the percentage of packets (messages) that incur jitters in Figure 5.3 for the same experiment (results are shown for each of the classes). QoSVIA has fewer packets incurring jitters, especially at higher loads suggesting that it is more conducive to meeting the QoS requirements of channels.

We would like to point out that this experiment has not really been able to stress the system enough to bring out differences more prominently. As was mentioned, the VIA implementations (all three, including the one in the public domain [19] that is used

here), do not have flow control capabilities. Consequently, we are not able to pump in messages at a faster rate (as this experiment suggests, the differences are likely to be felt at higher loads, or else just the raw bandwidth availability is enough to meet the QoS demands by any implementation). This is also the reason why the absolute values of θ in Figure 5.2 are very low. Even at these relatively small loads, some differences between the LCP implementations can be felt. Higher level messaging layers built on top of these LCPs may, on the other hand, be able to subject the system to higher loads to experience the differences. The differences between the schemes at higher loads are evaluated next using a simulator.

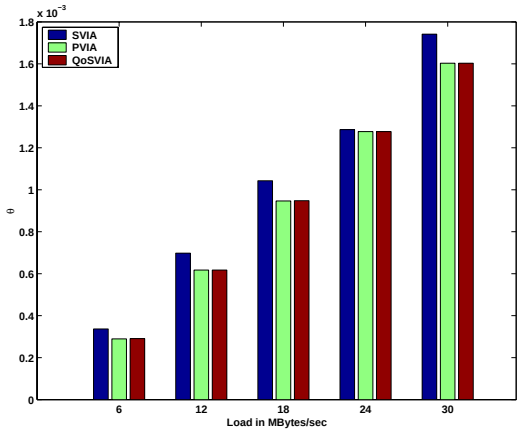
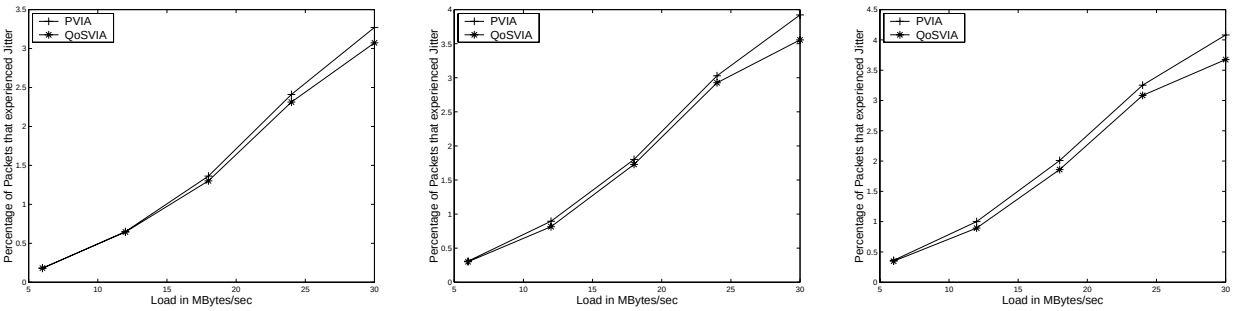


Fig. 5.2. Experimental Results Using 1 sender and 3 receivers with three classes (1, 2 and 3 MBytes/s) of QoS channels, with each receiver receiving one class. The θ shown is for the 2 MBytes/s class.



(a) 1 MBytes/s Channels (b) 2 MBytes/s Channels (c) 3 MBytes/s Channels

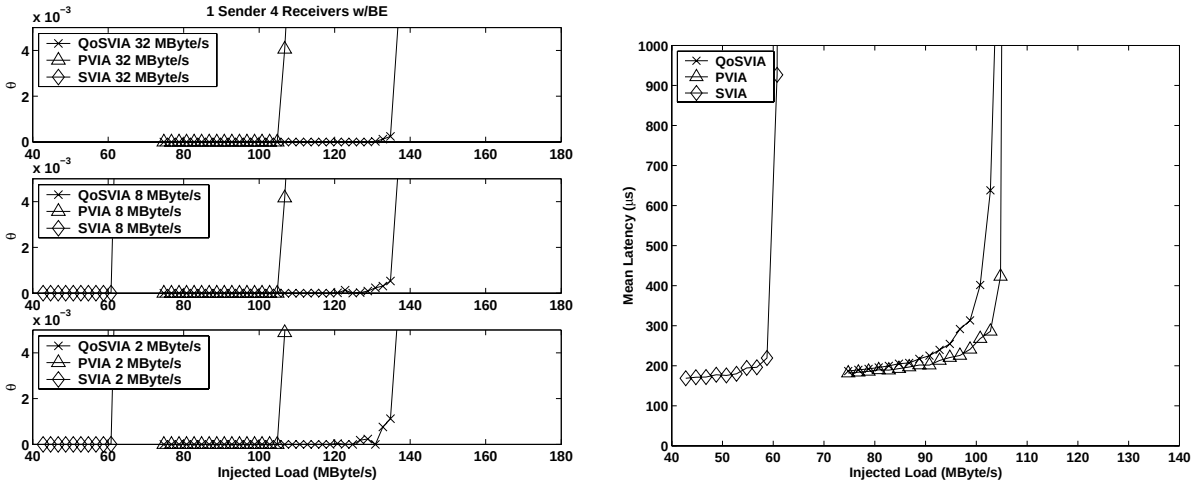
Fig. 5.3. For the same experiment as Figure 5.2, this graph shows the percentage of packets experiencing jitters for PVIA and QoSVIA.

5.4.2 Simulation Results

With the experimental set up discussed above, we are somewhat limited by how much we can stress the system, and by the accuracy of the workload generator in sending/receiving packets at the exact times (due to the vagaries of the system). This is where simulation comes in useful, and we have developed a detailed simulator modeling the different hardware components and the three firmware designs, using the breakup of time spent in the different stages from the measurement on the platform given earlier. The simulation results not only augment the experimental observations, but also serve to explore alternate hardware designs/enhancements (such as hardware queuing for the NSDMA) which is not possible on the current experimental platform. It should be noted that the simulator uses infinite sized buffers/queues, and thus gives an indication of what is possible with each scheme, since our goal is to see how the schemes compare at high loads.

5.4.2.1 Results for Higher Loads

The first set of simulation results shown in Figure 5.4 (a) shows the observed θ for three classes of QoS CBR channels (2, 8 and 32 MBytes/sec) on a 1-sender 4-receiver system as a function of the injected load for the three firmware designs. The injected load is varied by increasing the number of QoS channels, with the bandwidth on any channel remaining the same. In addition to QoS channels, there is also one BE channel that injects messages with a mean rate of 33 MBytes/sec. All QoS channels obey the negotiated bandwidth allocation, and inject messages (all of size 4 KB) as per this rate.



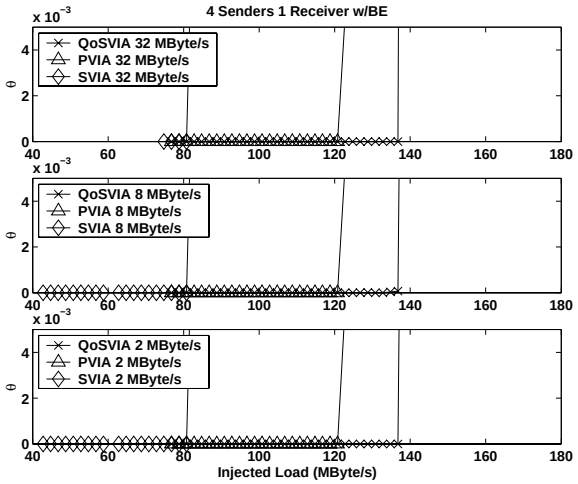
(a) θ for QoS Channels (b) Latency for BE Channel

Fig. 5.4. Simulation results for a 1-sender 4-receivers configuration with 3 QoS classes (2, 8 and 32 MBytes/s) and 1 BE channel (33 MBytes/s). The load on the BE channel, and on any QoS channel remains constant. The total injected load is varied by increasing the number of QoS channels. *All the graphs are plotted as a function of total injected load and not as a function of load on that channel class.* The line for 32 MBytes/s channel with SVIA is not shown because the system has already saturated at these loads.

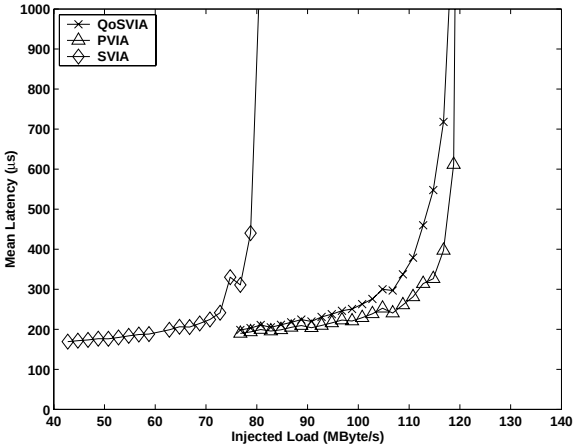
The first point to note is that the simulation results confirm the earlier experimental observation that at lower loads (less than 60 MBytes/sec), there is little difference between the schemes (the scale of the θ graph in the experimental results is much more amplified that what is shown here). However, at higher loads (after 62 MBytes/sec), we find that SVIA is not able to meet the demands of the QoS channels. On the other hand, PVIA is able to handle their demands until the load exceeds 100 MBytes/sec, and the injected load can go as high as 130 MByte/sec before QoSVIA sees any meaningful jitter. These results confirm the importance of a pipelined firmware such as PVIA for higher bandwidth, together with a QoS conscious service discipline. QoSVIA is able to reduce the jitter on each of the traffic classes compared to SVIA or PVIA.

Figure 5.4 (b) shows the latency of messages on the BE channel as a function of the total system load for the same experiment. It should be noted that the load in the BE channel itself remains the same (only the number of QoS channels is increased). This is one way of examining how the load on one class affects the performance of the other. We find that SVIA performance is much worse than the other two, saturating when the load exceeds 60 MBytes/sec. Between PVIA and QoSVIA, we find the former giving slightly lower latency than the latter. This is because PVIA does not differentiate between the traffic classes. On the other hand, QoSVIA gives higher priority to QoS channels, thereby penalizing BE performance a little. Still, the differences between QoSVIA and PVIA for BE traffic are not that significant, and the benefits of QoSVIA for QoS classes can offset this slight deficiency.

We have conducted similar experiments with other experimental configurations. Figures 5.5 and 5.6 show similar results with 4-senders 1-receiver configuration (reverse

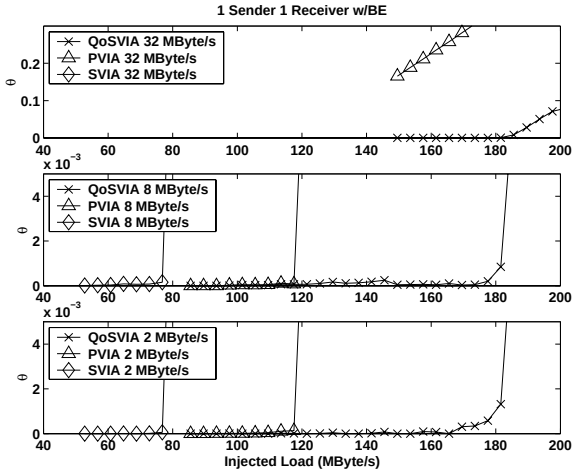


(a) θ for QoS Channels

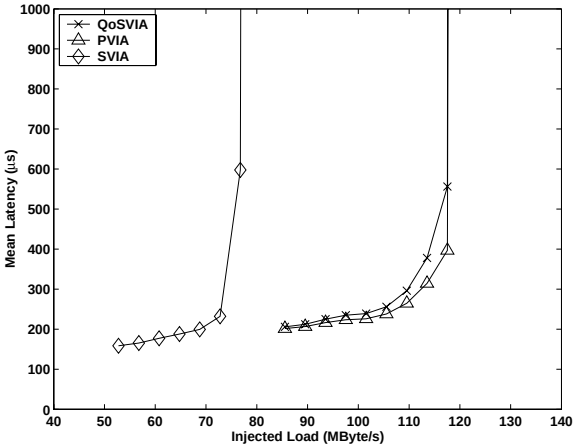


(b) Latency for BE Channel

Fig. 5.5. Simulation results for a 4-senders 1-receiver configuration with 3 QoS classes (2, 8 and 32 MBytes/s) and 1 BE channel (33 MBytes/s).



(a) θ for QoS Channels



(b) Latency for BE Channel

Fig. 5.6. Simulation results for 2 machines sending messages back and forth on 3 QoS channel classes (2, 8 and 32 MBytes/s) and 1 BE channel (33 MBytes/s) SVIA for the 32 MBytes/s channels has already saturated at these loads.

of previous experiment) and 1-sender 1-receiver (each node has a certain number of send channels and receive channels) respectively. The former stresses the receiver more (compared to the earlier experiment) and the latter examines a more balanced system. These results again confirm our earlier observations about the relative performance of SVIA, PVIA and QoSVIA.

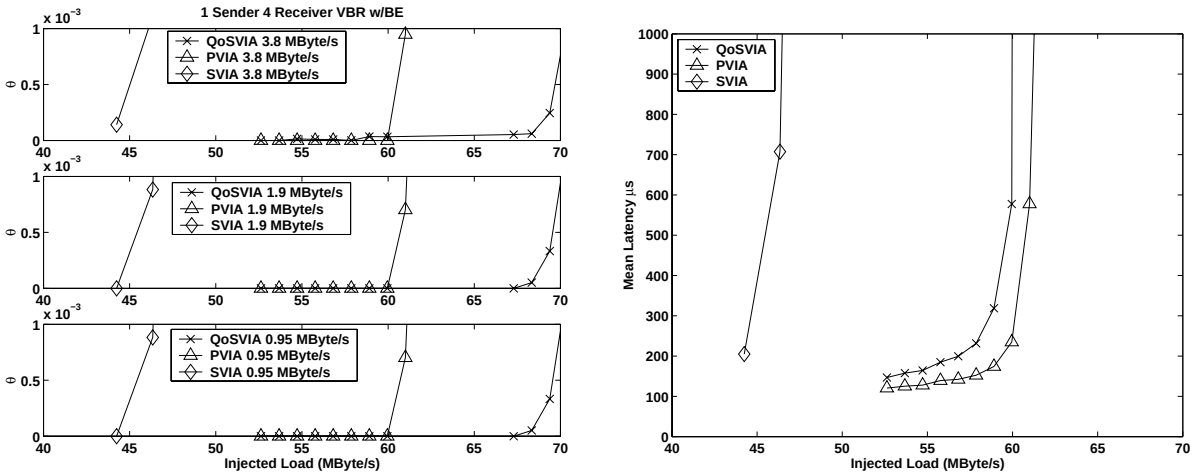
5.4.2.2 VBR QoS Channels

In addition to the experiments using CBR channels, we have also considered VBR traffic in our experiments. Specifically, we have used the MPEG-1 trace files from [77] to generate the workload. This experiment uses 1 sender and 4 receivers, with three classes of QoS channels (0.95, 1.9 and 3.8 MBytes/sec) and one BE channel (16.7 MBytes/s). Within each channel, we inject 20, 40 and 80 MPEG-1 streams (adhering to the specified data rates), and the jitter fraction for QoS channels and latency for BE channel are shown in Figure 5.7.

As with CBR traffic, we find that PVIA and QoSVIA, are much better than SVIA, with QoSVIA able to sustain the highest load before any meaningful jitter.

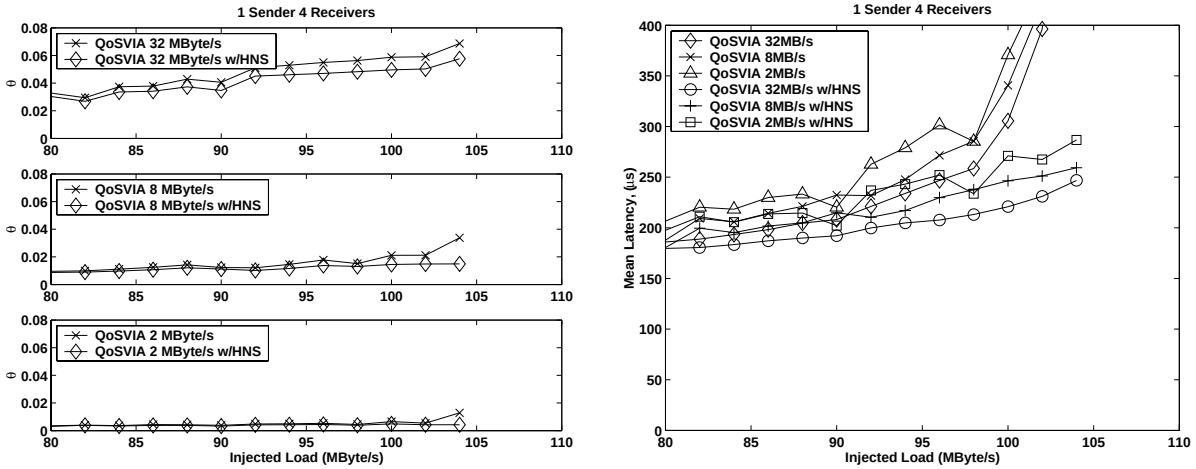
5.4.2.3 Hardware NSDMA Queue

An interesting observation that was made in an earlier study [69] was the benefit of hardware queuing for the network DMAs as well (these are currently software managed). This can decrease the cost of state maintenance in the PVIA and QoSVIA designs. With a simulator, we can easily incorporate this feature and examine the resulting benefits. Figure 5.8 shows the resulting performance benefits of hardware queuing support with a NSDMA (denoted "w/HNS" in the graphs) for QoSVIA in the 1-sender



(a) θ for QoS channels (b) Latency for BE channel

Fig. 5.7. Simulation results with MPEG-1 traces for the 1 sender 4 receivers configuration with 3 QoS classes (0.95, 1.9 and 3.8 MBytes/s) and 1 BE channel (16.7 MBytes/s)



(a) θ for QoS channels (b) Latency for QoS channels

Fig. 5.8. Simulation results with Hardware NSDMA for the 1 sender 4 receivers configuration with 3 QoS classes (2, 8 and 32 MBytes/s)

4-receiver experiment having 3 QoS channel classes (no BE in this case). We find that the hardware queuing support not only improves the latency of messages (Graph (b)) as observed in [69], but also reduces the jitter fraction. Hence, it would be valuable to incorporate hardware queuing for network DMAs from both the latency/bandwidth and QoS viewpoints.

5.5 Summary

With clusters taking on demanding applications with different QoS requirements, there is a critical need for the communication subsystem to handle these diverse, and perhaps coexisting, applications in such a way that they do not interfere with each other. This ambitious goal requires a reworking of several levels of the system architecture including the physical network, the network interface, the communication software and even the CPU scheduler on the workstation. A complete end-to-end solution is well-beyond the scope of this thesis, but this chapter has taken an important step by examining the issues in providing QoS capabilities for the network interface (NIC) and the low-level communication software (specifically in the context of VIA).

This work presents the design and implementation of firmware for the Myrinet NIC to provide both high bandwidth and QoS-aware communication. The PVIA LCP presented uses an event-driven approach to improving the degree of parallelism of activities on the NIC. Another approach outlined is QoSVIA, which extends PVIA by adding a rate-based service discipline to handle bandwidth sensitive channels.

The three firmware designs presented are compared using both measurements and simulation. Results from these indicate that PVIA and QoSVIA are significantly better

than SVIA in providing QoS support. Further, the simulation results show QoSVIA to be better than PVIA in providing QoS differentiation at higher loads.

Chapter 6

Communication-driven scheduling

Until now, this thesis has primarily focussed on developing and optimizing user-level communication mechanisms. However, optimizing communication alone may not necessarily translate to improved performance since the scheduling strategy could nullify any savings. For instance, a currently scheduled process on one node would experience a long wait for a message from a process not currently scheduled on another node regardless of the low latency for messages. Scheduling and communication are thus closely intertwined, and should be studied together. Scheduling of processes onto processors on a parallel machine has always been an important and challenging area of research. Its importance stems from the impact of the scheduling discipline on the throughput and response times of the system. The research is challenging because of the numerous factors involved in implementing a scheduler. Some of these influencing factors are the parallel workload, presence of any sequential and/or interactive jobs, native operating system, node hardware, network interface, network, and communication software. Previous studies on parallel schedulers have focussed their attention on closely coupled parallel systems. Communication and synchronization costs on such machines are relatively low, making complex scheduling schemes feasible. However, many of these scheduling strategies are not very practical for a loosely-coupled cluster environment.

Scheduling is usually done in two steps. The first step is assigning a process to a processor, and the second is scheduling the processes assigned to a processor. There is a considerable body of literature [79, 61, 73, 102, 91, 97, 34, 22] related to the first step on closely coupled multiprocessor systems. Some of these studies [91, 102, 61] exploit the relatively low communication and synchronization overheads of these machines, particularly those with shared memory capabilities, to dynamically move processes across processors based on CPU utilization. Other studies [34, 22] assume process migration to be expensive, and propose static processor allocation strategies in which the set of processors is spatially partitioned. On a cluster, communication and synchronization costs are relatively high. Further, processes, as implemented by the native operating system at each node, are heavyweight and expensive to migrate. Hence, process migration is cost-effective only for relatively long running jobs [1]. In this study, we assume that processes are statically assigned to the nodes of a cluster and do not migrate during execution.

The second scheduling step, which is perhaps more important for a cluster, is the scheduling of assigned processes at each cluster node. The choices here range from strategies based purely on local knowledge at a node to those using global knowledge across nodes for making more intelligent decisions. Local scheduling, which does not require any global knowledge, is relatively simple to implement. In fact, one could leave the processes to be scheduled by the native operating system of the node. The drawback is that the lack of global knowledge can result in lower CPU utilization and higher communication or context switching overheads. At the other end of the spectrum is *coscheduling* (also called gang scheduling) [71, 41, 42, 78], which schedules processes of a

job simultaneously across all processors, giving each job the impression that it is running on a dedicated system. While coscheduling has been shown to boost the performance of fine-grained parallel applications, an implementation on a cluster can become expensive. Further, from performance and reliability perspectives, it does not scale well with the number of nodes in the system.

Another class of scheduling strategies use communication behavior (which are local events at each node) to guide the scheduling. Studies like [84, 3, 35, 4] attempt to dynamically coschedule communicating processes to improve job performance. Only two of these dynamic approaches [5, 83] have been proposed, implemented and evaluated on an actual cluster. These two strategies, called implicit coscheduling [5] and dynamic coscheduling (DCS) [83, 18], use locally available information to estimate what is scheduled on the other nodes, without requiring any explicit messages for obtaining this information. Two actions, namely, *waiting for a message* and *receipt of a message*, form the basis for these schemes. Implicit coscheduling is based on the heuristic that a process waiting for a message should receive it in a reasonable time (as determined by the message latency and other factors) if the sender is also scheduled currently. Dynamic coscheduling, on the other hand, uses message arrival to indicate that a process of the same job (the sender) is scheduled on the remote node, and schedules the receiver accordingly. The former [5] has been implemented and evaluated on Active Messages [94], which offers a closer coupling between the sender and receiver processes than MPI on Fast Messages [72], which has been used in evaluating DCS [83]. Further, the version of Fast Messages used in [83] can handle only one parallel application per node, and as a result, the evaluation is rather limited.

These studies raise some important questions for cluster schedulers. First, what is the design spectrum for developing communication-based dynamic scheduling mechanisms on a cluster? In particular, what are the pros and cons of scheduling using message wait and message arrival information? Second, how can these techniques be implemented within the context of the current user-level messaging platforms (where the OS scheduler is unaware of communication events) ? Third, how do these schemes compare with each other, and how much do they deviate from ideal behavior in terms of throughput and response time? Finally, in addition to throughput and response times, how do the schemes compare in terms of fairness? Answers to these questions require an experimental testbed to design and implement various scheduling strategies and a detailed evaluation to understand the intricate interaction between several factors. To our knowledge, no previous study has extensively evaluated these issues on a unified framework, and these issues are explored in detail in the rest of this chapter.

6.1 Scheduling Strategies

Logically, there are two components to the interaction between a scheduler and the communication mechanism. The first is related to how the process waits for a message. This can involve: (a) just spinning (busy wait); (b) blocking after spinning for a while; or (c) yielding to some other process after spinning for a while. The second component is related to what happens when a message arrives and is transferred to application-level buffers. Here again, there are three possibilities: (a) do no explicit rescheduling; (b) interrupt the host and take remedial steps to explicitly schedule the receiver process; and (c) periodically examine message queues and take steps as in (b). These two components

can be combined to give a 3×3 design space of scheduling strategies as shown in Table 6.1.

What do you do on message arrival?	How do you wait for a message?		
	Busy Wait	Spin Block	Spin Yield
No Explicit Reschedule	Local	SB	SY
Interrupt & Reschedule	DCS	DCS-SB	DCS-SY
Periodically Reschedule	PB,PBT	PB-SB	PB-SY

Table 6.1. Design space of scheduling strategies

We have implemented a protected user-level messaging layer that provides the complete MPI [65] functionality on a cluster of UltraSPARC workstations running Solaris 2.5 and connected by Myrinet. The MPI implementation is based on the MPICH distribution [50]. Details of its implementation and performance are given in [9]. We refer to this platform as the baseline, which serves as a uniform framework for implementing and evaluating different scheduling strategies. Essentially there are three software components that are important to understand the rest of this discussion. The first is the LANai control program executing on the LANai processor of the Myrinet interface. This program performs the data transfer between the host memory and the network. Though this can raise an interrupt for the host processor, this feature is not used in the baseline implementation since message transfer is implemented by polling at the user level. The second is the set of user-level libraries, which includes U-Net [93] from Cornell, together with umlib and MPI Unet, which we have developed to provide an efficient MPI interface. The implementation incorporates several optimizations to eliminate multiple levels

of copying. These routines manage the send and receive queues mapped in directly to the user address space. There is no kernel invocation for data transfers. The third component is a kernel device driver, which in the baseline implementation is used only at the initialization stage to set up endpoints. The device driver also offers the potential for performing some actions in kernel mode (via an ioctl call), used in implementing certain scheduling schemes. A schematic showing the different components in the baseline implementation and potential additions for implementing the different scheduling strategies is shown in Figure 6.1. The baseline platform delivers one way latency of 32μs and a peak bandwidth of 26.2 MBytes/sec.

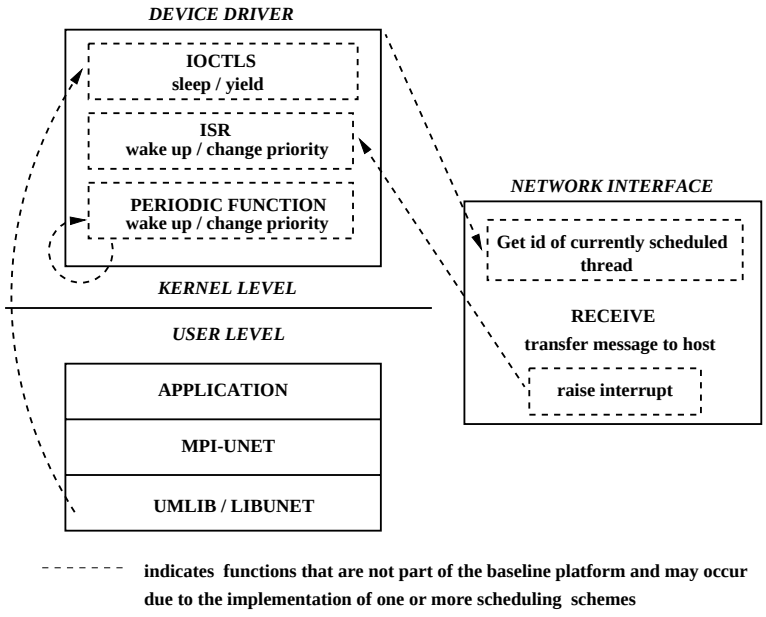


Fig. 6.1. Software components used in the implementations

In the following discussion, we present a brief description of each scheduling strategy considered in this study, its implementation and the potential pros and cons. The reader should note that the implementation of a strategy may require the modification of one or more of the following: the LANai Control Program (LCP), the device driver that interacts with the network interface and allows certain actions to be performed in kernel mode, and the umlib and MPI layer. Details of the modifications to each of these components are given below for each scheduling strategy. The term, parallel processes, is used to refer to the processes of the parallel application on the individual workstations while the term, serial processes, is used to refer to other interactive, background and sequential processes, which may be running on the workstation. We run single threaded applications, so the terms, thread and process, are used interchangeably. Further, each process of a parallel job uses only one endpoint for communication. As a result, there is a one to one mapping between a process and an endpoint on a given workstation.

6.1.1 Local Scheduling

This scheme has been considered as a baseline to show the need for a scheduling strategy based on global information. The parallel processes are treated the same as the serial ones, and it is left to the native Solaris scheduler to schedule all the processes at a workstation. There is no effort made to coordinate activities across workstations and each local Solaris scheduler makes independent scheduling decisions. A brief description of the Solaris scheduling mechanism is given below.

Processes inherit one of 60 priority levels from their parent when they start. Each priority level (from 0 to 59 with a higher number denoting a higher priority) has a queue

of runnable processes at that level. The process at the head of the highest priority queue is executed first. Higher priority levels get smaller time slices than lower priority levels, with the default ranging from 20 ms for level 59 to 200 ms for level 0. Process priority is often boosted when they return to the runnable state from the blocked state (completion of I/O, signal on a semaphore etc.) The amount of the boost is determined by values specified in `ts_disptbl` (a table that can be modified by privileged users) as is the time slice for a particular priority level. The scheduler, which runs every millisecond, ensures that lower priority processes are preempted if a higher priority process becomes runnable. This design strives to strike a balance between compute and I/O bound jobs, with I/O bound jobs typically executing at higher priority levels to initiate the I/O operation as early as possible. Starvation is prevented by boosting the priority of a process if it has not been able to complete its time slice even after a certain time limit. In addition, the priorities of all processes are raised to level 59 every second.

From the implementation viewpoint, this scheme is straightforward since it does not require any modifications/additions to the baseline. Leaving it to the Solaris scheduler to handle all processes (both parallel and serial) ensures fair CPU allocation. However, the absence of coordination between machines can hurt CPU efficiency. Going back to the earlier example, this scheme can result in a receiver process being scheduled and waiting for a message from a process that is not scheduled on another workstation. Since the receive operation is implemented by polling at the user level, valuable CPU cycles can be wasted by redundant polling in this scheme.

The following schemes attempt to ameliorate this problem by using local communication events to estimate what is happening at the other workstations, and using this

information to provide hints to the scheduler. The aim is to make the communicating parallel processes of a single job run at the same time to approximate the behavior of gang scheduling.

6.1.2 Spin Block (SB)

Versions of this mechanism have been considered by others in the context of implicit coscheduling [38, 5] and dynamic coscheduling [83]. In this scheme, a process spins on a message receive for a fixed amount of time before blocking itself. The fixed time for which it spins, henceforth referred to as *spin time*, is carefully chosen to optimize performance. The rationale here is that if the message arrives in a reasonable amount of time (spin time), the sender process is also currently scheduled and the receiver should hold on to the CPU to increase the likelihood of executing in the near future when the sender process is also executing. Otherwise, it should block so that CPU cycles are not wasted. While a theoretical analysis to calculate the optimal spin time can be done in a few situations (as in [5]), such an analysis can become exceedingly complex for a real application running on a generic message passing layer such as MPI. We have resorted to an empirical approach to quantify the optimal spin time for a given application (similar to [83]). By varying the spin time for several workloads, we have found the ideal spin time for the workloads considered here to be around 250-300 microseconds.

The reader should note that the SB mechanism described here is different from a similar mechanism implemented with implicit coscheduling in [5] for the following reasons. First, in [5], in addition to the fixed spin time (which they call baseline spin), the

receiver spins for an additional *pairwise* spin time to synchronize each pair of communicating processes. The pairwise spin time is not considered in this study. Second, the implementation in [5] is based on Split-C/Active Messages [94], which has a more tight coupling (than MPI considered here) with a reply message associated for most messages sent. As a result, the spin time before the reply is received can be relatively shorter (and is a better estimate of what is scheduled at the remote node) than the corresponding spin in a MPI messaging layer, where the receiver mostly relies on the application-level send, which can get arbitrarily delayed because of work-imbalance and other skews. Hence, we refer to our implementation as Spin-Block (SB) and not implicit coscheduling, and the resulting performance for SB can thus be different from the results presented in [5].

In our implementation of Spin Block, the polling loop in the user-level library receive call is modified to run until *spin time* elapses. If a message still does not arrive, an ioctl call is made to the kernel device driver, which makes the process block on a semaphore. The ioctl routine also registers a wakeup call for the corresponding endpoint with the network interface. When a message arrives for that endpoint, the LANai issues an interrupt, and the interrupt service routine signals the corresponding semaphore. The Solaris signaling mechanism moves the blocked process to the head of the runnable queue for a higher priority level most of the time. The woken up process can thus get a priority boost on receipt of a message (though this is done implicitly within Solaris and not explicitly in our implementation). It should be noted that there could be a race condition between the ioctl call registering with the LANai and blocking, and the LANai noticing an incoming message and checking if an interrupt needs to be raised based on its registry. In certain executions, this could lead to a situation where the process is

blocked but an interrupt would never be raised for that process. We have taken care of such race conditions in our implementation.

Spin Block improves performance in two ways. First, by reducing the number of CPU cycles spent in idle spinning, it increases CPU utilization per node. Second, due to the priority boost a process can receive on wakeup, it is more likely to be scheduled soon after getting a message. Since the sender of the message is also likely to be scheduled at that time (one way latencies being much smaller than an average time slice), the probability that a pair of communicating processes execute concurrently for some time, is higher. On the downside, it can increase the overhead for message transfer slightly since extra functionality is added to the LCP and the device driver.

6.1.3 Dynamic Coscheduling (DCS)

Sobalvarro et. al. [83] propose dynamic coscheduling to reduce scheduling skews between workstations. The idea here is to use incoming messages to schedule the processes for which they are intended. The underlying rationale is that the receipt of a message denotes the higher likelihood of the sender process of that application being scheduled at the remote workstation at that time.

Our implementation of DCS is similar to the one discussed in [83]. The library level of the messaging platform does not require any modifications from the baseline. However, the implementation requires additional functionality in the LANai Control Program (LCP) of the network interface. Periodically the LANai has to get the *id* of the thread (kernel variable `cpu[0]->cpu_thread`) currently executing on the host CPU. Since the LANai cannot directly access host memory, this variable has to be DMAed

onto the card each time. The frequency of this operation determines the accuracy of the estimated running process (the more frequent this operation, the more accurate is the estimate by the LANai) as well as the overhead for normal communication (the more frequent this operation, the higher the overhead in the LANai for normal send and receive operations). The frequency has been set at one per millisecond as suggested in [83]. On receipt of a message, the LANai checks whether the intended destination of the message matches the estimated currently running process. If there is a mismatch, an interrupt is raised and the interrupt service routine (ISR) of the device driver in the kernel is executed. The ISR first checks whether there really is a mismatch, (since the LANai estimate could be off by 1 millisecond). If so, it boosts the priority of the destination process to the highest value by placing it at the head of the queue for priority level 59. This ensures that the destination process is scheduled soon after the receipt of a message. DCS uses only a busy-wait for receiving as in the baseline implementation.

DCS can slow down the normal send and receive operations because of the additional work imposed on the LANai. Also, DCS does not handle application level skews (due to work imbalance) between the sender and receiver, and lets processes spin for the remainder of their time slices in such situations. However, since it boosts priorities of receiver processes even when they are spinning, it can potentially coschedule communicating processes more often (and sooner after message arrival) than Spin Block, which boosts priorities only on being woken up (and not during spinning).

6.1.4 Dynamic Coscheduling with Spin Block (DCS-SB)

DCS would let a process spin for the remainder of its time slice if the message it is waiting for does not arrive in a reasonable amount of time. One fix to this problem, as suggested in [83], is to limit the spin time which leads us to the next scheduling policy called DCS-SB. Spinning receivers go to sleep after a fixed spin time has elapsed. Incoming messages would either boost the priority of the destination process (if it has not yet blocked) or would wake up the destination process and boost its priority.

The user-level library (which does the spin) and the ioctl call in the driver (which registers with the LANai for an interrupt and does the block) for DCS-SB are identical to the corresponding operations for Spin Block discussed earlier. The LCP on the LANai is very similar to the corresponding LCP for DCS with one small difference. In DCS, the LANai would raise an interrupt only if it estimated the destination process to be not scheduled. For DCS-SB, in addition to this condition, an interrupt is raised if the destination process has registered itself with the LANai for a wakeup. The ISR on the host also checks for both these conditions, and performs a wakeup or just a priority boost as needed.

DCS-SB incurs the overheads of both Spin Block and DCS, though not always together. Its ability to increase coscheduling beyond what DCS can offer would be more significant at a higher load (there is other useful work to do when one or more processes have blocked).

6.1.5 Periodic Boost (PB and PBT)

The first of the newer schemes that we propose is called Periodic Boost (PB). Going back to either of the DCS schemes or SB, we observe that the solution to approach gang scheduling has been to boost the priority of the process (the destination of a message) on message arrival. However, this boost is done within the interrupt service routine since an interrupt is the only way of detection of message arrival by the kernel in these schemes. When there is a mixture of high communication workloads running at a workstation, these schemes can result in a large number of interrupts, thereby leading to a large overhead (defeating the purpose of user-level messaging). In the PB scheme, we propose that we do not have any interrupts being raised at all. Rather, we can have an entity within the kernel, which periodically examines the endpoints of the parallel processes and boosts their priorities. Though a number of criteria can be used for boosting priorities, we use two which leads us to the two schemes called PB and PBT. In PB, the periodic mechanism checks the endpoints in a round-robin fashion and boosts the first with an unconsumed (henceforth called pending) message; if no one has pending messages, no one is boosted. In PBT (Periodic Boost using Timestamps), the periodic mechanism boosts the process which has the most recently arrived pending message. The frequency with which these actions should be taken needs to be chosen carefully. We have used an experimental approach to find this frequency and have found that invoking the boosting function once every 10 milliseconds gives good performance.

PB and PBT are simpler to implement than the previous three schemes. There is absolutely no change in the user-level messaging libraries or in the LANai Control

Program from the baseline implementation. All that is needed is an additional function in the device driver which gets called periodically (via a timer mechanism) to examine the number and/or the timestamp of pending incoming messages for each endpoint and boost the priority of a process when needed.

The potential benefits of PB are fourfold. First, it allows a more complex heuristic to be used for making scheduling decisions for parallel processes. It is thus possible to come closer to coscheduling than is achieved with DCS or SB, which uses only message arrival information. This is particularly important when there are multiple parallel processes and more than one qualify for a priority boost. Blindly boosting the priority of all of them to a single value (which is what is done explicitly in DCS) or to an unknown value (done implicitly in SB), again leaves the interleaving of their schedules in the hands of the local scheduler. Instead, a more intelligent heuristic may lead to better performance. If the same policy is followed on all the workstations, the execution may better approximate coscheduling. Second, since there is no additional work done by the LANai, the overhead for normal send/receive operations is minimized. Third, it is possible to dynamically control the invocation of the function (in the device driver), which does the priority boost based on the changing nature of the workload. The function could be called less frequently if there are fewer endpoints or if they do not communicate often enough. Finally, since the priority boost function invocation is not really tied to communication events, unlike in DCS or SB, it could base its decision for boosting priorities on statistics totally unrelated to the communication. The function could even be used for implementing fairness/unfairness in scheduling. However, we do not fully

explore the third and fourth issues in this study, and base the priority boost purely on pending message information.

6.1.6 Periodic Boost with Spin Block (PB-SB)

An immediate extension of PB (or PBT) is to augment it with Spin Block. This would limit the redundant spin time (in the absence of a message) to a much lower value than the periodic interval of boosting priorities in PB. For instance, our experimental studies show that the ideal spin time is of the order of 250-300 microseconds while PB does best with one boost in around 10 milliseconds.

From the implementation viewpoint, the user-level messaging libraries, the LANai control program and the interrupt service routine for this scheme are identical to the corresponding routines for Spin Block. There is a required spin time following which there is an ioctl call, which registers with the LANai and then blocks on a semaphore. The LANai raises an interrupt if there is an incoming message and the corresponding process has registered itself. The ISR wakes up the corresponding process for which the message is intended. The only modification to the implementation is the additional function in the device driver, which is periodically called to check the message queues of each process and to boost one of them if needed. This function is similar to the one used in PB with a slight modification. It preferentially wakes up sleeping processes with pending messages. If there are none, it does exactly the same as what is done in normal PB.

On the downside, PB-SB has the same overheads as SB in the normal send/receive operations and the interrupt costs, and the same overheads as PB because of the periodic

execution of the boosting function. PB-SB can be expected to do better than PB when the load is high and there is other useful work to do when one or more processes have blocked.

6.1.7 Spin Yield (SY)

In SB, the process blocks after spinning. This has two consequences. First, an interrupt is required to wake the process on message arrival (which is an overhead). Second, the block action only relinquishes the CPU and there is no hint given to the underlying Solaris scheduler as to what should be scheduled next. We attempt to fix these two problems using the next scheduling strategy, which is called Spin Yield (SY). This has been mentioned to a limited extent in [83] without any implementation details or an evaluation. In this strategy, after spinning for the required spin time, the process does not block. Instead, it lowers its priority, boosts the priority of another process (based on the pending messages of the other processes at that workstation), and continues spinning. This avoids an interrupt (since the process keeps spinning albeit at a lower priority), and gives hints to the Solaris scheduler as to what should be scheduled next.

This scheme does not require an ISR and the LANai Control Program does not require any modification from what has been presented in the previous discussion. The user level communication libraries are similar to those for SB, where the receive call spins for a certain time followed by an ioctl call to the kernel driver. The ioctl call lowers the priority of the spinning process (to a level that is one below the lowest priority of a parallel process at that node), examines the incoming message queues of the other processes, and boosts (to level 59) the priority of one of these with a pending message. Care is taken

so that the priority is not lowered twice for the same expected message. After returning from the ioctl call, the user-level receive call continues spinning for the message.

Comparing SY with SB, SY would do worse when the workload has a high number of compute bound jobs since SY may not find any other process to boost and would unnecessarily consume extra CPU cycles spinning. However, SY can sometimes outperform SB because of the two reasons mentioned earlier.

6.1.8 Dynamic Coscheduling with Spin Yield (DCS-SY)

The reader should note that SY is an alternative to SB. Just as we had DCS-SB and PB-SB combinations, we could also have DCS-SY and PB-SY combinations.

The implementation of DCS-SY takes the implementation of DCS and adds the functionality of SY. After the process performing a receive operation spins for a fixed interval, it yields i.e. it lowers its priority and raises the priority of another process which has pending messages. The operations for the LANai and the driver remain the same as in DCS.

As for DCS-SY, the anticipated savings over DCS-SB are due to a potential avoidance of a context switch when the fixed spin time elapses, and the provision for scheduling a process with pending messages next than leaving it to the native operating system scheduler.

6.1.9 Periodic Boost with Spin Yield (PB-SY)

The implementation of PB-SY takes the implementation of SY and adds the extra function (which gets called periodically) in the driver to implement PB.

From the performance viewpoint, the relative benefits of PB-SB and PB-SY directly translate from the relative benefits of SB and SY identified earlier.

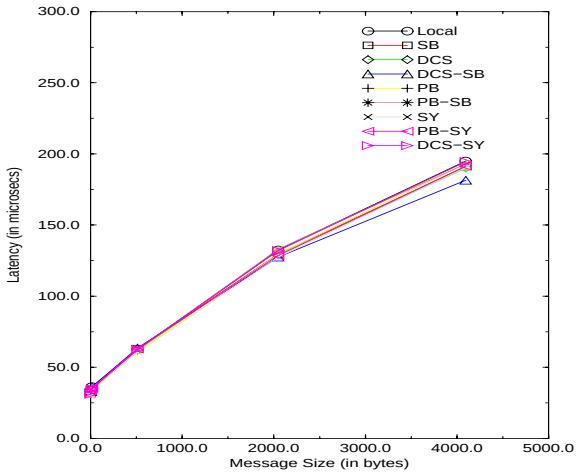


Fig. 6.2. Effect of scheduling policy on one-way latency

The implementation of the novel scheduling strategies presented in this section can, however, have a detrimental effect on the normal data transfer mechanism. To investigate if there is indeed a slowdown in message latencies, we run a one-way latency benchmark with each scheduling scheme in place, and plot the results in Figure 6.2. Contrary to expectations, we find that the one-way MPI latency is not significantly impacted by the implementation of any of these scheduling strategies. In fact, it sometimes even outperforms the baseline implementation because of potential coscheduling possibilities with the different strategies.

6.2 Performance Results

We conduct a comparison of the schemes presented in the previous section on a uniform platform using workloads that are a mixture of jobs with varying communication intensities. Our workloads are mixtures of parallel jobs and do not consider any explicit sequential/interactive ones. However, there are always some background/daemon processes executing on a workstation (even on an unloaded system) which can potentially perturb the execution of the parallel jobs. A description of the different workloads considered in this study and the experimental setup is presented first. The performance results comparing the scheduling strategies and the discussion of the results is presented next. It should be noted that since we provide the complete and exact MPI, off-the-shelf applications can be readily used for our evaluations.

6.2.1 Experimental Setup and Workloads

Our experimental platform is a network of eight Sun Ultra-1 Enterprise servers running an unmodified Solaris 2.5.1 operating system. The workstations have 167 MHz UltraSPARC processors with 64 MB of main memory and a 32 bit SBUS interface operating at 25 MHz. The eight workstations are connected by Myrinet through an 8-port switch with the interface cards having a 37.5 MHz LANai processor and 1 MB of SRAM.

The first application that we consider is LIFE, an example program that comes with the MPICH distribution, which is illustrative of near-neighbor communication in matrix computations. It simulates the game of life on a two-dimensional matrix of cells which is partitioned amongst the processors. Each processor communicates with its four

nearest neighbors along the boundary of the sub-matrix assigned to it. More importantly, from the scheduling perspective, the application is of the bulk synchronous type with distinct communication and computation phases and a barrier separating the iterations. LIFE is particularly suitable for our study because by varying two parameters, namely problem size (matrix size) and the number of iterations, it is possible to control the granularity of communication, while keeping the total execution time roughly the same. A large matrix size with small iterations results in a coarse grain application while a small matrix size with more iterations has fine grain communication characteristics.

Three other applications that we consider (MG, LU and EP) are from the NAS benchmark suite. MG is a simple multigrid solver that solves constant coefficient differential equations on a cubical grid. It is the most communication intensive of the three and spends 26% of the execution time on communication. LU is a matrix decomposition application that uses a large number of small messages. Of the three, it falls in the middle in terms of communication intensity with 16% of the execution time on communication. EP is an embarrassingly parallel application that is typical of many Monte-Carlo simulations. There is very little communication in this application (<1%) in the form of some global sums towards the end of a large computation. For the purposes of this study, it only serves as a competitor for processor cycles, which can skew the scheduling of other communicating parallel applications.

Using these four applications, we first construct nine different workloads (shown in Table 6.2), each containing four applications, which capture interesting mixes of the applications. The middle column shows the four chosen applications for the workload and the percentage of communication (of the total execution time) in that application.

	Applns. in Workload	Comm. Intensity
Workload 1	(LIFE (3.5%), LIFE (3.5%), LIFE (3.5%), LIFE (3.5%))	(lo,lo,lo,lo)
Workload 2	(LIFE (3.5%), LIFE (3.5%), LIFE (3.5%), LIFE (12%))	(lo,lo,lo,hi)
Workload 3	(LIFE (3.5%), LIFE (3.5%), LIFE (12%), LIFE (12%))	(lo,lo,hi,hi)
Workload 4	(LIFE (3.5%), LIFE (12%), LIFE (12%), LIFE (12%))	(lo,hi,hi,hi)
Workload 5	(LIFE (12%), LIFE (12%), LIFE (12%), LIFE (12%))	(hi,hi,hi,hi)
Workload 6	(EP (<1%), EP (<1%), EP (<1%), EP (<1%))	(lo,lo,lo,lo)
Workload 7	(MG (26%), LU (16%), LIFE (12%), EP (<1%))	(hi,me,me,lo)
Workload 8	(MG (26%), MG (26%), EP (<1%), EP (<1%))	(hi,hi,lo,lo)
Workload 9	(MG (26%), MG (26%), MG (26%), MG (26%))	(hi,hi,hi,hi)

Table 6.2. Four Process Mixed Workloads (% of time spent in communication is given next to each application)

The third column gives a quick overview of the mix of communication intensities of the applications in the workload (*lo* indicates relatively low communication, *hi* indicates relatively high communication, and *me* is in between). The first five workloads are constructed directly from the LIFE application which provides tunable parameters to vary the communication intensity. They range from all four processes on a workstation having low communication, through a mix of high and low communication intensities, to a fully communication heavy workload. The next four workloads choose a mix of the four applications, and again span from low to high communication intensities. To study the performance of the scheduling strategy with a different number of processes running at a workstation, we have also considered mixed 2 process workloads constructed from LIFE. These workloads are given in Table 6.3.

The problem size for the different applications are adjusted so that each takes approximately the same time (25 seconds) to complete if it were run alone, and they are reasonably small so that all of them can simultaneously fit in primary memory (to

minimize paging effects). For instance, executing the four job workloads on an ideal gang scheduled environment (without any overheads for scheduling) would result in a total completion time of 100 (4 * 25) seconds for workloads 1 through 9. Executing them together, however, increases the completion time of each instance by an amount that is dependent on the chosen mix and the scheduling scheme.

There are several criteria – such as throughput/utilization, average response/turn-around time, variance in response times, fairness, and degree of coscheduling – that can be used to qualify or quantify the performance of a scheduling scheme. While one could argue that the degree of coscheduling should be used to compare the scheduling schemes outlined here (because they try to approximate the behavior of coscheduling), the bottom line from the system designer's perspective is to maximize the throughput/utilization of the system while maintaining fairness (an equal/fair allocation of the CPUs to the jobs during execution). Similarly, the user is interested in minimizing the average response/turn-around time and its variance. Coscheduling is one way of meeting the system designer and user goals, but is not necessarily the only solution. We examine performance from the perspective of the system designer and user.

In the first set of results, the metric we use is the time taken for the last process to complete since the first process started executing (which we call the *completion time*). The lower this time the more effective the scheduling scheme. Of course, an ideal coscheduling implementation would give the lowest completion time in most cases. Any additional time taken beyond this lowest completion time is considered a overhead. Therefore, if one of the above 4 process workloads were to take 140 seconds to complete on some scheduling strategy, it is said to have a slowdown of 40% over coscheduling.

Hence, slowdown for a scheme (compared to ideal coscheduling) can be directly computed from the completion times given here. This time is also directly related to the system throughput (completion time divided by the number of jobs). In the next set of results, we give the completion times of the individual jobs to show the *variance in the turn-around times*. We also present figures monitoring the CPU utilization by each process during the course of an execution to discuss *fairness* issues.

	Applns. in Workload	Comm. Intensity
Workload 10	(LIFE (3.5%), LIFE (3.5%))	(lo,lo)
Workload 11	(LIFE (12%), LIFE (3.5%))	(hi,lo)
Workload 12	(LIFE (12%), LIFE (12%))	(hi,hi)

Table 6.3. Two Process Mixed Workloads

6.2.2 Comparison of Scheduling Schemes

	Workload				
	1	2	3	4	5
LOCAL	180	208	674	2524	3997
SB	124	153	773	1814	2849
DCS	162	192	350	533	764
DCS-SB	133	173	321	463	700
PB	130	138	152	226	451
PBT	152	190	252	295	284
PB-SB	130	158	655	1685	2660
SY	157	185	985	2320	3046
DCS-SY	166	205	347	527	717
PB-SY	141	158	287	459	733

Table 6.4. Completion Time in Seconds (Workloads 1 to 5)

Table 6.4 shows the performance of the first five workloads using different scheduling strategies. Considering the schemes individually, the slowdown for Local even with Workload 1 is 80% (compared to coscheduling). The slowdown increases steeply as the workload becomes more communication intensive because of the well known problem of Local (lack of global knowledge in making scheduling decisions). Local's performance is not significant other than as a baseline to show the need for a more sophisticated scheduling policy that bases its decisions on what may be scheduled at other nodes.

SB, DCS and DCS-SB show a less steep increase in slowdown (compared to Local) as communication intensity increases. Between these three, we find that SB does better for workloads with lower communication but worse than the other two at higher communication intensities (workloads 3, 4 and 5). One possible reason for its poor performance for workloads 3 and above is the following. In SB, blocked processes get woken up (via the interrupt service routine) on arrival of a message. Due to the policies of the default Solaris scheduler, these processes mostly receive a priority boost on being woken up. Since DCS boosts the priority of the destination process of a message even if it has not yet called the receive function or when it is spinning but switched out (and not just when it blocks as is done in SB), any reply from the destination in DCS is likely to be sent back faster (thus increasing the likelihood of being coscheduled). DCS-SB, which combines the benefits of DCS (immediate priority boost of the destination process) and SB (limited cycles wasted in spinning), performs even better than DCS.

As mentioned earlier in Section 6.1, it should be noted that the SB mechanism is different from implicit coscheduling presented in [5]. SB does not implement a pairwise spin component as in [5]. It also runs using a different programming model and messaging

layer (MPI and Unet respectively) than [5], which uses Split-C over Active Messages. In Active Messages, the equivalent of a send causes a reply to be sent back by a handler at the remote node. So a receive equivalent following the send can have a better estimate of what is scheduled at the remote node. A corresponding send followed by receive in MPI cannot distinguish between load imbalance and scheduling skews. These factors make it difficult to directly compare the performance of SB presented here with the results for implicit coscheduling presented in [5].

Contrary to expectations, we find SY not performing as well as SB. There are two possible reasons for this. Spinning, despite lowering of priority, instead of blocking may eat away valuable CPU resources. More significant than this is the fact that the priority boost for a destination process of a message is done only when some other process does a receive at that node (and its spin time has expired). This may delay the priority boosting action even further than when it would have happened in spin block, thereby further delaying the reply message. This effect may outweigh the potential benefits of avoiding interrupt processing costs. This suggests that SY should not be used in isolation, but only in conjunction with some other mechanism which boosts the priority much sooner after message arrival. PB-SY and DCS-SY are two such solution approaches. In fact, PB-SY performs better than many schemes for several configurations. DCS-SY performs quite similar to DCS with a small improvement shown for higher communication intensities (when the savings of yield over block are more apparent).

Uniformly, we find that the Periodic Boost (PB) scheme, proposed in this thesis, outperforms almost all other schemes and across all workloads. Even the rate of increase of slowdown (from 30% for workload 1 to 126% for workload 4) is much lower than

the rate of increase for the other schemes. As a result, while it does better than the others for a low communication intensity mix, it does even better (compared to the other schemes) at higher communication intensities. Adding SB to PB does not seem to help significantly, while adding the overheads of blocking and interrupt processing costs. This suggests that we should not use SB in conjunction with PB.

PBT performs worse than PB for all but the highest communication workload. This is is most likely due to the overheads in the scheme. We have also observed that the PBT mechanism is extremely sensitive. Since it uses time-dependent information in making scheduling decisions, its results tend to vary significantly from one run to another. Hence, one should be cautious in making strong pronouncements about the performance of PBT.

	Workload			
	6	7	8	9
LOCAL	106	1115	1088	3393
SB	104	438	301	1344
DCS	104	214	144	664
DCS-SB	101	221	134	525
PB	103	174	176	355
PBT	112	125	136	211
PB-SB	106	411	331	1467
SY	107	936	818	2674
DCS-SY	105	183	145	618
PB-SY	103	171	145	836

Table 6.5. Completion Time in Seconds (Workloads 6 to 9)

Moving to Table 6.5, which uses mixtures of different applications, we can see similar patterns that were observed in Table 6.4. For workload 6 (4 instances of EP), the communication is so low that there is negligible difference between the scheduling schemes. Even Local does as good as any smart scheduling strategy, and there is a slowdown of only around 6% over coscheduling. Even though at the beginning of this section we mentioned that we are not explicitly running sequential jobs concurrently with parallel jobs, this result suggests that EP can be considered a sequential job for most practical purposes. The workloads with EP can thus be viewed as a mix of sequential and parallel jobs.

In Table 6.5, we again find that PB and PBT outperform all other scheduling strategies in terms of the slowdown over coscheduling and controlling the rate of increase of slowdown with increased communication. DCS-SB and DCS come next, with PB-SY close behind. Once again, SB does not do as well at higher communication workloads.

To check the performance of the schemes with a different number of processes per workstation, we run three additional workloads (number 10 through 12) with two processes each. The results for these are given in Table 6.6. At low communication workloads, many scheduling schemes perform equally well. The PB schemes again do well for higher communication workloads, with PB doing the best for workload 11. As mentioned earlier, the benefits of SY are more apparent at higher communication intensities, and we observe that PB-SY and DCS-SY are reasonable choices for workload 12.

As mentioned earlier, the completion time of the last job of a workload may not necessarily be the only metric of importance. While this is important when looking at

	Workload		
	10	11	12
LOCAL	83	128	986
SB	62	75	692
DCS	78	98	240
DCS-SB	65	97	191
PB	63	67	219
PBT	70	69	220
PB-SB	62	87	698
SY	78	104	891
DCS-SY	78	108	270
PB-SY	69	72	164

Table 6.6. Completion Time in Seconds (Workloads 10 to 12)

the throughput of the system, the user (and even the system designer) is interested in lowering the average and variance of turn-around times together with ensuring that a fair share of the CPU is allocated to each process during execution. We should thus examine the completion times of each of the jobs in a workload and their variance, as well as monitor the allocation of CPU(s) to different jobs during execution.

	Workload 3						Workload 5					
	hi	hi	lo	lo	Mean	Coeff. of Var.	hi	hi	hi	hi	Mean	Coeff. of Var.
SB	773	760	139	137	452	0.80	2845	2836	2849	2807	2834	0.01
DCS	350	344	188	186	267	0.35	764	764	755	739	755	0.02
PB	150	152	54	137	123	0.38	287	451	449	409	399	0.19
PBT	252	198	203	210	215	0.11	63	284	264	49	165	0.76
PB-SY	256	269	287	286	274	0.05	719	721	733	719	723	0.01

Table 6.7. Individual Completion Times for Workloads 3 and 5 (in secs)

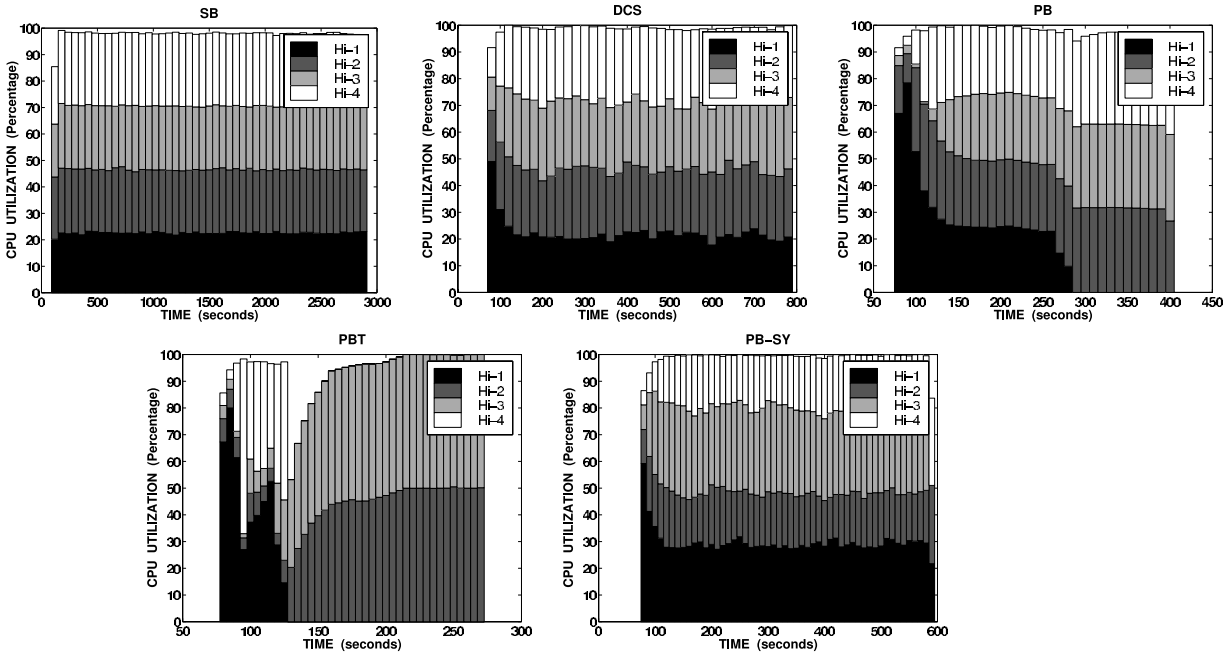


Fig. 6.3. Monitoring CPU Utilization for Workload 5

To examine these issues, we focus specifically on workloads 3 and 5, and the performance of SB, DCS, PB, PBT and PB-SY. Workload 3, with two lo and two hi jobs, and Workload 5, with all four hi jobs, would bring out the effect of heterogeneous and homogeneous communication intensity jobs on the different schemes. Table 6.7 shows the completion times for the individual jobs of the two workloads, the mean completion time and the coefficient of variation (standard deviation divided by the mean). In addition, Figures 6.3 and 6.4 show the percentage allocation of the CPU to the different jobs at different points in the execution at a representative workstation. This has been found by periodically probing for the CPU time allocated to each job and dividing by the probe interval. It should be noted that the completion times in Figures 6.3 and 6.4, and Table 6.7 may not match and may be different from the ones presented earlier because they

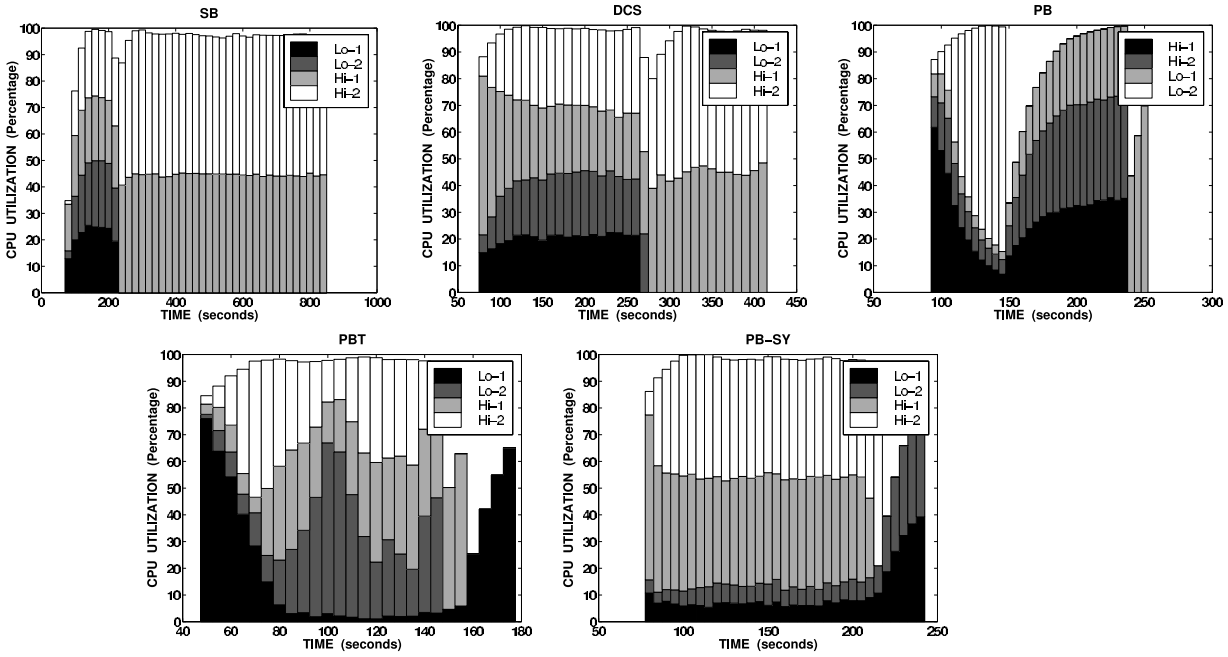


Fig. 6.4. Monitoring CPU Utilization for Workload 3

have been collected at only one representative workstation (and not necessarily at the machine where the maximum time is incurred). The applications do not begin execution at the origin of the X-axis in the graphs. From the user's perspective, a low mean completion time and a low coefficient of variation in completion times is desirable. From the fairness point of view, one would like to see equal CPU allocations to the current jobs in the system within each probe interval.

Focusing first on the homogeneous workload (Workload 5), the four schemes other than PBT have a relatively low coefficient of variation. Of these, PB has a low mean completion time as well, suggesting that this mechanism is preferable over the rest. PBT, which performed well in the total completion time results, is undesirable in terms of the variation in completion times. Even though all four jobs are equally communication

intensive, PBT could end up continuously boosting a single job in successive invocations of the periodic mechanism. In PB, this is avoided by checking message queues in a round-robin fashion. This effect can also be observed in fairness figures (Figure 6.3), where the periodic utilizations are imbalanced for PBT (the bars are not evenly split between the current jobs) compared to the four other schemes. These results suggest that for a homogeneous workload, PB is a good candidate to lower the completion times, has a low coefficient of variation of completion times, and a reasonably equal split of CPU utilizations between current jobs.

Moving to the heterogeneous workload (Workload 3), we find PB, PBT and PB-SY are reasonable from the user's perspective in terms of the mean and coefficient of variation of completion times (Table 6.7). An examination of the fairness criteria (Figure 6.4) shows that SB and DCS give an equal share of the CPU to the current jobs in the system. PB-SY comes next and is fair to the extent that it does not totally starve out a process, but it still favors higher communication jobs. The fairness provided by PB and PBT is undesirable. DCS is a reasonable choice if both criteria are considered together.

6.2.3 Discussion

A lesson learnt from this exercise is that it is important to immediately schedule the destination process (if it is not currently scheduled) for which an incoming message is intended. This achieves two goals. It potentially schedules the destination at the same time as the sender of the message. It allows the destination to send back a reply to the sender (if needed) at the earliest so that the sender does not have to wait longer for the reply. Local scheduling does nothing in this regard, and hence performs poorly.

In Spin Block, the priority is boosted (in the interrupt service routine) only when the destination has blocked waiting for the message. However, if the destination has not yet arrived at the receive point (due to application skews), or even if it has arrived but has been context switched out in the middle of its spin, there is no immediate boost of its priority (to absorb the message). This seems to have a detrimental effect on the performance of Spin Block on a programming model such as MPI, which has a coarser coupling between processes compared to a model such as the one used in [5]. In the implicit scheduling implementation on Split-C/Active Messages, a reply is sent back by the handler at the remote node in many cases. This tends to keep the sender and receiver more closely coupled, and as a result the blocking on a receive is expected to be more effective and is a better estimate of what is scheduled at the remote node. In MPI, the sends and receives are explicit and the effectiveness of our SB depends not just on message latencies and related overheads, but also on application work imbalance. This reiterates the need to study scheduling and communication jointly. One possible way of improving SB could be to keep a tighter coupling within the underlying MPI layer itself. For instance, we could transmit more flow-control messages (than strictly required) to implement the tighter coupling.

While Spin Yield seems attractive in terms of avoiding interrupt costs, the down-side is that the priority boost for the destination is delayed even longer (since it occurs when another process at that node is ready to block). This suggests that Spin Yield should never be used in isolation. However, it can be used in conjunction with other schemes such as Periodic Boost or Dynamic Coscheduling, which boost destination priority more often.

We find Periodic Boost consistently outperforming the other schemes for both high and low communication workloads. Periodic Boost is simple to implement since it does not require any additional functionality in the network interface or the user-level libraries. It does not add any overhead to the critical path of the message transfer mechanism either. Though not explicitly studied in this work, it also offers the flexibility of employing more sophisticated heuristics (than just communication information) in scheduling decisions. Also, it can be used in conjunction with several other heuristics.

However, the "always schedule on arrival" strategy, mentioned above, is not without its pitfalls. It can have a significant impact on the variance of completion times and on the fairness to jobs. This could either lead to a job (which gets coscheduled first) holding on to the CPUs more than the others in a homogeneous workload, or could unfairly favor communication intensive jobs in a mixed workload. For homogeneous workloads, we find that PB is still a good candidate in terms of lowering the coefficient of variation of completion times as well as giving an equal share of the CPU to the current jobs. However, with heterogeneous workloads, PB is inadequate since it favors communication intensive jobs. SB and DCS are more fair under these circumstances. We find that PB-SY does not completely starve out low communication jobs (unlike PB) in a mixed workload, and does ensure that some progress is made though not equally. This is analogous to the traditional UNIX System V scheduler, which can unfairly favor I/O bound jobs in a mixture of CPU and I/O bound jobs though ensuring their individual progress.

These results motivate the need for incorporating fairness criteria into the PB and PB-SY mechanisms, and we plan to explore this issue in our future work. It may be possible to use previously proposed schemes [83, 5, 4] for fairness (such as limiting the

number of priority boosts for a particular process or periodically raising everyone to the highest level) in these mechanisms. In addition to these schemes, it is also possible to use current CPU utilizations in limiting the number of boosts that a process receives [83]. Since the PB function is executed independent of communication events, the scheduling decisions for fairness can also be taken at a different frequency than what is dictated by communication.

Finally one could argue that the coscheduling heuristics perform significantly worse than what one could achieve by just batching the jobs (or space sharing them if there are enough number of nodes). Batching, however, has the well-known problem of poor response times which is particularly disastrous for interactive jobs. With high-performance systems being increasingly used for graphics, visualization, databases and web services, in addition to traditional large-scale scientific applications (short response times are important when debugging large-scale applications as well), we believe that dynamic coscheduling strategies are a more viable option than batching. The disparity in results between these two options should rather serve as a motivation for future research in dynamic coscheduling strategies towards bridging this gap. More recent research [106] has been examining these issues, together with analytical models for these strategies [105].

6.3 Summary

Efficient scheduling of processes on processors offers interesting challenges. This problem is even more important on loosely coupled clusters of workstations where the choice of a scheduling strategy can largely determine the performance and scalability of

the system. From the commercial viewpoint, it is important to use common off-the-shelf (COTS) components to put together a high performance system. The same rationale also suggests that we should use COTS software (ie. the operating system, development tools etc.) in putting together this system. As a result, it is rather tempting to just leave it to the native operating system scheduler to take care of managing the processes assigned to a workstation. However, it is well-known that such a *local* scheduling policy would be extremely inefficient for fine-grained communication. Coscheduling (or gang scheduling) falls at the other end of the spectrum, wherein there is strict coordination between the operating systems of different nodes to ensure that communicating processes of an application are scheduled on their respective nodes at the same time. The problem with coscheduling is that it is impractical to implement on a loosely coupled environment such as a cluster, where the costs of communication and synchronization are high. Further, coscheduling may not be a very scalable option from a reliability perspective.

Two operations, namely, waiting for a message and receipt of a message, have been traditionally used to examine local information and take remedial action to guide the system execution towards coscheduling (without explicitly requiring extra communication/synchronization). Independently combining the possible actions for each of these two operations, a 3×3 design space of scheduling strategies has been presented in this part of the thesis. Five of the these schemes are original contributions of this work, while the remaining four have been proposed in the past. This work exhaustively evaluates the pros and cons of this design space through implementations on a uniform platform consisting of a cluster of Sun UltraSPARC workstations, running Solaris 2.5.1, connected

by Myrinet. These schemes have been evaluated using varying mixes of several real parallel applications with different communication intensities. We find the newly proposed Periodic Boost (PB) scheme outperforming the others over a range of different workloads.

Chapter 7

Conclusions

Cluster computing is a very cost-effective way of satisfying the ever-increasing need for computation. The main characteristic which distinguishes a cluster is the communication network that links its various nodes. This thesis has examined some critical issues related to cluster communications and the way that it impacts the overall performance offered by the cluster. In particular it examines the use of user-level networking in improving the performance of communication and scheduling for cluster workloads.

The thesis starts off by looking at whether user-level networking has the potential for improving cluster communication performance. In the work on pSNOW in Chapter 2, a simulation-based study examines the use of user-level communication in a typical cluster. In it, three kinds of network interface hardware with varying degrees of "intelligence" are combined with three kinds of messaging software and their performance is compared using microbenchmarks and a few scientific application kernels. Broadly, the study shows that user-level networking does indeed deliver performance gains to cluster workloads compared to the traditional OS kernel based approach. An interesting secondary result shows the need for the network interface processor to be about half as powerful as the main host CPU. The fraction is not so interesting as the revelation about the need for a powerful CPU on the network interface when we enter the realm of user-level networking.

The next part of the work uses an experimental approach to reveal the design decisions that need to be made for implementing user-level networking. Using a state-of-the-art network interface card (Myrinet), it examines the performance impact of several aspects of a working design. In particular, it focusses on the challenge of providing protected access to the network interface hardware to multiple users. The results show that such access can be provided while maintaining good performance.

These two parts of the thesis together provide a proof-of-concept of user-level communication for clusters. They show that not only does the idea have inherent merits but that it can be realized using affordable off-the-shelf network interface cards. But a proof-of-concept is not sufficient to propose the widespread adoption of user-level networking and that is why the third and fourth parts of the thesis look at two other important aspects of the issue.

Chapter 4 addresses the all-important issue of scalability of user-level networking. It looks at how performance of ULN degrades with an increase in both, the number of connections between cluster nodes and in the load that each connection carries. The study shows that the simple design used earlier offers the best single-channel performance but is not scalable. The work then goes on to suggest ways to improve the scalability through changes in the software and hardware of the networking subsytem. The improvements proposed are validated through simulations. The fundamental trade-off between single-program performance and system utilization manifests itself in this part of the work. Depending on the expected workload (or rather the expected communication load), system designers could choose one or more of the proposals to meet their performance criteria. The scalability study introduces the ideas of resource management

into the realm of user-level networking. The goal of providing adequate performance to a large number of channels naturally leads us to investigate the behaviour of the system when it is overloaded. At such a time it is necessary to start questioning which of the channels should get the resources needed to maintain acceptable performance and to what extent other channels should get penalized for the same. So far the assumption had been that all communication channels are equal and the system has to try and distribute resources such as network processor time, wire bandwidth as best as it can. Relaxing this assumption brings the thesis to Chapter 5 which investigates the provision of quality of service in user-level networking. Quality of service is a broad term even in the context of networking and this part of the work focusses on providing latency and bandwidth guarantees to communication channels. It looks at the design implications of trying to provide such guarantees in the context of a scalable ULN implementation. While the results do indicate that QoS can be provided with a moderate amount of hardware support on the network interface, it is not clear whether it is desirable to do so. For cluster communication, QoS guarantees to applications typically require support in more than just the networking subsystem. In user-level networking, it would not be very useful to ensure that incoming data arrived with low jitter if it did not also mean that applications would be able to be scheduled in time to receive and process this data or be assured that they could allocate enough buffers for it. As a result, this part of the thesis motivates further investigation into the issue of QoS provisioning and whether it is appropriate to do it for ULNs.

The final part of the work deals with what would appear to be an orthogonal issue : the scheduling of processes of a parallel job on the nodes of a cluster. However, Chapter 6 reveals the close coupling between scheduling and communication in parallel programs. Ignoring this coupling is shown to have severe performance penalties. User-level communication offers a unique opportunity of influencing scheduling by using information provided by the communication layer. This thesis classifies the space of communication-driven coscheduling strategies and outlines five new schemes within it. The main ideas behind the five schemes are implemented and shown to significantly outperform the state-of-the-art in the area. More importantly, they underscore the importance of looking at closely coupled subsytems as a whole while making performance related design decisions.

Overall, the thesis presents new insights into various facets of user-level networking in the cluster environment. Alongwith work done at other universities and research institutions, it serves to underline the importance of this new networking technology. At the time of writing, it looks like industry is well on its way to standardizing some of the more proven aspects of user-level networking in the form of the Infiniband architecture [54].

7.1 Future Work

Each of the topics explored in the thesis has considerable scope for further research. The most important of these is the examination of the questions that remain unanswered by the quality of service work. It is not clear whether the network interface

card (NIC) is the right place for implementing QoS, even for the networking layer. It appears that to support a reasonable number of communication channels with QoS needs, the NIC would need more hardware support than is currently available. This brings up issues of cost-effectiveness which is the very foundation of modern cluster computing. Even if the hardware support is not a problem, it is clear that the host OS kernel has access to a lot more information that would enable it to make better QoS decisions. While the OS cannot be in the critical path of communication by definition, it would be interesting to see if it can help some other entity (e.g. the NIC) make more informed QoS related decisions. Finally, the issue of QoS provisioning might be better addresses in the context of a hostwide or even clusterwide resouce management system.

The scalability work is another rich source of research work. Hardware support for easing some of the scalability bottlenecks has only been addressed partially by this thesis. Is a programmable NIC suitable for high performance user-level networking or is its flexibility a liability ? Would it be better to have hardware controlled queues and direct paths from the I/O bus to the wire with the NIC processor playing a much smaller role in data transfer ? If the system has to support a larger number of channels, should it use more NIC's on the same I/O bus or should it focus on making a single NIC handle greater load ? The answers to these questions have a direct impact on the results of the study in this thesis.

Finally, the scheduling part of the thesis opens up a whole host of issues, some of which are being actively addressed by other students in the same group. The basic mechanism of dynamic coscheduling has been established through implementation using a constant sized workload. This immediately brings up questions about its suitability for

different classes of parallel workloads. It is also important to know the effect of system parameters such as interrupt latency, I/O bus characteristics and multiprogramming levels on the performance of the proposed schemes. [106] looks at some of these issues using simulations.

7.2 The Ideal Network Interface

Much of the research done in this thesis has centered around the network interface. Undoubtedly there is much more work that needs to be done before definitive trends can be established in any of the areas explored. Yet it is useful to step back and enunciate what would be an ideal network interface, based on the experiences of the work done.

One observation has been made time and again in this work : the network interface hardware needs to be more powerful than it is today. The elimination of the operating system kernel from the networking path often eliminates the use of host resources as well. The primary motivation for ULN is to increase the performance of the networking layer. These conflict between non-availability of host resources and need for increased performance demands that network interface hardware be reasonably powerful. Providing more hardware is not merely a matter of increasing network processor speed or memory. Rather, it depends on a hardware assist to the most critical parts that are specific to ULN. A hardware queue which allows a trusted interface (say the consumer) to remove elements from arbitrary positions but enforces some queuing discipline for the untrusted interfaces (say the producers) has a lot of potential for improving scalabilty, providing QoS and for reducing the demands on the NIC CPU and firmware. Another useful feature would be the ability to directly stream data from the I/O bus interface

of a NIC to and from its interface with the wire/switch. The NIC CPU typically needs to process only the first few bytes of incoming and outgoing messages and once it has determined the direction and ordering of the data stream, neither it nor the NIC memory needs to be involved in performing the data transfer. Of course, there always has to be the option of buffering these streams using NIC memory in order to affect the ordering and timing of message delivery.

Other than hardware support, there is a strong need for interfaces to efficiently export information that is currently known only to the OS kernel. For trusted host applications, such an interface already exists in the form of the /proc filesystem (on Linux). For a similarly trustworthy NIC firmware, there is no similar interface. Part of this problem has to do with their being separated by an I/O bus. But with so many periodic tasks being performed by the OS kernel or by its trusted daemons, it might possible to ensure that the NIC and the firmware on it have access to the data they need to make better decisions. This could involve peer devices (with respect to the I/O bus) such as disks, as staging areas for data that could then be retrieved by the NIC.

Network interfaces have occupied an important role in the widespread deployment of clusters and clusters have, in turn, been very useful in meeting the computational demand of this electronic age. It is hoped that this thesis has played a small but useful role in these developments.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proceedings of the ACM SIGMETRICS 1997 Conference on Measurement and Modeling of Computer Systems*, pages 225–236, June 1997.

[2] A.Indiresan, A.Mehra, and K.G.Shin. The END: A Network Adapter Design Tool. In *Proceedings of INFOCOM*, March 1998.

[3] R. H. Arpaci et al. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 267–278, 1995.

[4] A. C. Arpaci-Dusseau and D. E. Culler. Extending Proportional Share Scheduling to a Network of Workstations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1997.

[5] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the ACM SIGMETRICS 1998 Conference on Measurement and Modeling of Computer Systems*, 1998.

[6] NERSC PC Cluster Project at Lawrence Berkeley National Laboratory). M-VIA: A High Performance Modular VIA for Linux. Available at http://www.nersc.gov/research/FTG/via.

[7] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[8] S. Balakrishnan and F. Ozguner. A Priority-Based Flow Control Mechanism to Support Real-Time Traffic in Pipelined Direct Networks. In *Proc. Intl. Conf. on Parallel Processing*, volume I, pages 120–127. IEEE CS Press, August 1996.

[9] A. Banerjee. Implementation and Evaluation of MPI over Myrinet. Master's thesis, Dept. of Computer Science and Engineering, Penn State University, University Park, PA 16802, May 1999.

[10] E. Barton, J. Cownie, and M. McLaren. Message Passing on the Meiko CS-2. *Parallel Computing*, 20(4), April 1994.

[11] M. Bazikazemi, V. Moorthy, L. Herger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters. In *Proceedings of International Parallel and Distributed Processing Symposium*, May 2000.

[12] F. Berry, E. Deleganes, and A. M. Merritt. *The Virtual Interface Architecture Proof-of-Concept Performance Results.* Intel Corp. Available at ftp://download.intel.com/design/servers/vi/via_proof.pdf.

[13] R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal. Efficient Multicast on Myrinet using Link-Level Flow Control. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 381–390, 1997.

[14] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[15] M.A. Blumrich, R.D. Alpert, Y. Chen, D.W. Clark, S.N. Damianakis, C. Dubnicki, E.W. Felten, L. Iftode, K. Li, M. Martonosi, and R.A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[16] N. J. Boden et al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[17] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS : A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1991.

[18] M. Buchanan and A. Chien. Coordinated Thread Scheduling for Workstation Clusters under Windows NT. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[19] P. Buonadonna, A. Geweke, and D. E. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of Supercomputing '98*, November 1998.

[20] G. D. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 245–259, October 1996.

[21] M. B. Caminero, F. J. Quiles, J. Duato, D. S. Love, and S. Yalamanchili. Performance Evaluation of the Multimedia Router with MPEG-2 Video Traffic. In *Network Based Parallel Computing : Proc. Third Intl. Workshop on Communication, Architecture and Applications on Network Based Parallel Computing (CANPC'99)*, Lecture Notes in Computer Science, pages 62–76. Springer, January 1999.

[22] M. S. Chen and K. G. Shin. Processor Allocation in an N-Cube Multiprocessor Using Gray Code. *IEEE Transactions on Computer Systems*, 36:1396–1407, December 1987.

[23] CLARiion, Dell, Giganet, IBM, Intel, Qlogic, and Visual Insights Corporations. Demonstrating the Benefits of Virtual Interface Architecture. Available at http://www.giganet.com/technology.

[24] K. Connelly and A. A. Chien. FM-QoS: Real-time Communication using Self-synchronizing Schedules. In *Proceedings of Supercomputing'97*, November 1997.

[25] Compaq Corp., Intel Corp., and Microsoft Corp. Virtual Interface Architecture Specification, Version 1.0. Available at http://www.viarch.org.

[26] Dell Computer Corp. Virtual Interface Architecture for Clustered Systems. Available at http://www.dell.com/r&d/whitepapers/wpvia/wpvia.htm.

[27] Intel Corp. Intel Virtual Interface Architecture Conformance Suite Users Guide, Version 0.5. Available at http://developer.intel.com/design/servers/vi/developer.

[28] Intel Corp. Intel Virtual Interface Architecture Developer's Guide, Revision 1.0. Available at http://developer.intel.com/design/servers/vi/developer.

[29] Tandem(Compaq) Corp. ServerNet and the VI Architecture. Available at http://www.servernet.com.

[30] Cray Research, Inc., Minnesota. *The Cray T3D System Architecture Overview Manual*, 1993.

[31] H. Davis, S. R. Goldschmidt, and J. L. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II 99–107, 1991.

[32] Martin de Prycker. *Asynchronous Transfer Mode: solution for broadband ISDN.* Ellis Horwood, West Sussex, England, 1992.

[33] R. Dimitrov and A. Skjellum. *An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing.* MPI Software Technology Inc. Available at http://www.mpi-softtech.com/.

[34] J. Ding and L. N. Bhuyan. An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:193–200, August 1993.

[35] X. Du and X. Zhang. Coordinating parallel processes on Network of Workstations. *Journal of Parallel and Distributed Computing*, 46:125–135, 1997.

[36] José Duato, Sudhakar Yalamanchili, M. Blanca Caminero, and Francisco J. Quiles. MMR: A High Performance Multimedia Router Architecture and Design Trade-offs. In *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA-5)*, January 1999.

[37] 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*, August 1997.

[38] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 1996.

[39] H. Eberle and E. Oertli. Switcherland: A QoS Communication Architecture for Workstation Clusters. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 98–108, July 1998.

[40] B. Falsafi and D. A. Wood. Scheduling Communication on a SMP Node Parallel Machine. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, February 1997. To appear.

[41] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[42] D. G. Feitelson and L. Rudolph. Coscheduling Based on Runtime Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[43] Fore Systems, Inc., Pittsburgh, PA 15238. *TCA-100 TURBOchannel ATM Computer Interface*, 1992.

[44] Fore Systems, Inc., Warrendale, PA. *ForeRunner SBA-100/200 ATM SBus Adapter User's Manual*, 1995.

[45] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, Inc., Reading, MA, 1995.

[46] D. Garcia and W. Watson. Servernet II. In *Proc. of the 1997 Parallel Computing, Routing, and Communication Workshop (PCRCW'97)*, June 1997.

[47] M. Gerla, B. Kannan, B. Kwan, P. Palnati, S. Walton, E. Leonardi, and F. Neri. Quality of Service Support in High-Speed, Wormhole Routing Networks. In *Proc. Intl. Conference on Network Protocols*, October 1996.

[48] Giganet. cLAN Product Overview. Available at http://www.giganet.com/products/overview.htm.

[49] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of 2nd Symposium on Operating System Design and Implementation*, pages 107–122, October 1996.

[50] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[51] G. Henley, N. Doss, T. McMahon, and A. Skjellum. BDM: A Multiprotocol Myrinet Control Program and Host Application Programmer Interface. Technical Report MSSU-EIRS-ERC-97-3, ICDCRL, Dept. of Computer Science and HPCL, NSF Engineering Research Center for Computational Field Simulation, Mississippi State University, May 1997.

[52] Myricom Inc. M2F-PCI32C network interface card. Available at http://www.myri.com/myrinet/PCI/m2f-pci32.html.

[53] Myricom Inc. PCI64 Programmers Documentation. Available at http://www.myri.com/myrinet/PCI64/programming.html.

[54] Infiniband Trade Association. http://www.infinibandta.org.

[55] Intel Corporation, Oregon. *Paragon User's Guide*, 1993.

[56] B. Kim, J. Kim, S. Hong, and S. Lee. A Real-Time Communication Method for Wormhole Switching Networks. In *Proc. Intl. Conference on Parallel Processing*, pages 527–534, August 1998.

[57] J. H. Kim. *Bandwidth and Latency Gurantees in Low-Cost, High-Performance Networks*. PhD thesis, Department of Electrical Engineering, University of Illinois at Urbana-Champaign, 1997.

[58] J. H. Kim and A. A. Chien. Rotating Combined Queueing (RCQ): Bandwidth and Latency Gurantees in Low-Cost, High-Performance Networks. In *Proc. Intl. Symp. on Computer Architecture*, pages 226–236, May 1996.

[59] A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler, and K. Yellick. Evaluation of Architectural Support for Global Address-based Communication in Large Scale Parallel Machines. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, October 1996.

[60] R. Krishnamurthy, K. Schwan, R. West, and M. Rosu. A Network Co-processor-Based Approach to Scalable Media Streaming in Servers. In *Proceedings of the 2000 International Conference on Parallel Processing*, August 2000.

[61] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.

[62] J-P. Li and M. Mutka. Priority Based Real-Time Communication for Large Scale Wormhole Networks. In *Proc. Intl. Parallel Processing Symposium*, pages 433–438, May 1994.

[63] L.Prylli and B.Tourancheau. BIP: a new protocol designed for high performance. In *PC-NOW Workshop held in parallel with IPPS/SPDP98, Orlando,USA*, March30-April 3 1998.

[64] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of Supercomputing '97*, November 1997.

[65] Message Passing Interface (MPI) Forum. *Document for a standard Message Passing Interface*, 1994.

[66] Microelectronics and Computer Technology Corporation, Austin, TX 78759. *CSIM User's Guide*, 1990.

[67] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[68] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 96–105, June 1999.

[69] S. Nagar, A. Sivasubramaniam, J. Rodriguez, and M. Yousif. Issues in Designing and Implementing a Scalable Virtual Interface Architecture. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 405–412, August 2000.

[70] NCUBE Company. *NCUBE 6400 Processor Manual*, 1990.

[71] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[72] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[73] E. W. Parsons and K. C. Sevcik. Coordinated Allocation of Memory and Processors in Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 57–67, June 1996.

[74] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I–237–240, August 1993.

[75] S. K. Reinhardt et al. The Wisconsin Wind Tunnel : Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[76] Jennifer Rexford, John Hall, and Kang G. Shin. A Router Architecture for Real-Time Point-toPoint Networks. In *Proc. Intl. Symp. on Computer Architecture*, pages 237–246, May 1996.

[77] O. Rose. http://nero.informatik.uni-wuerzburg.de/MPEG/.

[78] S. Setia. Trace-driven Analysis of Migration-based Gang Scheduling Policies for Parallel Computers. In *Proceedings of the 1997 International Conference on Parallel Processing*, August 1997.

[79] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[80] A. Sivasubramaniam, P. Hatcher, and M. Annaratone. Communication Mechanisms for Workstation Farms. Internal Technical Report, Digital Equipment Corporation, Maynard, MA, December 1992.

[81] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[82] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. On characterizing bandwidth requirements of parallel applications. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 198–207, May 1995.

[83] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1997.

[84] P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75, April 1995.

[85] H. Song, B. Kwon, and H. Yoon. Throttle and Prempt: A New Flow Control for Real-Time Communications in Wormhole Networks. In *Proc. Intl. Conf. on Parallel Processing*, pages 198–202. IEEE CS Press, August 1997.

[86] W.E. Speight, H. Abdel-Shafi, and J.K. Bennett. Realizing the performance potential of the Virtual Interface Architecture. In *Proceedings of the ACM 1999 International Conference on Supercomputing*, June 1999.

[87] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[88] R. Tewari, R. Mukherjee, D. Dias, and H. Vin. Design and Performance Tradeoffs in Clustered Video Servers. In *International Conference on Multimedia Computing and Systems*, June 1996.

[89] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. In *Lecture Notes in Computer Science, 1225*, pages 708–717. Springer-Verlag, April 1997.

[90] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[91] A. Tucker. *Efficient Scheduling on Shared-Memory Multiprocessors.* PhD thesis, Stanford University, November 1993.

[92] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared Memory Multiprocessors. In *Proceedings of MASCOTS '94*, pages 201–207, February 1994.

[93] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[94] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[95] R. Wang, A. Krishnamurthy, R. Martin, T. Anderson, and D. Culler. Modeling Communication Pipeline Latency. In *Proceedings of the ACM SIGMETRICS 1998 Conference on Measurement and Modeling of Computer Systems*, June 1998.

[96] R. West, R. Krishnamurthy, W. Norton, K. Schwan, S. Yalamanchili, M. Rosu, and S. Chandra. Quic: A quality of service network interface layer for communication in NOWs. In *Proceedings of the Heterogeneous Computing Workshop, in conjunction with IPPS/SPDP, San Juan, Puerto Rico*, April 1999.

[97] C. Z. Xu and F. C. M. Lau. *Load Balancing in Parallel Computers: Theory and Practice.* Kluwer Academic Publishers, Boston, MA, 1996.

[98] M-T. Yang. *An Automatic Pipelined Scheduler for Real-Time Vision Applications.* PhD thesis, Dept. of Computer Science & Eng., The Pennsylvania State University, September 2000.

[99] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–55, May 1996.

[100] K. Yocum, J. Chase, A. Gallain, and A. R. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, August 1997.

[101] K. Yum, A. Vaidya, C. Das, and A. Sivasubramaniam. Investigating QoS Support for Traffic Mixes with the MediaWorm Router. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, pages 97–106, January 2000.

[102] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.

[103] H. Zhang and S. Keshav. Comparison of rate-based service disciplines. In *Proceedings of the conference on Communications architecture & protocols*, pages 113 – 121, 1991.

[104] L. Zhang. VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks. *ACM Trans. on Computer Systems*, 9(2):101–124, May 1991.

[105] Y. Zhang and A. Sivasubramaniam. Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 2001.

[106] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. In *Proceedings of the ACM 2000 International Conference on Supercomputing*, pages 100–109, May 2000.

Vita

Shailabh Nagar completed an M.S (Integrated) in Mathematics and Computer Applications from the Indian Institute of Technology, Delhi (India) in 1995. Thereafter, he joined the Department of Computer Science and Engineering at the Pennsylvania State University for the PhD program. He worked as a research and teaching assistant till he defended his thesis in December 2000. He received the department's Graduate Research Assistant Award for Outstanding Research for the year 1999. His doctoral research interests were in communication and scheduling support for cluster computing. Specifically, his work dealt with the design, implementation and performance evaluation of scalable, user-level network interfaces, quality of service provision and communication driven coscheduling.

In Summer 1996, Shailabh was an intern at the Unix Adapter Group at FORE Systems, Inc. in Pittsburgh, PA where he worked on ATM device drivers.In Summer 1999, he did a research internship at the Performance Development group of IBM Corp.'s PC Server division. His work there focussed on scalability issues for network interfaces and resulted in the submission of two patent applications. Shailabh is currently working on scalability and security of the Linux operating system at IBM's T.J. Watson Research Center in Yorktown Heights, NY.