NETWORK INTERFACE FOR PROTECTED, USER-LEVEL COMMUNICATION

Matthias Augustin Blumrich

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1996

© Copyright by Matthias Augustin Blumrich 1996
All Rights Reserved

To my parents, Christoph and Susanne,
and my wife, Aimee

Abstract

Scalable parallel architectures are converging on the general framework of a distributed memory parallel processor (DMPP), consisting of a group of interconnected compute nodes, each typically consisting of a CPU, some memory, and a network interface. DMPP architectures range from tightly-coupled multicomputers to loosely-coupled networks of personal computers (PCs), and support a number of parallel programming paradigms including shared-memory and message-passing. Networks of PCs are very attractive DMPPs because of the high performance and low cost of the PC nodes.

Custom-built multicomputers have one principal advantage over networks of PCs: they are faster. The major reason for this advantage is a gap in communication performance between multicomputer interconnects and local area networks. Recent networking trends have seen a migration of technology from multicomputers to networks of PCs, shrinking this communication performance gap considerably. However, software overhead—largely attributable to local area network interfaces—continues to maintain the performance gap.

This dissertation describes how network interfaces can bridge the gap by providing appropriate support for protected, user-level communication with extremely low software overhead. Such network interfaces can be used to build scalable, high-performance, really inexpensive multiprocessors out of low-cost, commodity components.

This assertion is demonstrated via a working implementation of a prototype SHRIMP network of PCs. The prototype achieves data transfer latency competitive with custom multicomputers, and efficient use of bandwidth down to small message

sizes. In addition, SHRIMP supports general multiprogramming, extending its benefits from uniprocessors to multiprocessors.

The key to SHRIMP performance is virtual memory-mapped communication, which strips away overhead from time-critical data transfer operations without sacrificing all the services that systems have come to depend on. Equipped with virtual memory-mapped communication, networks of PCs could very well become commodity scalable systems.

Acknowledgements

There's a proverb that says a journey of a thousand miles begins with a single step, which is why I am dedicating this dissertation to my parents. My desire to seek higher education could undoubtably be traced all the way back to my formative years. I am also dedicating this dissertation to my wife, who helped me in countless ways and only *sometimes* doubted I'd ever finish...

I owe a big debt of gratitude to the dream team of advisors: Kai Li, Edward Felten, and Douglas Clark. I feel very fortunate to have been surrounded by such talent, and I hope some of it rubbed off. I am especially grateful to my principal advisor, Kai, who always found new ways to challenge me and bring out my best work. He is an inspiration and an example.

I have really enjoyed working with so many dedicated graduate students and technical staff members on the SHRIMP project; notably Cezary Dubnicki, Liviu Iftode, Malena Mesarina, Bin Wei, Richard Alpert, Stefanos Damianakis, Yuqun Chen, Angelos Bilas, and Wenjia Fang. Special thanks to Malena for her dedication and enthusiasm, and for building the *first* SHRIMP network interface, which paved the way for the design described in this dissertation.

Richard Lipton and Jonathan Sandberg were instrumental in guiding the early SHRIMP design decisions with their experience and insight. Jaswinder Pal Singh and Margaret Martonosi provided lots of excellent suggestions and critiques. Many people at the Intel Scalable Systems Division have helped us with SHRIMP, especially Roger Traylor, Dave Dunning, and Jonathan Lee.

The department of computer science is a wonderful place to work, largely because of the excellent staff. Special thanks to Michael Brantley for setting up our CAD

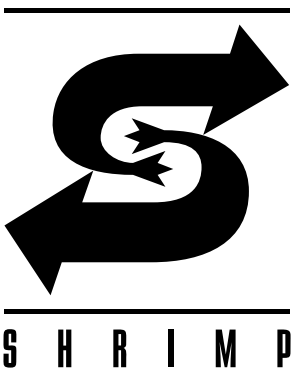
environment and keeping it running, and Patricia Zabielski for cheerfully managing the tangle of SHRIMP purchase orders.

I cannot be accused of all work and no play. Thanks to Adam Buchsbaum for many great Cache Hitters softball games, and Stefanos Damianakis and Chris Dunworth for taking us all the way to the top of the league in 1995! Over the years, a good deal of my spare time was spent with "Big Wave" Dave Heyman, who introduced me to the region's best attractions, and would never forgive me if I didn't mention him in my dissertation....

I have Stefanos to thank for pointing out that my thesis "provides hope for everybody," and for being a great officemate and all-around friend. Other memorable officemates include Jim Plank, Mark Greenstreet, Karin Petersen, and K. Balasubramanian.

This work was funded by ARPA grants N00014-91-J-4039 and N00014-95-1-1144, ARPA contract DABT63-92-C-0053, NSF grant MIP94-20653, and the Intel Corporation. I am grateful to ARPA and NASA for two years of support through an Assistantship in Parallel Processing. Much thanks goes to Johanna Weinstein and Larry Davis at the Institute for Advanced Computer Studies, University of Maryland for administering the assistantship, and providing more than just financial support.

The SHRIMP project logo below is the excellent work of my father, Christoph.



Contents

Abstract iv

Acknowledgements vi

1 Introduction 1

1.1 Thesis 1
1.2 Main Ideas 6
1.3 SHRIMP 8
1.4 Main Results 9
1.5 Related Work 9
1.5.1 Traditional DMA-Based Network Interfaces 9
1.5.2 Communication Coprocessors 11
1.5.3 Processor-Integrated Network Interfaces 13
1.5.4 Memory-Mapped Network Interfaces 14
1.5.5 Eager-Sharing Network Interfaces 15
1.5.6 Direct Memory Data Transfer 17
1.5.7 Remote Memory Reference 18
1.5.8 Implicit Multiprocessor Communication 18
1.5.9 Address Mapping Techniques 19

2 Virtual Memory-Mapped Communication 20

2.1 The VMMC Model 22
2.1.1 Import-Export Mappings 23
2.1.2 Transfer Strategies 24

2.1.3 Notifications . . . 25
2.2 VMMC Application Programming Interface . . . 25
2.2.1 Destination Proxy Space . . . 25
2.2.2 Export . . . 26
2.2.3 Import . . . 27
2.2.4 Deliberate Update . . . 27
2.2.5 Automatic Update . . . 28
2.2.6 Destruction of Mappings . . . 29
2.2.7 Notifications . . . 30
2.3 A VMMC Example . . . 31
2.4 Summary . . . 34

3 Hardware Support for VMMC **36**
3.1 Support for Direct Data Transfer . . . 37
3.1.1 Global naming . . . 37
3.1.2 Direct Virtual Memory Transfer . . . 38
3.1.3 Direct Physical Memory Transfer . . . 39
3.2 Support for Automatic Update . . . 40
3.2.1 Snooping the CPU . . . 41
3.2.2 Snooping the First-Level Cache . . . 42
3.2.3 Snooping the Second-Level Cache . . . 43
3.2.4 Snooping the Memory Bus . . . 44
3.2.5 Snooping the I/O Expansion Bus . . . 44
3.3 Support for Deliberate Update . . . 45
3.3.1 System-Level . . . 45
3.3.2 User-Level . . . 45
3.4 Notifications . . . 52
3.5 The Impact of Hardware Support . . . 52
3.6 Conclusions . . . 55

4 The SHRIMP Network Interface **57**
4.1 Design Goals and Constraints . . . 57

4.1.1 Low-Overhead Communication 57
4.1.2 Support for Multiprogramming 58
4.1.3 Bus Bandwidth 59
4.1.4 Use of Commodity Components 59
4.1.5 Hardware Design Simplicity 59
4.2 Architecture of the SHRIMP Prototype 60
4.2.1 Hardware Components 60
4.2.2 Software Components 62
4.2.3 The Network Interface 63
4.3 Implementation of Automatic Update 63
4.3.1 Binding for Automatic Update 64
4.3.2 Hardware Support for Automatic Update 66
4.3.3 Kernel Extensions for Automatic Update 67
4.3.4 Automatic Update Pros and Cons 67
4.4 Implementation of Deliberate Update 68
4.4.1 Memory Mapping for User-Level DMA 69
4.4.2 Use of the Network Interface Page Tables 70
4.4.3 Interpretation of Proxy Space 70
4.4.4 Providing Protection 71
4.4.5 User-Level DMA Restrictions 72
4.4.6 User-Level DMA Transfer Initiation 72
4.4.7 User-Level DMA Hardware 73
4.4.8 Operating System Support for UDMA 79
4.4.9 Summary of UDMA 81
4.4.10 Deliberate Update Pros and Cons 82
4.5 Implementation of Notifications 83
4.6 SHRIMP Network Interface Datapath 84
4.6.1 Arbitration 86
4.6.2 Flow Control 88
4.6.3 Support for Performance Monitoring 90
4.6.4 Un-Supported Features 91

4.7 Fault Tolerance 93
4.8 Conclusions 94

5 Prototype Performance **96**
5.1 Network Interface Hardware Parameters 98
5.2 Experimental Prototype Configuration 99
5.3 Hardware Datapath Latency 100
5.3.1 Outgoing Automatic Update Send 100
5.3.2 Network Transfer 101
5.3.3 Incoming Receive 102
5.3.4 Outgoing Deliberate Update Send 103
5.3.5 Summary 105
5.4 Automatic Update Latency 105
5.5 Deliberate Update Bandwidth 107
5.5.1 Effect of EISA Bus Possession 108
5.5.2 Effect of Deliberate Update Packet Size 111
5.6 Automatic Update Bandwidth 115
5.6.1 Effect of Automatic Update Packet Size 116
5.6.2 Latency-Bounded Combining 118
5.7 The Effect of Timestamping 121
5.8 Cache Effects 122
5.8.1 PC Cache Architecture 122
5.8.2 Automatic Update 123
5.8.3 Deliberate Update and Incoming DMA 124
5.8.4 Discussion 125
5.9 Conclusions 127

6 Conclusions **131**
6.1 Communication Performance Gap Revisited 131
6.2 Two Low-Overhead Transfer Strategies 132
6.3 A Flexible Interface 133
6.4 The Hard Choices 135

6.5 A Few More Words About UDMA 136
6.6 A Project with a Future 137

A Network Interface Details **138**
A.1 Outgoing Page Table Entry 138
A.2 Incoming Page Table Entry 139
A.3 Network Packet Format 140
A.4 Deliberate Update 142

Bibliography **145**

List of Figures

1 The basic architecture of a distributed memory parallel processor. . . 2
2 Message passing latency on the Intel Paragon multicomputer. 10
3 Some virtual memory mappings between processes on two nodes. . . . 21
4 A receive buffer imported as a proxy receive buffer. 26
5 A simple data transfer example. 31
6 Basic data transfer using traditional message passing. 32
7 Basic data transfer using VMMC. 33
8 Automatic update binding of virtual memory spaces. 40
9 A typical memory hierarchy. 41
10 The network interface page table maintains bindings between the physical pages of the source data and the receive buffer. 47
11 A large transfer performed as a series of deliberate updates, each within source and destination page boundaries. 51
12 Architecture of the prototype SHRIMP system. 60
13 Hardware and software layers in the prototype SHRIMP system. Custom components are shaded. 62
14 Datapath of the SHRIMP network interface. 64
15 Automatic update binding between the physical pages of the source data and the receive buffer. 66
16 Specification of a deliberate update binding in virtual memory space. 69
17 Mappings for UDMA destination specification. 70
18 Mappings for UDMA source specification. 71

19 Traditional DMA hardware configured for a memory to device transfer. In general, transfer in both directions is possible. 73
20 UDMA hardware configured for a memory to network interface transfer. 76
21 State transitions in the UDMA hardware state machine. 77
22 Datapath of the SHRIMP network interface. 84
23 Bandwidth of deliberate update transfers as a percentage of the maximum measured bandwidth. 97
24 Hardware latency components of a single-word, automatic update transfer. Solid lines indicate fixed latencies and dashed lines indicate variable latencies. 101
25 Latency components of a single-word, deliberate update transfer. . . . 104
26 The inner-loop code for optimized ping-pong. 105
27 Block sending program used to measure deliberate update bandwidth. 108
28 Latency of a 4 Kbyte transfer with packet sizes of 4 Kbytes and 1 Kbyte.112
29 Variable definitions for the latency components of a packet transfer. . 112
30 Bandwidth vs. transfer size for various settings of the deliberate update packet size parameter. 114
31 Bandwidth vs. transfer size for various settings of the automatic update packet size parameter. 117
32 Bandwidth of a one-page (4096-byte) automatic update transfer as the combining delay is varied. 119
33 Bandwidth of a four-page (16384-byte) automatic update transfer as the combining delay is varied. 120
34 The effect of timestamping on the latency of various small transfers. . 122

List of Tables

1 Illustration of the communication performance gap between LAN-based DMPPs and multicomputers. 5

2 Components of message passing overhead on three kinds of network interface designs. 54

3 Maximum deliberate update bandwidth vs. EISA bus possession time. 109

4 Bandwidth of the `bcopy` function with various source caching. The source is cached write-back and the destination is cached write-through. 123

5 Bandwidth of 4096-byte DMA transfers between memory and the network interface. 125

6 Bandwidth comparison between automatic update and deliberate update with source data caching considered. 126

Chapter 1

Introduction

1.1 Thesis

2048 Hands are Better than One

Scalable systems research is motivated by the belief that two processors are better than one, but 2048 are better still. This belief forms the foundation of a "divide and conquer" approach to computation, where parallelism is the basis for solving ever harder and ever bigger problems.

Although there is no clear consensus on a definition of scalability [NA91, Hil92], the intuitive notion of a scalable system is one whose computational power can be increased through the replication of a basic computing element, such as a processor paired with some memory. Attempts to quantitatively define system scalability usually include some measure of application speedups over uniprocessor run times, but such measures are themselves controversial [SHG93, KG91].

The Killer Micro

In the last decade the microprocessor has taken over the computer industry to become the heart of practically every system from personal computers (PCs) to supercomputers [HP96]. This trend is best described by the popular phrase "attack of the killer micros" [Bro89]. As a product, the microprocessor has been used most successfully

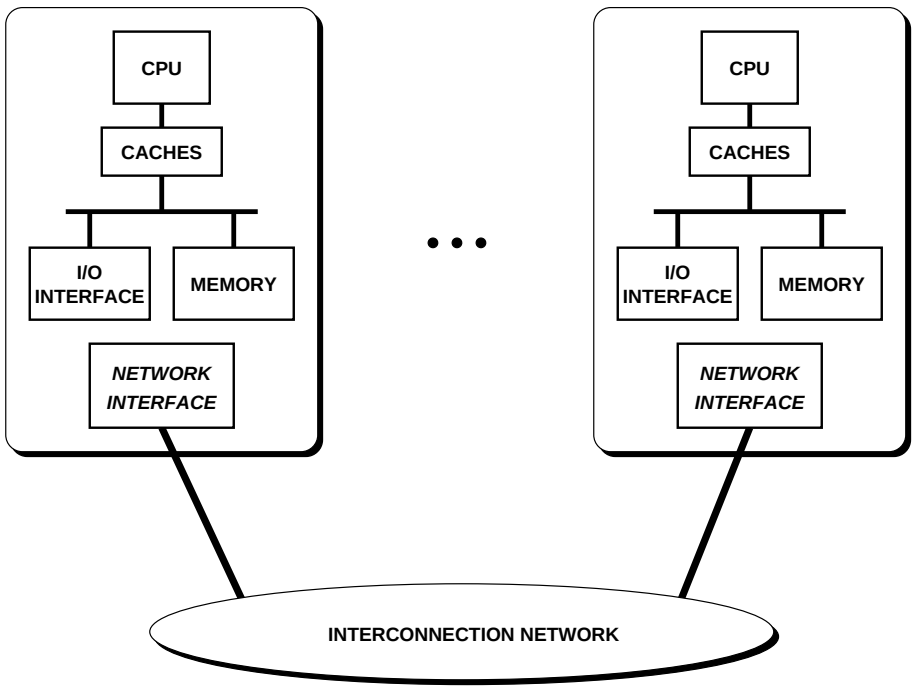


Figure 1: The basic architecture of a distributed memory parallel processor.

in PCs, and the huge PC market, in turn, is fueling continued improvements in microprocessor performance, to the benefit of the entire industry.

The Convergence of Scalable Architectures

The overwhelming commercial success of the PC has created gigantic specialized industries devoted to producing each of its individual pieces with the highest performance at the lowest price. Some of the pieces, like memory, are improved primarily through innovations in implementation technology. Others, like the CPU, benefit from architectural innovations as well as improvements in implementation technology [HP96].

In either case, the building blocks of PCs are typically fast, relatively inexpensive, and readily available. In the increasingly competitive market of scalable parallel systems, the advantage of utilizing these building blocks has not been overlooked. In

fact, one could say that scalable parallel architectures are converging on the general framework of a distributed memory parallel processor (DMPP) as shown in Figure 1.

The DMPP architecture basically consists of a group of interconnected compute nodes, each typically consisting of a CPU with caches, memory, and a network interface. Common variations on the basic architecture include nodes with multiple CPUs (symmetric multiprocessor nodes), and various creative connections between the network interface and the other node components (hence the vagueness in the diagram).

In terms of *physical* scalability, the DMPP architecture has a clear advantage over shared memory architectures due to the bandwidth limits of a shared memory. Writing parallel programs, however, is generally considered to be much easier using shared memory. These two influences have led to the emergence of distributed shared memory (DSM) systems, which are designed to provide applications with the illusion of a single, shared memory on a DMPP using hardware techniques [LLG+92], software techniques [Li86], or a combination of the two [ABC+95]. Because of its generality, Figure 1 does not show specific hardware support for DSM; this is often included within the network interface.

Common examples of DMPPs are multicomputers (tightly coupled) and distributed systems (loosely coupled). Node architectures range from the fully custom designs of multicomputers to the stand-alone, commodity workstations used in distributed systems. The interconnection networks can be custom designs or commodity products such as local area networks (LANs).

In terms of cost effectiveness, an extremely attractive commodity product in the PC market is the PC itself. Networks of PCs have the potential to provide more "bang for the buck" than any other DMPP architecture [ACP95], especially since improvements in PC designs have moved beyond the microprocessors themselves. More than ever before, engineers appreciate the degree to which the memory and I/O subsystems contribute to overall performance.

A DMPP's performance, however, is not just the sum of its parts. That is, the performance of interprocessor communication plays a major role—often *the* major role—in the speedup attainable with a DMPP.

The Limits of Communication

Whether written for a scalable DMPP with message passing or shared memory, a program can realize the full power of the system only when it is subdivided such that all of the processors are doing useful work all of the time. With the exception of a handful of simple computations, this ideal situation cannot be achieved because of two characteristics that are the bane of scalable systems: the latency and the bandwidth of inter-processor communication.

In general, a parallel computation requires the frequent exchange of data among processors. The latency of data transfer and the amount of available bandwidth affect the delay associated with this exchange. A computation is said to "tolerate latency" if it can perform useful work while data transfers are taking place. In general, the more latency a computation can tolerate, the greater its speedup on a scalable system will be.

Unfortunately, computations tend to become less latency tolerant as systems scale because of the increased amount of interprocessor communication. Because every data transfer includes software overhead, the increase in communication leads to a decrease in processor utilization for computation. This effect is exacerbated by the fact that physical limitations cause worst-case latencies to increase and worst-case bandwidths to decrease as systems are scaled up.

When a computation cannot tolerate the system's latency, its speedup decreases because processors waste time either sending data or waiting for its arrival. Therefore, in the absence of software with unlimited latency tolerance, communication performance remains a critical determinant of overall system performance.

The Communication Performance Gap

Probably the best reason for building multicomputers instead of networks of PCs has been the superior communication performance attainable with custom interconnection networks. As an illustration of this "communication performance gap" between custom and commodity interconnects, consider a hypothetical parallel program in which each node sends ten million 100-byte messages. Table 1 shows typical communication

characteristics for a LAN-based DMPP and a multicomputer, and compares the time spent passing the messages of the hypothetical program for each architecture. The total communication time is broken down into the message overhead[1] and the time spent actually transmitting message data[2].

Architecture	Latency of a single message	Peak bandwidth	Total communication time	
			Overhead	Data
LAN-based	300 μs	20 MB/s	2950 secs	50 secs
Multicomputer	50 μs	150 MB/s	493 secs	7 secs

Table 1: Illustration of the communication performance gap between LAN-based DMPPs and multicomputers.

A comparison of the last two columns of Table 1 shows the dramatic difference between the time spent *starting* small messages and the time spent transferring the actual message data. Obviously, high bandwidth, which primarily effects the last column of Table 1, is not much help in reducing the communication performance gap in this example. In fact, if the multicomputer were reduced to a bandwidth of 20 Mbytes/sec, its total communication time would increase by only about 9% (keeping the overhead constant). Message overhead, however, has a dramatic effect on the total communication time and is primarily responsible for the communication performance gap between these two architectures.

This example is meant to highlight the communication performance gap for small messages. Clearly, a larger message size would increase the effect of the data transmission time on the total communication time, increasing the importance of transfer bandwidth. Small messages, however, are quite common in parallel applications. For instance in [CHKM93], eight scientific applications were studied, and their communication characteristics measured. Of all the messages transferred by all eight applications, it was found that 78% were under 1K in size and 54% were under 128 bytes in size.

[1] $Overhead = (Message\ latency \times Number\ of\ messages) - Data\ time$

[2] $Data\ Time = (Message\ size \times Number\ of\ messages) \div Peak\ bandwidth$

Closing the Gap

In the absence of a communication technology whose latency is independent of physical distance, the communication performance gap between tightly-coupled systems and distributed systems will continue to exist. Its extent, however, is shrinking dramatically, thanks to a migration of technology from multicomputer interconnects to LANs. New products such as Myricom's Myrinet [BCF+95] and DEC's Memory Channel [Gil95] reflect this trend. For instance, an 8-port Myrinet switch has a latency of just 550 ns and a single channel bandwidth of 80 Mbytes/sec. Even ATM switches with sub-microsecond latencies and high bandwidths are becoming available [DEIK92].

In fact, interconnect latency alone is no longer the enemy of high performance communication. The principal culprit is the network interface; specifically, the high overhead of communication software. Much of this overhead can be attributed to the fact that communication-related tasks such as enforcing protection, managing message buffers, and controlling network interface hardware are performed for every individual data transfer.

This dissertation describes how network interfaces can bridge the communication performance gap by providing appropriate support for communication software with extremely low data transfer overhead. Such network interfaces can be used to build scalable, high-performance DMPPs out of low-cost, commodity components.

1.2 Main Ideas

There are four main ideas in this dissertation:

1. providing applications with protected virtual memory-mapped communication,
2. separating data destination (and hence protection) from data movement,
3. separating control transfer from data transfer, and
4. overlapping communication with computation.

Virtual memory-mapped communication is an efficient software model for specifying data transfers between regions of virtual memory. It provides protected, user-level data transfer so that multiple processes and users can share a single DMPP and communicate efficiently. This mechanism eliminates the costly overhead of a system call from the critical path of communication, yet it allows applications to transfer data directly at user level without sacrificing protection. In addition, virtual memory-mapped communication can be used in conjunction with commodity operating systems.

The separation of data destination from data movement is motivated by the fairly static communication patterns of many multicomputer applications [BLA+94]. The virtual memory-mapped communication model, described in this dissertation, allows the destination of a data transfer to be specified independently of the transfer operation itself. This greatly reduces the overhead of the transfer because such high-overhead tasks as protection checking and buffer management are associated with the destination specification.

The separation of data and control transfer is useful because many data transfers do not require a control transfer and its associated overhead [TLL94]. This dissertation describes mechanisms for transferring data directly to destination memory without the need for a receive interrupt or a receive system call. Because control transfer should be a synchronization option rather than a requirement, every data transfer can include an optional notification, which causes an interrupt at the receiver.

Overlapping communication with computation is desirable because it can increase the amount of parallel computation performed by a DMPP. This dissertation describes mechanisms for transferring data using direct memory access (DMA), and a mechanism for initiating such transfers from user level with very little software overhead. Because of the low software overhead, this "user-level DMA" mechanism minimizes CPU involvement in communication, freeing CPUs for computation.

1.3 SHRIMP

A SHRIMP (Scalable, High-performance, Really Inexpensive Multi-Processor[3]) system consists of commodity PC "nodes" connected by a high-speed interconnection network. The nodes are not physically distributed, but remain a short distance from the interconnect, forming, in effect, a closely-coupled network of PCs. The key components in the system are the custom SHRIMP network interfaces, which connect the nodes to the network.

The SHRIMP network interface[4] supports both message passing and eager sharing using Virtual Memory-Mapped Communication (VMMC), described in Chapter 2. The VMMC model [DIFL96] allows processes to transfer data directly between their virtual memory address spaces over the interconnection network. With network interface hardware support, VMMC achieves extremely low-overhead message transfer, allowing applications to utilize most of the raw latency and bandwidth of the underlying communication hardware.

Some existing network interface designs support low-overhead communication through user-level message passing, but they either prohibit multiprogramming, or constrain how processes are scheduled. SHRIMP supports the general multiprogramming of commodity operating systems while providing low-overhead, user-level communication.

A SHRIMP system is basically a server, and is expected to operate in a cleanroom environment, like a multicomputer. Because a closely-coupled system in such an environment is expected to have a low error rate, SHRIMP provides error detection, rather than error correction, in hardware. Software is given the optional responsibility for providing fault tolerance.

[3] Acronym coined by Michael Carey, University of Wisconsin, Madison

[4] The network interface discussed in this dissertation was referred to as the SHRIMP-II network interface in some prior publications [BDF+94, BDF+95, DIFL96].

1.4 Main Results

The principal contribution of this dissertation is to describe how the four main ideas of Section 1.2 can be combined to create DMPP network interfaces that provide low-overhead, protected, user-level communication. This assertion is demonstrated via a working implementation of a SHRIMP system consisting of 60 MHz, Pentium-based PC nodes, an Intel interconnection network, and custom SHRIMP network interface boards.

Actual measurements of the SHRIMP system are used to demonstrate very low communication overhead for various transfer sizes. A minimum-sized packet (32 bytes with a 4-byte payload) was sent back and forth between two nodes and achieved a roundtrip time of just 7.42 μs. Analyses revealed the hardware component of this measurement to be about 5 μs, leaving just 2.42 μs of program and communication overhead.

Bandwidth measurements also demonstrate the advantage of low-overhead communication. These measurements revealed that the bandwidth asymptotically approached a maximum, as expected. When this maximum was compared to the bandwidth of small transfers, it was found that a 300-byte transfer achieves 50% of the peak bandwidth, and a 4-Kbyte transfer reaches 98%. Chapter 5 explains the details of these measurements and describes several more experiments performed with the prototype SHRIMP system.

The prototype system is a very cost-effective implementation of the main ideas of this dissertation; however, the ideas are applicable to virtually any DMPP network interface.

1.5 Related Work

1.5.1 Traditional DMA-Based Network Interfaces

Traditional network interface hardware is based on DMA data transfer. Recent examples include the NCUBE [Pal88], iPSC/2 [Arl88] and iPSC/860 [Int91a]. The network interface designs of these machines are very similar. An application sends

messages by making operating system calls that initiate DMA data transfers from memory to the network interface. The network interface packetizes the data and injects it into the network. At the receiving node, the network interface initiates an incoming DMA data transfer to memory, and interrupts the local processor when the transfer has finished so that it can dispatch the arrived message. Finally, the application makes a system call to receive the message.

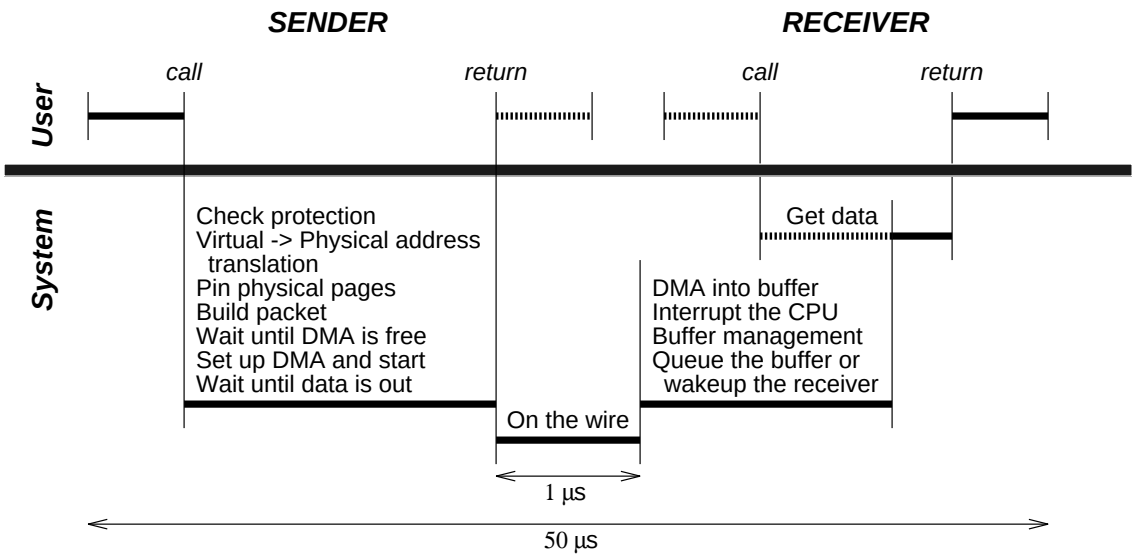


Figure 2: Message passing latency on the Intel Paragon multicomputer.

A network interface based on this approach is simple and can be used with commodity PCs or workstations. The main disadvantage is that the software overhead due to the system calls, DMA configuration, and interrupt is usually thousands of CPU cycles, with the best implementation [ECGS92] still requiring over 100 CPU cycles.

For example, Figure 2 shows the steps involved in a small message transfer on the Intel Paragon multicomputer. Of the total transfer time, only 2% is actually spent on the network. The rest is attributable to network interface hardware and software overheads, with software dominating significantly.

1.5.2 Communication Coprocessors

An interesting means for increasing communication performance is to add a programmable processor to each node, and devote it to communication. The "compute" and "message" processors can work in parallel to overlap communication and computation. In addition, the message processor can poll for message reception, eliminating receive interrupt overhead. Systems have been proposed and built with both commodity and custom communication co-processors, with custom designs far outnumbering commodity.

An early example of the commodity coprocessor approach is described in [RSV87], which proposed a node architecture for a distributed system. The idea was to offload communication-specific functions of the node operating system to a second processor, and allow the processors to communicate through queues in shared memory. Although the proposed architecture called for commodity processors, it required custom "smart bus" and "smart shared memory" for efficient sharing between the processors. Simulation studies of the proposed system are presented in [RSV87], and it is not known whether a real system was ever built.

A recent example of the commodity coprocessor approach is the Intel Paragon node [Int91b], which is a symmetric multiprocessor design where one processor is used for message handling. Bulk data transfer between node memory and the network is performed by two simple DMA controllers, which must be configured and managed by the message processor. Nodes are available with either two or three processors, and user-level interprocessor communication is performed through the shared-memory.

Several custom coprocessor designs have been proposed specifically for message passing. An early example is the message passing coprocessor (MPC) described by Hsu and Banarjee [HB90]. The MPC is microprogrammable and has its own local memory for data structure and message header storage.

Another example of a message-passing coprocessor is the Elan, used in the design of the Meiko CS-2 [HM93] node. The Elan shares memory with a commodity Sparc microprocessor and provides user-level message passing. Message protocols are stored as threads in main memory and executed by the Elan in response to user-level transfer

requests. The Elan contains its own TLB and page tables, which are kept consistent by system software.

Many coprocessors have also been designed to support communication models other than message passing. For example, the MIT *T architecture is based on a node with three custom processors [NPA92]. *T is designed to support both dynamic dataflow computation and code developed for more standard massively parallel processor (MPP) systems.

A recent example of a coprocessor that supports various communication models is the Stanford FLASH multiprocessor [KOH+94], which features a node containing a commodity microprocessor and the custom MAGIC "node controller" chip. The MAGIC chip integrates the memory controller, I/O controller, and network interface of the node, and includes a programmable protocol processor to provide both cache coherent distributed shared memory and message passing [HGDG94]. Although the centralized functionality and programmability of the MAGIC chip results in a very flexible communication architecture, its complexity results in a long development cycle, possibly reducing the performance of FLASH in comparison to simpler, state-of-the-art designs. At the time of this writing, the implementation of FLASH is not yet complete.

The Typhoon node architecture [RLW94] is very similar to FLASH, and the project has the same goal: integrating shared memory and message passing in a single design. The Typhoon design described in [RLW94] specifies the architecture of a custom, integrated network interface processor (NP) that is situated between the network and the shared node bus.

Complexity realizations have led to the design of Typhoon-0 [RPW96], which handles communication using a commodity network interface, a commodity microprocessor, and some custom hardware. The network interface provides message passing using memory-mapped network interface FIFOs, and DSM support is provided by a combination of custom fine-grain access control hardware and the commodity protocol processor. In order to improve the performance of DSM, a follow-on effort is underway to integrate the fine-grain access control hardware and the network interface into a single design [RPW96].

All of these designs dedicate a processor for message handling in order to eliminate the system calls for transferring data, but this does not reduce the overhead of the software protocol on that processor. Furthermore, the additional processor (or two) increases the complexity and cost of each node, especially when the processor is a custom design. When the additional processor is commodity, its cost is easier to justify if it can be used for general computing as well as communication.

1.5.3 Processor-Integrated Network Interfaces

Several projects have taken the approach of lowering communication overhead by integrating the network interface with the CPU, and providing a mechanism for directly writing and reading the network interface FIFOs in order to queue and de-queue data respectively.

For example, iWarp [BCC+90] was designed to perform fast, systolic computation using communication FIFOs mapped to processor registers. Each node's output FIFO is mapped to a single register and its input FIFO is mapped to another register. Writing and reading these registers queues and de-queues message data, respectively. There is no protection mechanism and applications are required to explicitly drain their input FIFOs, so iWarp is not well suited for general-purpose programming.

Another well known example of a processor-integrated network interface is the MIT J-Machine [NWD93]. The J-Machine processor is single-threaded and messages are injected into the output FIFO through a series of individual "send" instructions. Each message begins with the address of a message handler that is run in order to de-queue the message on the receiving node. While this mechanism can be good for very fine-grained sharing, a good deal of software overhead is required for larger transfers. In addition, the J-Machine does not support multiprogramming because there is no protection mechanism.

A similar network interface architecture, also developed at MIT, is described in [HJ92]. In this design, a message is composed in processor registers and launched into an output FIFO with a single instruction. Received messages are provided in a separate set of registers and are loaded from a message input FIFO by a single

instruction. Protection is provided by attaching the process identifier of the running process to each message, and directly delivering only those messages destined for the running process. All other messages are spooled to memory by hardware, or buffered in memory by the operating system (requiring a context switch). Therefore, this scheme seems to require gang scheduling[5] for efficiency.

A final example of the processor-integrated network interface approach is the MIT M-machine multicomputer [FKD+95], which follows in the footsteps of the J-machine. Each M-Machine node consists of a single, highly-integrated custom processor containing all the node functionality, and some external memory. Messages are composed in registers and launched with a single instruction that specifies a virtual destination address. This causes the message data to be transferred to an output FIFO, and out onto the network. Received messages are queued in an input FIFO, which is drained by a system-level message handler. Protection is provided through a virtual memory system.

The M-Machine can issue up to four instructions simultaneously, each consisting of up to three operations. Because the instructions can come from different threads, it is difficult to predict what the overhead of servicing the communication subsystem will be. In general, however, computing resources are required.

To summarize, the principal drawbacks of the processor-integrated network interface approach are complexity and lack of general protection mechanisms. Machines using this approach require custom processors, which are costly and time-consuming to design and build. In addition, these machines are not well suited for general-purpose use because they do not support multiprogramming efficiently.

1.5.4 Memory-Mapped Network Interfaces

An effective method for providing protected, user-level communication is to memory-map the network interface hardware into a node's address space. Designs using this approach are typically similar to the register-mapped network interfaces in that

[5]Timesharing such that all the processors are executing the same user task at the same time.

writing and reading special memory locations causes the queueing and de-queueing of data to network interface FIFOs, respectively.

For example, the Connection Machine CM-5 [LAD+92] provides access to network interface FIFOs through physical memory locations, and allows user-level applications to access them directly through virtual-to-physical memory mappings. Protection is provided by these mappings, but the scheme only provides protection between subsets of nodes. Within a subset, gang scheduling is used to provide multiprogramming.

The CM-5 network interface is very basic in that a node's processor must explicitly write each word of a message to the output FIFO, and explicitly read each received word from the input FIFO. Therefore, the overhead for message passing is high, even though message handling can be done at user level. For example, the best active message latency on the CM-5 is about 3.3 μs, with software overhead on the order of 100 instructions [ECGS92].

A more recent example of a similar memory-mapped network interface architecture is that of Typhoon [RLW94, RPW96]. Like the CM-5, Typhoon message passing is done by reading and writing 64-bit-wide, memory-mapped hardware queues.

Several systems use memory-mapped network interfaces to support user-level remote memory references. Examples include the Cray T3D [Res93]—an MPP—and Telegraphos [MK96]—a workstation cluster. The basic idea is to allow access to remote physical memory by performing ordinary loads and stores to the memory-mapped network interfaces. The network interfaces maintain appropriate "mappings" between local and remote physical addresses.

1.5.5 Eager-Sharing Network Interfaces

Eager sharing is a mechanism for maintaining consistency among copies of shared data by forwarding updates to the data when they are generated, as opposed to demand sharing, which forwards updates when the data is accessed.

Many of the eager-sharing ideas presented in this dissertation grew out of the Princeton Pipelined RAM (PRAM) project [LS88]. Every host in a PRAM network has a small portion of its physical memory mapped to equivalent portions in all other

hosts. When any host updates its PRAM memory, the update is propagated to all other PRAM memories in the network, with no coherence enforced by the hardware. The drawbacks of this simple approach are a potentially large amount of network traffic, and a very limited shared space.

Several DSM architectures use a page-based, eager-sharing mechanism to propagate updates. The general idea is to allow applications to specify which memory pages are shared, and provide hardware that automatically propagates updates to all copies of shared pages. This is a straightforward extension of the PRAM mechanism described above.

An early example of this architecture is Merlin [WM89], where a host's local physical pages could be individually shared by mapping them to "global virtual address" space pages. Whenever two or more hosts mapped to the same global virtual page, they would eager-share updates to that page. Merlin lacked support for cache coherence, which was addressed in the design of its successor, SESAME [WHL92].

The Princeton PRAM model was extended in [San91] to include support for page-based sharing using a network "virtual shared memory" address space to which hosts could map individual pages of their physical memory. Like Merlin, physical pages mapped to the same shared memory page eager-share updates. This model is very flexible, but complex, and it was never implemented.

All of these network virtual address schemes share the drawback of hardware complexity, required to translate from network addresses to physical addresses at receiving nodes. In order to simplify the hardware required to implement eager-shared DSM updates, the Galactica Net [JJT93] project, and its successor, Jacquard [JPCI94], use a mechanism to link physical pages into "virtual sharing rings". Any write to a shared page is propagated to all the other pages on the ring in serial order. This mechanism does not require hardware translation between local physical addresses and global network addresses, but relies on a simple table of pointers to remote physical pages used to form a "linked list" between pages. The hardware update mechanism is combined with release consistency to implement a consistent shared memory model.

A common drawback of all of these DSM projects is that they do not provide a mechanism for high-bandwidth, low-overhead block data transfer needed for message passing.

1.5.6 Direct Memory Data Transfer

In order to reduce the overhead of receiving a message, several projects have taken the approach of requiring a sending node to specify the message destination in the receiver's memory, and having the receiver's network interface transfer the data directly to memory.

Hamlyn [Wil92] supports "sender-specified", user-level message passing by establishing a level of address indirection through "message areas" from which, or to which, data can be transferred. The relationship between message areas and the physical pages that comprise them is maintained by network interface mapping tables, which are stored in main memory. The mapping tables are configured by the operating system and utilized directly by the network interface hardware.

Hamlyn provides a DMA interface for large transfers, and a direct interface, which allows small messages and their headers to be written to a memory-mapped FIFO for low-latency transfer. In order to specify a DMA transfer, a user-level application must build a message header that specifies the offsets within source and destination message areas, and the size of the transfer. At the time of this writing, the Hamlyn ideas are being implemented on modified Myricom network interfaces [BJMW95].

A similar idea is explored in the D^2ART architectures, which provide a "direct deposit" model for communication over ATM LANs [Osb94]. The proposed hardware required to implement this model appears to be quite complex.

The SHRIMP network interface, described in this dissertation, provides a DMA interface that is similar to Hamlyn, but uses the existing virtual memory translation hardware to provide a level of address indirection on the sending node for protection. In addition, sender-specified message destinations are supported by allowing the sender to specify the physical destination address directly. Finally, SHRIMP provides

a low-latency, direct data transfer mechanism that does not require the explicit generation of headers.

1.5.7 Remote Memory Reference

The remote memory reference model was originated by Spector [Spe82] and demonstrated on a network of Xerox Altos. A remote reference causes a single remote operation to be performed by a remotely located process, and optionally, a value returned. Recently, it has been revived as a model to program message-passing primitives for high performance [TLL93, ECGS92, vEBBV96]. All of these implementations require explicit CPU action or new instructions to initiate communication and require CPU intervention to receive messages.

The Cray T3E and its predecessor, the T3D [Res93], support simple load and store remote memory references in hardware. These machines require a very fast network and a large amount of custom hardware design because they do not cache remote data. It is not clear whether the overhead from not caching degrades the performance of message-passing applications.

Telegraphos [MK96] supports user-level remote memory references on a workstation cluster using memory-mapped network interfaces. Each network interface maintains a portion of a node's physical address space that can be "mapped" to remote physical memory on a per-page basis. Remote references are performed by writing and reading mapped pages through ordinary virtual-to-physical mappings. Because Telegraphos uses a LAN interconnect, the latencies of remote references will be even higher then those of the T3E.

1.5.8 Implicit Multiprocessor Communication

Distributed-memory machines that only provide support for DSM generate implicit data transfers in order to keep a simulated shared memory consistent. They do not provide explicit communication mechanisms such as message passing or remote

memory reference. Typical examples include the Tera Multi-Threaded Architecture [ACC+90], the MIT Alewife multiprocessor [ACJ+92], the Stanford Dash multiprocessor [LLG+92], Memnet [Del88], the PLUS system [BR90], and the Kendall Square Research KSR1 [Ken92]. Because the communication mechanism described in this dissertation is basically explicit, it is difficult to compare it to these architectures.

1.5.9 Address Mapping Techniques

Several systems use address mapping techniques to provide protected, user-level access to hardware. For example, this dissertation describes a method for initiating DMA data transfers through specially mapped memory.

The AP1000 multicomputer uses special memory-mapped regions to issue "line send" commands to the network interface hardware [SHI92]. This mechanism allows a user-level program to specify the source address of a transfer, but not the size or destination address. The size is hard-wired to that of a cache line, and the destination address is a fixed circular buffer in the receiver's memory. In addition, the transfer is not a DMA, since the CPU stalls while the transfer is occurring.

Address mapping techniques are used to access the vector units on the CM-5 [Thi92]. A program running on the main processor of a computing node communicates command arguments to its four vector co-processors by accessing a special region of memory. The vector co-processors decode this region and utilize the data as a command and the address as an operand location specification.

The Flash system [HGDG94] uses a technique similar to the one described in this dissertation to allow user programs to specify data transfer source and destination memory addresses to communication hardware. In the context of FLASH, this memory is called "shadow memory", while SHRIMP refers to it as "proxy memory".

Chapter 2

Virtual Memory-Mapped Communication

Low-overhead SHRIMP communication results from a combination of hardware and software techniques. The hardware techniques were used in the design of the SHRIMP network interface to produce a number of mechanisms for communicating between physical memory pages, as described in Chapter 4. The software techniques have been incorporated into Virtual Memory-Mapped Communication (VMMC), an efficient model for specifying data transfers between regions of virtual memory, and the subject of this chapter. Although VMMC is designed to take advantage of the SHRIMP hardware mechanisms, it can be used with a wide range of network interfaces, as described in Chapter 3.

VMMC was developed out of the need for a basic DMPP communication mechanism with extremely low latency and high bandwidth. This is achieved by allowing applications to transfer data directly between two virtual memory address spaces over the network. VMMC can either be used directly by applications, or used to support more extensive models of communication such as message passing, shared memory, and remote procedure call.

The basic VMMC mechanism is used to create mappings between equally sized regions of virtual address spaces as shown in Figure 3. Each of the processes shown here has its virtual address space divided into two subspaces, one depicted with a

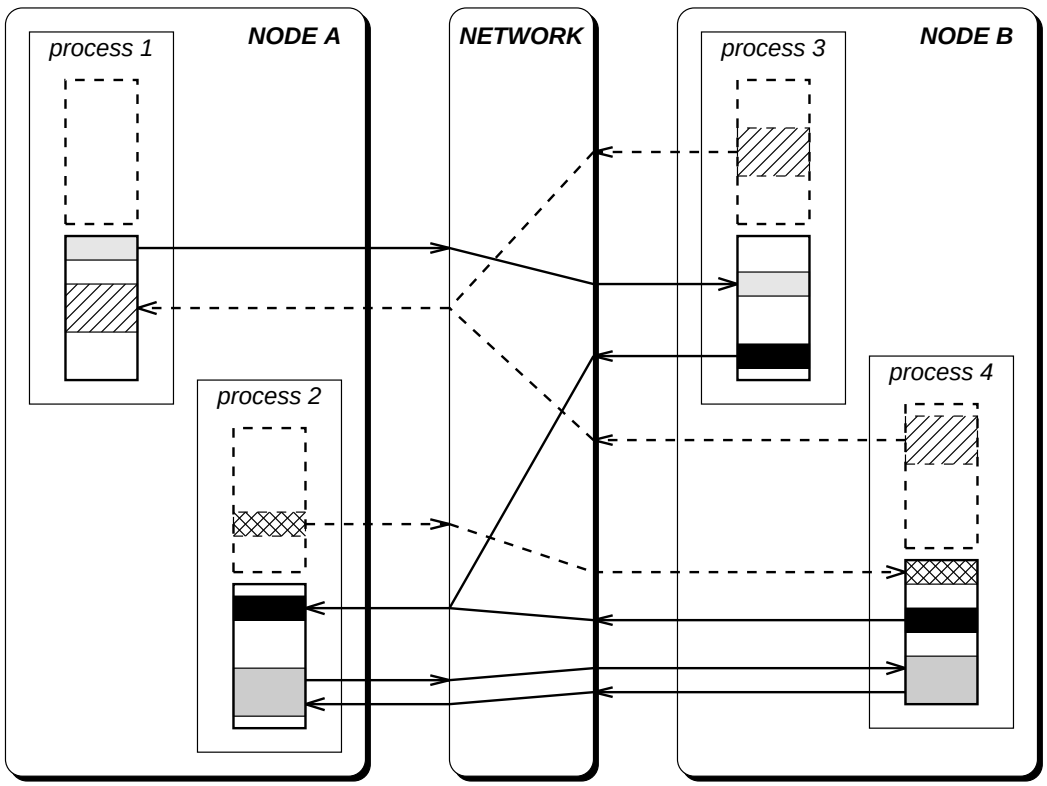


Figure 3: Some virtual memory mappings between processes on two nodes.

solid box and the other with a dashed box. The subspace depicted with a solid box is the portion of the process' virtual address space that is backed by real, physical memory, and commonly referred to as the virtual memory of the process. The other subspace is called virtual proxy memory and will be described in Section 2.2.

A process can "map" a region of its virtual memory or its virtual proxy memory to a region of another processes' virtual memory for the purpose of transferring data to that memory. Mappings from virtual memory to virtual memory (depicted by solid arrows) are used to eager-share data, while mappings from virtual proxy memory to virtual memory (depicted by dashed arrows) are used to transfer data through explicit send operations.

As Figure 3 shows, many-to-one mappings (processes 3 and 4 to process 1) can be created because a process specifies the "destination" of its mappings. In addition,

bidirectional mappings (processes 2 and 4) can be created between virtual memory regions by specifying two inverse mappings.

VMMC has several advantages over the traditional, kernel dispatch-based message passing. One of the main advantages is that it allows applications to perform low overhead communication since it can be implemented such that data can move between user processes without context switching and message dispatching.

Another main advantage of VMMC is that it moves the management of memory buffers to user level. Applications or libraries can manage their communication buffers directly without having to pay the expensive overhead of unnecessary context switches and protection boundary crossings in the common cases. Recent studies and analyses indicate that moving communication buffer management out of the kernel to the user level can greatly reduce the software overhead of message passing. By using a compiled, application-tailored runtime library, the latency of multicomputer message passing can be improved by about 30% [Fel93].

Finally, VMMC takes advantage of the protection provided by virtual memory systems. Since mappings are established at the virtual memory level, virtual address translation hardware guarantees that an application can only use mappings created by itself. This eliminates the per-message software protection checking found in traditional message passing implementations.

Virtual memory-mapped communication can be performed entirely at user level with sufficient hardware support, but the creation of virtual memory mappings is always done by the operating system in order to provide protection.

2.1 The VMMC Model

The VMMC model described in this section, and the application programming interface described in the next section are primarily the work of Cezary Dubnicki, Liviu Iftode, Edward Felten, Kai Li, and the author [DIFL96]. It should be understood that the VMMC model is continuing to evolve as the SHRIMP project progresses. The description provided here pertains to the model as implemented on the prototype

SHRIMP system. Features that have been considered for future implementations, such as a remote fetch operation, are not included in the following description.

2.1.1 Import-Export Mappings

The VMMC model defines a *receive buffer* as a region of virtual address space to which data can be received. A user-level process creates a receive buffer by *exporting* a portion of its address space, together with a set of permissions to define access rights for the buffer. In order to send data to a receive buffer, a user-level process *imports* the buffer, and succeeds if allowed by the permissions established by the exporter.

Transfer of data in VMMC can only occur after a successfully exported receive buffer has been successfully imported by a sender, creating an *import-export mapping*. The need for such a mapping arises for several reasons:

- *Protection*: In order to provide protection, no process can be granted the right to arbitrarily transfer data into another process's address space. A VMMC import-export mapping restricts transfer to a pre-approved region of address space only.

- *Allocation of receive buffers*: Upon export, memory is committed for receive buffers so that the data send operation does not need to check for the availability of receive buffer space.

- *Identification of receive buffer memory*: Because of the export operation, the system can keep track of memory that is writable by a remote sender.

- *Caching of destination information by the sender*: After a successful import, a sender can succinctly name a receive buffer from user level, allowing for low-overhead transfer initiation.

Establishing an import-export mapping requires a trusted third party such as the operating system kernel or a daemon[1] in order to enforce protection, modify process

[1]A daemon is a background process that either waits for some event to occur, or waits to perform some specified task on a periodic basis [Ste90].

page tables if necessary, and configure networking hardware if necessary. Creation of a mapping can be relatively expensive, but should occur infrequently since only one import is required for each receive buffer.

2.1.2 Transfer Strategies

The VMMC model defines two user-level data transfer strategies: *deliberate update* and *automatic update*. Deliberate update derives its name from the fact that a deliberate *send* command is used to update an imported receive buffer with data from a process's local address space. Automatic update is a form of eager-sharing communication. In order to use it, a sender *binds* a portion of its address space to an imported receive buffer, creating an *automatic update binding* between the local and remote memories. All writes performed to the local memory are automatically performed to the remote memory as well, eliminating the need for an explicit send operation.

An important distinction between these two transfer strategies is that under automatic update, local memory is bound to a receive buffer at the time a mapping is created, while under deliberate update the binding does not occur until an explicit *send* command is issued, and the binding only lasts for the duration of the transfer. Automatic update is optimized for low latency, and deliberate update is optimized for flexibility and high bandwidth.

VMMC guarantees the in-order, reliable delivery of deliberate update transfers. It also guarantees the in-order, reliable delivery of automatic update transfers. However, it is possible to perform an automatic update transfer before an in-progress deliberate update transfer completes. One benefit of this is that short automatic updates can overtake long deliberate updates for expeditious delivery.

The VMMC model does not include any buffer management since data is transferred directly between user-level address spaces. This gives applications the freedom to utilize as little buffering and copying as needed. The model directly supports zero-copy protocols because both the send and receive buffers can be known at the time of a transfer initiation.

The VMMC model assumes that receive buffer addresses are specified by the sender, and received data is transferred directly to memory. Hence, there is no explicit receive operation. CPU involvement in receiving data can be as little as checking a flag, although a hardware notification mechanism is also supported.

2.1.3 Notifications

A VMMC *notification* is used to transfer control to a receiving process, or to notify the receiving process about external events. It consists of a message transfer followed by an invocation of a user-specified, user-level handler function. The receiving process can associate a separate handler function with each exported buffer, and notifications only take effect when a handler has been specified.

Notifications are similar to Unix signals in that they can be blocked and unblocked, they can be accepted or discarded (on a per-buffer basis), and a process can be suspended until a particular notification arrives. Unlike signals, however, notifications are queued when blocked.

2.2 VMMC Application Programming Interface

The VMMC model can be applied to multiprocessors of all sorts, ranging from shared-memory SMPs to networks of commodity computers. However, the description of the VMMC application programming interface (API) provided here is tailored to the SHRIMP system prototype and may not be sufficient to meet the needs of all systems. For instance, security and fault-tolerance were not strongly emphasized for this first implementation.

2.2.1 Destination Proxy Space

The API for deliberate update is based upon the concept of *destination proxy space*, which is a region of a sender's address space reserved for mappings to imported receive buffers. This region is usually a portion of the virtual address space, although it can be a completely separate space depending on the implementation of VMMC.

Destination proxy space is not backed by any real physical memory, and its addresses do not refer to code or data. Instead, they are used to specify the destinations of data transfers.

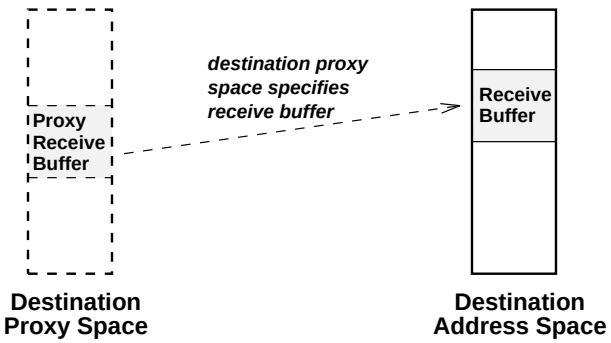


Figure 4: A receive buffer imported as a proxy receive buffer.

After a successful import, a representation of the imported receive buffer is created in the sender's destination proxy space, as shown in Figure 4. This *proxy receive buffer* is the same size as the imported receive buffer, and a reference to this region can be translated by VMMC to a data destination specification consisting of a node, process, receive buffer, and buffer offset.

2.2.2 Export

In order to establish an import-export mapping, a receiver must export a region of its address space with the following operation:

```
Export(mapId, recvBuf, nBytes, permissions)
```

where `recvBuf` is the virtual address of the start of the receive buffer, and `nBytes` is its size. The `permissions` argument describes a set of processes that are allowed to import the buffer, and `mapId` is a user-defined integer identifier that is used by importers to refer to the buffer.

2.2.3 Import

To complete an export-import mapping, a sender imports a receive buffer into its destination proxy space with the following operation:

```
Import(mapId, exporter, &impStart, &nBytes)
```

where `exporter` uniquely identifies the exporting process, and `mapId` matches the identifier of the exported buffer. A successful import returns the starting proxy address of the imported receive buffer in `impStart`, and its size in `nBytes`. An import is matched with an export if and only if the following conditions are met:

- Both specify the same `mapId`.
- `exporter` identifies a process that performed `Export`.
- `permissions` allows the process attempting `Import` to do so.

The standard `Import` operation blocks until a matching `Export` is done, which can easily lead to deadlock if imports are attempted in the wrong order. To avoid deadlock and provide better performance, the VMMC API includes a non-blocking variant of `Import` that takes the same arguments, but returns an integer *handle* identifying the import request. This handle can be used later to inquire about the status of the import.

Deadlock can be avoided if all exports are issued, followed by all imports in the blocking form. Alternatively, all imports can be issued in the non-blocking form, followed by all exports, followed by a wait for all import completions.

2.2.4 Deliberate Update

Deliberate update is an explicit command to transfer data from any readable region of a sender's memory to a previously imported receive buffer. The basic deliberate update send operation has the following form:

```
SendMsg(srcAddr, destAddr, nBytes)
```

This is a request to transfer `nBytes` of data beginning at the sender's virtual address `srcAddr`, to a receive buffer beginning at the destination proxy address `destAddr`. It is the responsibility of VMMC to insure that data is not transferred outside of the selected receive buffer.

For example, the request `SendMsg(Srcaddr, ImpAddr+200, 4096)` will transfer 4096 bytes starting at `SrcAddr` to the receive buffer imported to `ImpAddr` starting at offset 200, assuming the size of the receive buffer is at least 4296 bytes. If the size of the receive buffer is 4096 bytes, then only the first 3896 bytes will be transferred, and `SendMsg` will return an error.

If no error occurs, then `SendMsg` blocks until all the data has been transferred from memory to the network interface. In order to minimize the overhead of sending data, the VMMC API also provides a non-blocking variant of `SendMsg` that returns immediately with a handle identifying the request. The handle can be used later to inquire about the status of the request.

Another variant of `SendMsg` is provided to transfer data with a notification attached so that the receiver is informed of the data arrival. The VMMC API does not provide a data transfer request that informs the sender when data has been written into the receive buffer or consumed by the receiver. This requires a higher-level protocol.

Because `SendMsg` transfers data to a previously defined and allocated receive buffer, there is no need to verify buffer space on the receiver before beginning a transfer. Since user code allocates receive buffers, data can be transferred directly from its source to its final destination, avoiding expensive copy operations.

2.2.5 Automatic Update

The VMMC API supports automatic update communication by allowing a sender to bind a region of local virtual memory to a previously imported receive buffer with the following operation:

```
Bind(srcAddr, destAddr, Mode)
```

The `Bind` operation binds the local memory starting at `srcAddr` to the receive buffer starting at the proxy address `destAddr` so that all writes to the local memory are automatically propagated to the receive buffer, obviating the need for an explicit send operation. `Mode` specifies various implementation-specific characterizations of the binding such as whether combining of multiple automatic updates will be performed. The separation of `Bind` and `Import` is useful since `Bind` can be implemented as a local operation.

A variant of `Bind` is provided to transfer data with a notification attached so that the receiver is informed of the data arrival.

2.2.6 Destruction of Mappings

Both the sending and receiving processes can destroy an import-export mapping. A sender uses `Unimport` to indicate that it no longer intends to transfer data to a specified receive buffer. After `Unimport` completes, the destination proxy space allocated for the receive buffer is free to be used for another `Import`.

A sender can destroy an automatic update binding to a receive buffer using `Unbind`. Like the `Bind` operation, `Unbind` is local to the sender and does not unimport the specified receive buffer.

A receiver uses `Unexport` to indicate that it will no longer accept any data transfers into a specified receive buffer. After `Unexport` completes, all import mappings of the specified buffer have been destroyed, and any messages sent along these mappings have been delivered. For instance, if VMMC is used to implement a client/server communication model, then each client usually imports a unique receive buffer from a server. To break the connection with a given client, the server simply unexports the client's buffer. `Unexport` can also be used to free destination proxy space if it is limited, as in the case where all receive buffers need to be locked in physical memory.

Since `Unexport` may require communication with multiple senders to flush and destroy all mappings, the VMMC API provides a non-blocking variant that returns a handle to identify the request. The handle can be used later to inquire about the status of the request.

2.2.7 Notifications

Recall that a notification is like a Unix signal and is used to transfer control to a receiving process or to notify the receiving process about external events. A process indicates its desire to receive notifications for a particular receive buffer by specifying a user-level handler function when exporting the buffer.

When a message with a notification attached is received, the handler is called by the system with two arguments: the address of the last word of data transferred by the message, and the value of this word. Because notification messages are not synchronous, the message data in memory can be overwritten by subsequent messages before the notification handler is called. The word of data passed to the handler, however, is the correct last word of the notification message.

Processes can block and unblock notifications in order to insure the consistency of data structures that are modified by both the notification handler and the main thread of execution with the following two operations:

```
BlockNotifications()
status = UnblockNotifications()
```

Blocking notifications affects all the receive buffers of a process, and causes all subsequent notifications to be queued by the system rather than delivered immediately. When notifications are unblocked, they are delivered in arrival order through the normal notification handler calls.

Pairs of `BlockNotifications()` and `UnblockNotifications()` calls can be nested, with notifications being unblocked only when the outermost `UnblockNotifications()` is called. This is indicated by a positive `status` return value. The only situation where notifications cannot be unblocked is within the handler function itself, which is assumed to be non-reentrant.

A process can suspend, pending the arrival of a notification with the following operation:

```
Select(fd_set, timeout)
```

This custom select operation is identical to Unix select, but returns when any notification comes in for the process that called it, as well as the standard return conditions. The current API does not include support for process suspension pending the arrival of a specific notification or set of notifications.

2.3 A VMMC Example

As a simple motivating example of the benefits of VMMC, consider a program that repeatedly transfers data to the same destination, as is commonly done with the boundary elements of a grid computation. Figure 5 illustrates such a data transfer from a source region to a destination region.

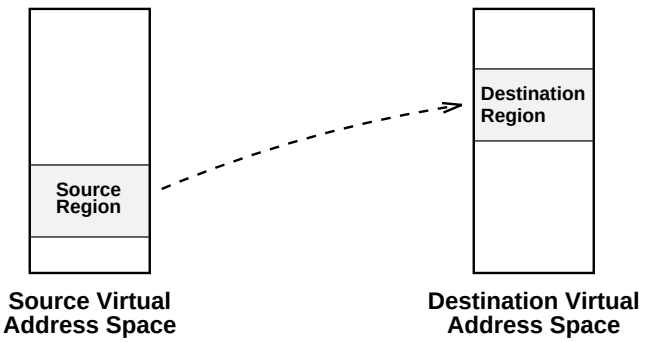


Figure 5: A simple data transfer example.

In a traditional message-passing environment such as Intel NX [Pie88], the semantics for sending and receiving data are very straightforward, but the implementation is complex and heavy-weight. Typical code to transfer the data of Figure 5 is shown in Figure 6.

The `Send` operation is a system call that verifies protection, allocates space at the receiving node, `DestNode`, and then transfers `MsgSize` bytes of data from `SrcRegion`. The `Receive` operation is a system call that either posts `RcvBuf` as the receive buffer if the message identified by `MsgId` has not arrived, or transfers it from a temporary system buffer to `RcvBuf` if it has. The receiver controls the number of bytes it is willing to receive with its `MsgSize` argument.

Sending Node:

```
...
loop:
   /* transfer the message data */
   Send(MsgId, SrcRegion, DestNode, MsgSize);
   ...
endloop:
...
```

Receiving Node:

```
...
loop:
   /* receive the message data */
   Receive(MsgId, RcvBuf, MsgSize);
   ...
endloop:
...
```

Figure 6: Basic data transfer using traditional message passing.

VMMC eliminates the software overhead of the data transfer by pre-allocating space at the destination and verifying protection *before* the actual data transfer through an import-export mapping, as shown in Figure 7.

Somewhere near the beginning of the program, the receiver exports a receive buffer and a receive flag (without notification handlers). The sender imports these data structures into its proxy space and binds the `Flag` variable to the remote receive flag for automatic update.

Within the loop, the data is transferred using `SendMsg`, which is *not* a system call, but a user-level function call to a run-time library. The synchronous `SendMsg` operation performs a very lightweight deliberate update transfer initiation to start a hardware DMA transfer, and then waits for its completion. There is no need for an explicit receive operation corresponding to the `SendMsg` operation because the message destination has already been established by the import-export mapping.

Receiving Node:

```
/* export a receive buffer */
Export(BufId, RcvBuf, BufSize, NULL, ...);

/* export a control flag */
Export(FlagId, FlagBuf, 4, NULL, ...);
...
loop:
   ...
   /* poll flag for control transfer */
   while(FlagBuf == 0) ;
   ...
endloop:
...
```

Sending Node:

```
/* complete import-export mapping to the receive buffer */
Import(BufId, DestNode, &ProxyRcvBuf, &BufSize);

/* complete import-export mapping to the flag buffer */
Import(FlagId, DestNode, &ProxyFlagBuf, &FlagSize);

/* create an automatic update binding to the remote flag */
Bind(&Flag, ProxyFlagBuf, AUenable);
...
loop:
   ...
   /* transfer some data using synchronous SendMsg */
   SendMsg(SrcRegion, ProxyRcvBuf, MsgSize);
   ...
   /* transfer control by setting the remote flag */
   Flag = 1;
   ...
endloop:
...
```

Figure 7: Basic data transfer using VMMC.

At some point in the loop code, possibly after multiple data transfers, the sender transfers control by simply setting the `Flag` variable. This causes an automatic update transfer of the new value to the remote receive flag.

Alternatively, `FlagBuf` could have been exported with a notification handler specified (instead of NULL), and the `Bind` operation could have specified that a notification be attached to the update packet. Then the update of `Flag` by the sender would have transferred control to the specified notification handler at the receiver.

In general, a VMMC program will perform most of its receive buffer exports and imports during initialization, and then utilize them during its computation. This has the effect of "factoring out" the expensive protection checking and buffer management operations from the critical path of data transfer, and allowing the actual transfer initiation to be performed at user-level. For inner-loop data transfers in large applications, especially those with dependencies, the savings in software overhead can be very significant.

2.4 Summary

Virtual memory-mapped communication is a simple model for specifying data transfers between regions of virtual memory. Although VMMC is a general model, the API presented in this chapter allows applications to easily utilize the features of the SHRIMP network interface and achieve extremely low-overhead data transfer, as shown in Chapter 5.

The API separates data destinations from data movement through import-export mappings. This allows the costly protection checking and receive buffer allocation requirements to be dealt with when mappings are created, rather than each time data is transferred.

The API supports the separation of control transfer and data transfer through the VMMC notification mechanism. Because the SHRIMP network interface transfers received data directly to memory, control transfers are optional. The notification mechanism provides a convenient abstraction of the hardware interrupt facilities,

and extends their functionality with useful features such as interrupt (notification) queueing.

The API allows communication to overlap computation by providing a non-blocking `Sendmsg` operation together with a mechanism to check for its completion. This operation simply specifies a transfer, which is actually performed with DMA by both the sending and receiving network interfaces. While the DMA is occurring, some amount of computation can be performed.

Chapter 3

Hardware Support for VMMC

Although virtual memory-mapped communication can be used with existing network interfaces, its efficiency can be dramatically improved with custom hardware support. Sufficient support can eliminate practically all of the communication overhead required by current network interfaces.

This chapter explores the design space of network interface hardware that supports the VMMC model to one degree or another. Although it is possible to implement VMMC on virtually any DMPP using system calls to send and interrupt service routines to receive, this chapter only considers hardware that improves the efficiency of VMMC over this basic approach.

The coverage of this chapter is by no means exhaustive. It is intended to give the reader a good understanding of the range of opportunities for improving VMMC by adding hardware to existing systems, or by modifying existing architectures. The designs presented in this chapter are only outlines of solutions, with enough detail to clarify the advantages and/or disadvantages of each. The interested reader should continue with the next chapter, which presents the design of an entire system in detail.

Each of the following sections deals with one aspect of the VMMC model and considers various approaches to improving its performance.

3.1 Support for Direct Data Transfer

The VMMC model assumes that a sender specifies the destination of a message at the receiver through an import-export mapping. The most efficient way to implement this is to have the sender specify a destination base address in the receiver's memory, and have the receiver transfer the data directly to that destination.

The software implementation of the receive operation is to have the network interface generate an interrupt to the host CPU, which copies the received data from the network interface to the specified destination. A far better, hardware-supported approach is to have the network interface transfer received data directly to memory without host CPU involvement.

3.1.1 Global naming

Direct data transfer can be accomplished with the use of a global memory naming scheme. The basic idea is to specify regions, or segments, of destination memory using global names that are understood by all nodes. A receiving network interface associates each name with a DMA descriptor that specifies the physical pages of the named region. If a page is re-mapped or un-mapped, this information is reflected in the descriptor and dealt with when data arrives.

The simplest global naming scheme allows destination pages to be assigned to "network virtual addresses" in a manner not unlike normal virtual-to-physical mapping. A receiving network interface performs the translations from network virtual addresses to local physical addresses using the DMA descriptor, and transfers received data directly to memory. The DMA descriptors are basically translation tables that are maintained by the operating system, and page faults can occur on un-mapped pages.

To reduce the average table lookup latency, a Translation Lookaside Buffer (TLB) could be added. The TLB is a small, associative memory that caches recent translations from network to physical addresses, and can produce a result very quickly. If the TLB does not contain a translation, the full descriptor table must be consulted and the new translation can be stored in the TLB.

The performance of this scheme is dependent upon the rate of network interface TLB misses, which lead to relatively costly TLB entry replacements. Since an entry can only provide translation for the addresses of a single network virtual page, and large messages are composed of many pages, a large TLB might be required to avoid constantly replacing entries. Associative memory tends to be very expensive, so large TLBs are generally impractical.

A related scheme is to use just the TLB without the hardware translation table, and have the operating system service TLB misses. This reduces the amount of network interface hardware, but may still require a large TLB.

The principal advantage of global naming schemes is the decoupling of sender's and receiver's addressing; receivers have the freedom to change mappings locally, without involving senders. Although this can be accomplished with a traditional receiver-side interrupt and operating system intervention, the hardware supported global naming schemes are far more efficient and allow communication to overlap computation.

The principal drawback of global naming schemes is their complexity. The translation tables can be quite large, depending on the granularity of naming, and mechanisms must be implemented to perform fast translations. This can be especially difficult at the receiver if the global name space is large. Also, if the naming scheme applies to source and destination data, then translations are required to send as well as receive. See [Wil92, Osb94, TLL94, Gil96] for examples of global naming schemes.

3.1.2 Direct Virtual Memory Transfer

Another method for supporting direct data transfer is to have packets carry a destination virtual address and a process identifier (PID) to identify a virtual address space. The network interface must translate the virtual page number and PID to a specific physical page number in order to transfer the data.

To perform this translation efficiently and not involve the host CPU, the network interface could keep a table (probably a hash table) of process-specific virtual-to-physical page translations, and reference the table for every received page. The operating system would be responsible for keeping the table consistent with main

memory mappings. The obvious disadvantages of this scheme are the complexity of the hardware table, and the latency of the table lookup.

However, the average lookup latency could be reduced with a TLB, as described in the previous section. In this case, the TLB would cache translations from local virtual addresses to physical addresses. Again, the effectiveness of such a scheme depends upon the rate of TLB misses, and there is a complexity drawback.

3.1.3 Direct Physical Memory Transfer

The situation is less complicated if the received packet contains a physical destination address. The only concern in this case is that the intended destination virtual page be mapped to the destination physical page when the data arrives. This can be insured by pinning[1] the intended virtual page to the destination physical page specified by the sender. Although pinning is simple and straightforward, it reduces the number of usable physical pages.

A software scheme for avoiding page pinning is to provide a mechanism with which a receiver can inform all senders to a particular physical page of any changes to the mapping of that page. The senders would either be informed of a new destination page, or told to invalidate the destination. Another protocol could be used to determine the correct destination at the time of the next send on a demand basis. This scheme has the obvious disadvantages of considerable software overhead and latency due to consistency maintenance.

Another way to avoid pinning would be to have the sender send the intended virtual page number and PID along with the physical page number. The receiving network interface could keep a table with an entry for every physical page that specifies which virtual page of which process is presently mapped to the physical page. If the intended page is mapped, the data transfer could proceed without CPU intervention. If not, a page fault would be generated to the host CPU.

Since the network interface mapping table requires only one entry for every page of physical memory, its size is relatively small. In addition, no TLB is required because

[1]When a virtual memory page is pinned, then the underlying physical page cannot be swapped out or replaced by another virtual page.

the table can be indexed directly, unlike the virtual-to-physical address translation table described for direct virtual memory transfer in the previous section.

For this scheme to work, the operating system pager must keep the network interface mapping table consistent with virtual memory mappings. Therefore, it must be careful not to change a mapping to a page that is being updated by the network interface. This can be accomplished by simply having the pager always avoid the destination physical page when the network interface is busy.

A disadvantage of this scheme is that the page fault handler would have to insure that the destination virtual page gets mapped to the specified destination physical page, probably requiring some other virtual page mapped to the destination physical page to be moved and re-mapped. Another disadvantage is the increased overhead of storing and sending the virtual page number and PID.

3.2 Support for Automatic Update

Recall from Section 2.1.2 that automatic update allows the local memory of a process to be bound to an imported receive buffer, such that all updates to the local memory cause updates to the receive buffer as well. Figure 8 shows this binding between a *source virtual address space* and a *destination virtual address space*.

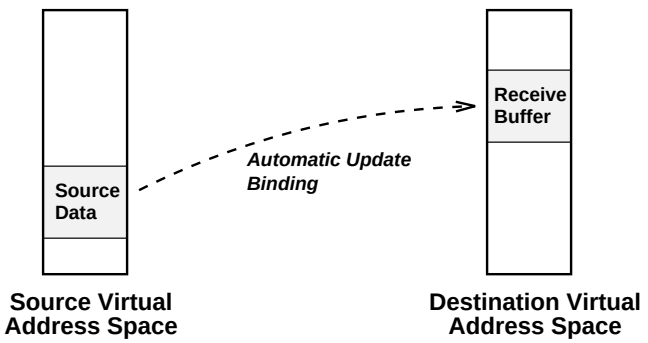


Figure 8: Automatic update binding of virtual memory spaces.

Automatic update can be implemented in software on virtually any machine by inserting code after every mapped memory write to duplicate the write to the bound receive buffer. If the write is performed only for the purpose of sending the data, then

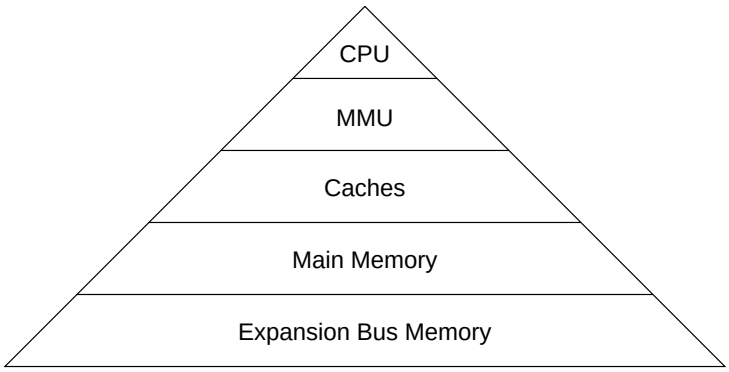


Figure 9: A typical memory hierarchy.

the local write could simply be replaced by the remote write to the receive buffer, although this approach violates the VMMC model and API described in Chapter 2.

In any case, it is far more efficient to add hardware support for automatic update, with the only architectural requirement being that a write to a processor's memory must be seen and propagated to the bound memory in a consistent manner. In order for network interface hardware to see local writes, it must be able to "snoop" the datapath between the CPU and memory somewhere.

Figure 9 shows a typical memory hierarchy for a single-processor system. In principle, automatic update snooping could be performed at any level of this hierarchy with appropriate hardware and software support.

3.2.1 Snooping the CPU

When a virtual memory write operation is generated by the CPU, it is translated to a physical memory write by the memory management unit (MMU). It is certainly possible, though not practical, to add network interface hardware that snoops all virtual memory writes before MMU translation and sends them to the appropriate destination if they are involved in an automatic update binding. Since most processors integrate the MMU with the CPU, this method of snooping requires integration of network interface hardware into the heart of the memory write datapath.

To implement automatic update, the network interface needs a way to select writes to bound addresses and send only those to the appropriate destination. This

can be accomplished by maintaining a table of automatic update bindings along with a method of searching the table.

The advantage of snooping the CPU is that it is a direct implementation of the VMMC model. Any virtual address can be involved in an automatic update binding irrespective of its physical mapping, and any cache update policy can be used since the write cycle is snooped before the first-level cache.

There are several practical disadvantages, however. First of all, non-trivial hardware is required to snoop any high-speed CPU bus. Second, in the case of most microprocessors this hardware must be designed into the processor chip complicating it, and probably slowing down its clock rate. Third, since low latency is important, a hardware hash table of some sort is required to store the automatic update bindings so that a fast lookup can be performed. Alternatively, a TLB-like scheme could be used with an associated lookup penalty on a miss.

3.2.2 Snooping the First-Level Cache

All memory writes from the CPU are propagated to the first-level cache as either virtual or physical addresses, depending on the relationship between the MMU and the cache. It is possible to build network interface hardware that snoops all writes to the first-level cache and sends those involved in automatic update bindings to the appropriate destination.

In the case of a virtually-addressed cache, this scheme basically has the same advantages and drawbacks as the previous scheme since the cache is typically integrated into the processor, often at the same level as the MMU and TLB in the memory hierarchy.

In the case of a physically-addressed cache, automatic update bindings must be maintained in terms of physical addresses. Therefore, a physical page involved in an automatic update binding cannot be shared by more than one virtual page, unless the aliasing of the binding is desired. Snooping before a first-level, physically-addressed cache still has the advantage that any cache update policy can be used, including write-back.

If the first-level cache is external to the CPU chip (which is highly unlikely) then snooping before it is far more practical. However the high speed of the first-level cache bus makes this quite challenging. If the first-level cache is on the CPU chip, the drawbacks of integrating snooping hardware with the CPU discussed in the previous section remain.

Another advantage of snooping physical addresses is that the physical address space is considerably smaller than the virtual address space, so the lookup in the table of automatic update bindings is potentially much more practical. For instance, if bindings are maintained on a per-page basis, then the maximum number of table entries is equal to the number of physical memory pages, and direct indexing can be used.

3.2.3 Snooping the Second-Level Cache

The second-level cache is usually external to a microprocessor and is thus easier to snoop. However, if the first-level cache is write-back, then writes to dirty lines will not be seen at the second level. Therefore, lines that are involved in automatic update bindings must either remain uncached in the first level, or be cached write-through.

An alternate communication scheme is to allow all lines to be cached write-back in the first level, and flush the appropriate lines in order to send the data. The flush can either be performed with a dedicated instruction if the processor supports it, or by reading some other memory that is guaranteed to be cached in the same line, causing a write-back of the data to send. Note that this approach is only useful for explicit data transfer and does not implement the automatic update model, where remote updates occur as side-effects of local memory writes.

The technical challenges of snooping the high-speed second-level cache bus are as daunting as snooping an external first-level cache. Since second-level caches are typically physically-addressed, the advantage of an easier table lookup is enjoyed.

Cache hierarchies deeper than two levels are uncommon and the issues in regard to snooping them are similar to the second-level case discussed here. Therefore, they will not be considered.

3.2.4 Snooping the Memory Bus

Snooping the memory bus is far less technically challenging than snooping the caches because it is slower and more accessible. The principal drawback is that the memory bus is lower in the memory hierarchy, so there is a larger penalty associated with propagating writes there. For instance, if all addresses involved in automatic update bindings are not cached or are cached write-through, an application's runtime could be increased. If they are cached write-back, the line flushing operation for sending is more expensive because all levels of the cache hierarchy need to be flushed.

The principal advantage of snooping the memory bus instead of the cache buses or the CPU is implementation feasibility. As long as the memory bus is accessible, this option does not require costly modifications to highly-integrated components.

Another attractive advantage of the memory bus over cache buses is that it is usually more standardized, so a single network interface design could potentially be used on multiple platforms and survive system revisions. Snooping the memory bus is especially attractive on a symmetric multiprocessor, where multiple CPUs share a single memory bus.

As is the case for a physically addressed cache, memory bus snooping requires automatic update bindings to be maintained in terms of physical addresses.

3.2.5 Snooping the I/O Expansion Bus

Most multiprocessor systems include an expansion bus associated with each CPU or group of CPUs, and often allow virtual memory to be mapped to physical memory on the expansion bus. It is possible to implement automatic update by requiring that all physical addresses involved in bindings be mapped to memory on the expansion bus. Since building expansion cards is well understood, this alternative is technically straightforward.

Building the snooping hardware on the expansion bus enjoys the advantage of standardization to an even greater degree than the memory bus. The drawback is the access time of memory on the expansion bus, which is usually quite a bit longer than main memory access since arbitrations and transfers on both buses are involved. In

addition, many systems do not allow expansion bus memory to be cached, possibly inducing a severe performance penalty if applications use that memory directly. If not, then copying is required for data transfer.

3.3 Support for Deliberate Update

Recall that by using deliberate update, an application can send data from local memory to any previously imported receive buffer, creating a deliberate update binding for the duration of the data transfer.

3.3.1 System-Level

When deliberate update is implemented entirely in system software, the `SendMsg` operation described in Section 2.2.4 is a system call. For this implementation, virtually any network interface hardware can be used as long as it allows for a sender-specified receive address. The system software performs the necessary address translation, and transfers the data from main memory to the network interface by copying or initiating DMA.

Although such an implementation is relatively simple and flexible, it is unfortunately slow because a system call and a good deal of software overhead are involved.

3.3.2 User-Level

The overhead of deliberate update transfer initiation can be dramatically reduced by eliminating the `SendMsg` system call and replacing it with a user-level deliberate update macro. The challenge is to find schemes that provide protection between users and processes.

Standard network interface hardware, which is designed to be used by the operating system, is generally not able to support user-level deliberate update because it does not provide protection. For instance, many network interfaces expect the operating system to build headers that determine message destination. If this responsibility is given to the users, then an errant or malicious process could send messages to other

users, possibly corrupting their computations. What is needed is hardware support for protection so that deliberate update transfers can be initiated without operating system involvement.

Network Interface Memory

A simple solution to the user-level protection requirement is to provide send buffer memory on the network interface and allocate portions of it as send buffers to individual protection domains. User-level transfer is provided by mapping this memory into a user's virtual address space, and then copying the message data into the send buffer with a `SendMsg` macro.

This solution has the advantage of simplicity since the network interface is literally handed the data, and only needs to packetize and send it. The drawbacks are that an explicit copy by the CPU is required, and that the amount of send buffer memory is limited.

Static Binding

A better scheme for supporting VMMC is possible; one that minimizes the amount of network interface hardware by taking advantage of the address translation and protection mechanism found in almost all processors: the MMU. The following scheme was developed as part of the SHRIMP project by the author, Cezary Dubnicki, Edward Felten, Kai Li, and Jonathan Sandberg [BLA+94].

The idea is to put a DMA engine on the network interface, and provide a user-level mechanism for initiating protected, single-page data transfers from main memory to the network interface. The advantage of restricting transfers to single pages is that the normal MMU page-oriented address translation and protection mechanisms can be utilized, as described below.

In order to send data, a *static* deliberate update binding is created between a region of source data and an imported receive buffer, just like the automatic update binding in Figure 8. This is a slightly restricted version of the VMMC model, which allows dynamic binding for deliberate update transfers.

The network interface on the sender keeps a table, indexed by physical page number, of destination physical pages in the system. That is, each entry contains a pointer to a destination node and a physical page on that node. Since virtual pages of the source data on the sender and the receive buffer on the receiver are mapped to real physical pages, the network interface page table essentially keeps track of which source pages are mapped to which destination pages as shown in Figure 10. Since the network interface page table is indexed by physical page number, there is a one-to-one, linear association between source physical pages and entries in the page table.

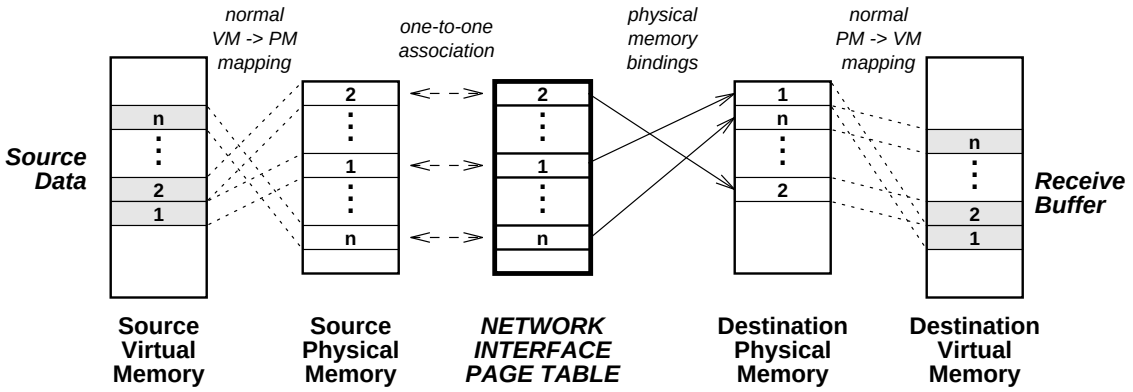


Figure 10: The network interface page table maintains bindings between the physical pages of the source data and the receive buffer.

As described in Section 2.2.1, destination proxy space is used to specify the destination of a deliberate update transfer. In the case of static deliberate update bindings, the proxy space also specifies the source of a transfer because the source is bound to the destination. Therefore, a proxy page number indicates both a source physical page number and a destination network interface page table entry.

In lieu of network interface scatter-gather capability, n individual deliberate update page transfers are required to transfer the entire buffer in Figure 10. That is, the simple scheme outlined here does not include a mechanism for the network interface to know, or be told, which physical pages are mapped to contiguous virtual pages at either the source or destination.

To initiate a page transfer, a user-level compare-and-exchange instruction is issued to a proxy address, causing an atomic operation consisting of a write followed by a read. The value written is the number of words to transfer, and the address is the proxy address associated with the desired base address of the transfer. The network interface returns a success or failure value to the atomic read, and begins the transfer if initiation is successful.

The source base address of the transfer is taken directly from the address of the compare-and-exchange instruction, and the destination is determined by using the proxy page number to index into the network interface page table. The end result is that data is transferred from a source physical page to the bound destination physical page of the receive buffer.

The status of an individual transfer can be checked by performing a load from the proxy space address used in the compare-and-exchange instruction. The load returns a busy/idle bit, the number of words remaining to be transferred, and a match bit, which is set if the load address matches the base address of the current transfer. Therefore, if the status is busy with a match, then the transfer is not yet complete. Synchronous `SendMsg` implementations can repeatedly poll the status to determine when a transfer is complete, or backoff for some amount of time based upon the number of words remaining in the transfer. Also, any process can use the status check to determine if some transfer is in progress, and how far it is from completion.

Protection is provided by the MMU, since pages of proxy space are mapped to virtual pages in the same way as real physical pages. In order to access a proxy page a user-level process must have a virtual page mapped to it, allowing the operating system to control access. Applications are only granted access to proxy pages associated with the physical pages they own, and the associated network interface page table entries will point to physical pages that the applications are allowed to transfer to.

A major drawback of this scheme is that source and destination data structures must be identically aligned. One way to work around this is to allow each local page to be split between two destination pages. That is, each page table entry contains *two* destinations and an offset to specify the split address on the page. With data

transfers allowed between sub-pages, this scheme works as long as the granularity of communication data structures exceeds the page size.

However, another major drawback of this scheme is that it requires the offset of a transfer on a page to be the same on the source and the destination. Therefore, unaligned transfers within pages are impossible.

The polling and backoff schemes described above have the drawback that a bus read cycle to the network interface is required to get the status, interfering with the deliberate update DMA. An improvement for synchronous `SendMsg` is to have the network interface maintain a large integer value in some fixed, cacheable memory location, and return the current value of the integer to the compare-and-exchange read. `SendMsg` simply polls on the fixed memory location in the cache until the network interface changes it, which occurs as soon as the transfer is complete.

Another improvement to the polling scheme is to add hardware to queue deliberate update transfer requests, so that asynchronous `SendMsg` can quickly initiate all the page transfers required for a buffer transfer and computation can resume. Besides the additional complexity of the queueing hardware, the match function of the status check becomes harder because all the pages on the queue must be examined.

Dynamic Binding

To address the major drawbacks of the static physical memory binding scheme, a more flexible scheme using dynamic physical memory bindings is presented here. The basic difference is that this scheme allows any source memory region to be transferred to any imported receive buffer, with the selection performed at the time of the send using *two* proxy spaces, one associated with the source physical memory and one associated with the network interface page table entries. They are called *source proxy space* and *destination proxy space*, respectively.

In this scheme, the network interface interprets a store to destination proxy page "D" as a request to use the contents of network interface page table entry D as the destination of the next deliberate update transfer. The offset of the store on the page is the offset to use on the destination page, and the value stored is the number of words to transfer.

Similarly, a load from source physical proxy page "S" is interpreted as a request to use real physical page S as the source of the next deliberate update transfer. The offset of the load is the offset on the source physical page to begin the transfer from. The load also has the effect of starting the transfer if a valid store to destination proxy space has already been performed.

The network interface hardware interprets a store that specifies a negative number of words to transfer as a command to invalidate a partially initiated transfer (store, but no load). This feature is primarily used by the operating system to maintain atomicity of the transfer initiation sequence by performing the invalidation at every context switch.

Because of the decoupling of source pages and destination entries in the network interface page table, any entry can be used for a destination specification. Therefore, a single source page is no longer statically bound to a destination, but dynamically bound at the time of the data transfer. This has the advantage that data can be transferred to any imported receive buffer (if permission is granted).

Permission to specify source physical pages and entries in the network interface page table is again provided by the existing virtual-to-physical page mapping mechanism of the MMU since access to proxy pages is granted by mapping them to virtual pages.

As in the static scheme, a single dynamic deliberate update transfer cannot exceed the size of a page, and cannot exceed the boundaries of the source or destination physical pages. However, there is no restriction on the alignment of source data and receive buffers. For larger data transfers, a user-level `SendMsg` macro simply performs the individual deliberate updates required to transfer all the data. For example, Figure 11 shows an unaligned transfer that requires five individual deliberate updates to complete.

The dynamic scheme addresses the two major alignment drawbacks of the static scheme. Because a separate offset is specified for the sender and the receiver, data structures need not be page aligned, or at the same offset on sender and receiver. Additionally, there is no restriction on the size of data structures, and any number of

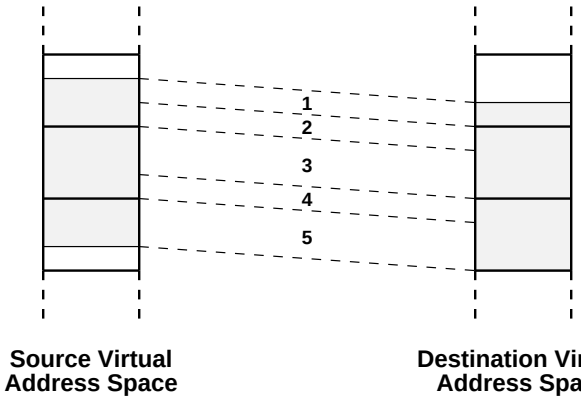


Figure 11: A large transfer performed as a series of deliberate updates, each within source and destination page boundaries.

data structures can share a single page as long as they are within the same protection domain.

This scheme was implemented in the SHRIMP network interface prototype and will be thoroughly discussed in the next chapter.

Eliminating the Network Interface Page Table

On machines with very large virtual address spaces, the network interface page table of destinations can be eliminated by simply including destination node and page numbers in destination proxy addresses.

For example, suppose a network interface page table entry contains an 8 bit destination node field, a 15 bit destination page field, and 5 bits of control information. If every possible entry value is encoded in a destination proxy space page number, then the size of the space is $2^{(8+15+5)} = 2^{28}$ pages.

When an import-export mapping is created, contiguous pages of a process's virtual memory are mapped to the appropriate destination proxy pages that point to the destination pages of the imported receive buffer and have the desired control bits. These proxy pages are likely to be widely scattered in the proxy space.

With a page size of 4 Kbytes, the destination proxy space in this example is 2^{40} bits in size, which is easily accommodated on an architecture with 64-bit virtual and

physical address spaces. Machines such as this are becoming commonly available at the time of this writing.

3.4 Notifications

Recall that a VMMC notification consists of a message transfer followed by an invocation of a user-specified, user-level handler function, and can be used to transfer control or simply notify a receiving process of some event. The hardware needed to support notifications, in addition to the normal received data transfer support, is simply a host CPU interrupt mechanism.

In the VMMC model, each exported receive buffer can have one handler associated with it, so the hardware must provide a mechanism to identify which receive buffer has been used. This can be done by freezing the receive path when the interrupt is generated and allowing a general handler to read some identifying information, such as the destination address, from the network interface. That information should determine which user-level handler function to run.

Although one can imagine more complex schemes involving tables of interrupt vectors on the network interface, there is no good reason not to allow the operating system to do the work since an interrupt is required in any case.

3.5 The Impact of Hardware Support

This chapter began with the claim that VMMC, together with sufficient hardware support, could eliminate practically all of the communication overhead required by current network interfaces. To understand this claim, consider two network interface designs, one with minimal hardware support and one with sufficient support for fully user-level communication.

The minimal design is a traditional, DMA-based network interface modified to include support for a sender-specified destination address (Section 3.1). Such a design was studied in [BDF+95].

The fully user-level design provides hardware support for both deliberate update and automatic update, allowing programs to initiate fully protected, low-overhead data transfers entirely from user level. This design is described in detail in the next chapter.

A traditional network interface design utilizes a receiver-specified destination address and transfers data between memory and the network using DMA. A typical message send operation requires the following steps:

1. The application performs a system call to send a message, crossing to the protection domain of the operating system.
2. The operating system processes the arguments and verifies protection.
3. The operating system allocates a system buffer.
4. The operating system copies the message data to the system buffer, usually consisting of pinned memory.
5. The operating system configures and initiates a DMA engine to transfer the buffer to the network.
6. The operating system waits for the data to be transferred out of the buffer.
7. The send system call returns, crossing back to the protection domain of the application, which resumes.

When the message arrives at the receiver, the following steps are typically performed:

1. The network interface generates an interrupt to the host CPU, crossing to the protection domain of an interrupt service routine.
2. The service routine configures and initiates a DMA engine to transfer the data from the network interface to a receive buffer in memory.
3. The application performs a system call to receive the message, crossing to the protection domain of the operating system.

4. If necessary, the operating system copies the data into user space.

5. The receive system call returns, crossing back to the protection domain of the application, which resumes.

In many systems, the protection domain crossings for system calls involve a costly hardware context switch. Table 2 shows the overhead components of message passing on the traditional network interface, the minimal design, and the fully user-level design. The traditional network interface serves as the baseline of comparison and requires all of the overhead components.

	Message passing overheads	Traditional	Minimal	User-Level
Sender Software	Send system call	x	x	
	Send argument processing	x	x	
	Verify/allocate receive buffer	x		
	Preparing packet descriptors	x	x	
	Initiation of send	x	x	x
Hardware	Data transfer over the network	x	x	x
Receiver Software	Interrupt service	x	optional	optional
	Receive buffer management	x		
	Message dispatch	x		
	Copy data to user space (small)	x		
	Receive system call	x		

Table 2: Components of message passing overhead on three kinds of network interface designs.

On the sender's side, the minimal design does not need to allocate a receive buffer because the VMMC import-export mapping guarantees the space and the sender specifies the destination address. The user-level design eliminates *all* of the send overhead because VMMC transfers are initiated from user level, and the data is taken directly from its source in application space.

On the receiver's side, VMMC eliminates all of the receive overhead for both the minimal and user-level designs because messages are transferred directly to their final destinations in application space.

3.6 Conclusions

This chapter has presented a host of hardware mechanisms to support virtual memory-mapped communication and improve its efficiency. The tradeoffs among these mechanisms follow the general trend of performance versus cost: the more closely a mechanism is integrated with a node, the better its performance and the more costly it is to implement.

As discussed in Chapter 1, the most cost effective DMPPs are networks of commodity PCs. On such machines, the hardware support for VMMC must be completely contained within the network interfaces, limiting the choices for VMMC hardware support considerably.

All PCs provide access to a standard expansion bus, and some provide access to the memory bus. Integration further up the memory hierarchy is not easily accomplished because access is not provided. Reference addresses on both the memory and expansion buses are physical, and memory on the expansion bus is generally not cacheable.

With these constraints in mind, network interface support for VMMC on a PC-based DMPP would include direct physical memory data transfer. An ambitious implementation would support global naming or allow page swapping, as described in Section 3.1, while a straightforward implementation would simply pin destination pages and provide direct physical memory transfer.

The PC network interface would probably include deliberate update with dynamic binding because its implementation is not significantly more complicated than the static binding scheme. Depending upon the capabilities of the accessible memory bus interface, deliberate update transfer initiation and DMA would be performed on either the memory bus or the standard expansion bus. Regardless of capability, implementing deliberate update on the memory bus would likely be a far more challenging option due to high speed and electrical loading limits of modern memory buses.

A similar tradeoff between the memory bus and the expansion bus exists for automatic update snooping. However, providing expansion bus snooping affects more

than just automatic update communication performance since expansion bus memory is unlikely to be cacheable. If mapping is done on a page basis, than all pages used for automatic update would have to be mapped to the expansion bus and all references to those pages would suffer the full access latency. Fortunately snooping is far less intrusive than performing DMA, so the implementation challenges of providing memory bus snooping are not nearly as daunting as those of providing deliberate update DMA on the memory bus.

To summarize, the VMMC hardware support on a network of PCs must remain near the bottom of the memory hierarchy because of the access provided. However, the only feature of VMMC that could benefit from better access to the memory hierarchy is automatic update snooping, because deliberate update DMA and direct transfer of incoming data must operate at the memory or expansion bus level anyhow. Even so, the potential performance gain of tightly integrated snooping hardware would probably not be worth the cost of the implementation.

Chapter 4

The SHRIMP Network Interface

This chapter describes the design and implementation of the SHRIMP network interface, which was built in order to demonstrate the advantages of hardware-supported virtual memory-mapped communication, and understand its weaknesses. The network interface has been used to construct a prototype SHRIMP system, which is also described. Preceding these descriptions is a discussion of the goals and constraints, both self-imposed and otherwise, that guided the project.

4.1 Design Goals and Constraints

There were several key goals and constraints guiding the design of the SHRIMP network interface and the prototype system, ranging from the purely philosophical to the blatantly practical. The philosophical goals were derived from the beliefs of the project members, while the practical constraints were imposed by the availability of resources such as people, equipment, time, and funding.

4.1.1 Low-Overhead Communication

Probably the most important design goal for the SHRIMP network interface is that it provide low overhead data transfer between nodes. Reduction in overhead has a direct effect upon data transfer latency, which is roughly the sum of the software

overhead and the hardware latency. Data transfer latency remains a key determinant of parallel application performance because of program dependencies and because software overhead occupies processors.

Support for low overhead data transfer is reflected in the design of VMMC. Both automatic update and deliberate update can be implemented with almost no software overhead if sufficient hardware support is provided. Since VMMC was designed with this support in mind, designing such hardware is relatively straightforward.

4.1.2 Support for Multiprogramming

Uniprocessor systems utilize multiprogramming because it allows useful processing to be done during I/O and paging, it supports interactive jobs well, and it allows multiple users to share a system. These benefits should be extended to parallel machines, so an important design constraint for the SHRIMP network interface is that it support general multiprogramming under a variety of scheduling policies. The key design impact of this constraint is the need for protected communication.

Some existing network interface designs support low-overhead communication through user-level message passing, but they either prohibit multiprogramming, or constrain how processes are scheduled. Systems that prohibit multiprogramming use hardware to divide the machine into partitions, and then run a single parallel job in each partition. Systems that restrict multiprogramming allow several parallel jobs to share the machine, but guarantee protection only if certain scheduling policies are used. For example, the CM-5 hardware provides safe user-level communication, provided the operating system uses strict gang scheduling within each partition [Thi92].

Because the SHRIMP system is a research machine, it should not prohibit multiprogramming or constrain how processes are scheduled. Therefore, the communication mechanism must be safe, irrespective of which processes are executing at any given time. Furthermore, the communication mechanism should allow for experimentation with various scheduling policies in order to determine which are the best.

4.1.3 Bus Bandwidth

Since the host buses are usually the communication bandwidth-limiting factor (in the absence of network contention), it is impractical to design hardware with bandwidth significantly exceeding that of these buses. The bandwidth goal for the SHRIMP system is that it be able to utilize all of the available bandwidth on the expansion bus, and as much as possible on the memory bus. It is not possible to utilize all of the memory bus bandwidth when accessing memory via the expansion bus because the expansion bus is slower. The network interface does have a direct, read-only connection to the memory bus, however, and this connection is expected to operate at full bandwidth.

4.1.4 Use of Commodity Components

The computer equipment market has become so large that there are commodity components ranging from software applications to entire computer systems. Competition within these markets drives performance up and prices down, making commodity components an excellent value. One of the primary constraints of the SHRIMP project is to utilize commodity components wherever possible in order to take advantage of the savings attainable, both in cost and design time.

4.1.5 Hardware Design Simplicity

Although it is impossible to avoid complexity in the implementation of a useful communication interface, it is worthwhile to avoid pushing that complexity into the hardware for several reasons. First, hardware is the least mutable portion of the interface, so changes require the highest expense and longest delay. Second, complexity can easily have unintended negative impacts on hardware efficiency by enlarging the control path or extending datapath pipelines. Third, the lowest levels in the communication interface hierarchy are forced to change the most frequently by technological advances such as new I/O or memory buses. Therefore, hardware complexity can impede the ability of the interface to track new technology. Fourth,

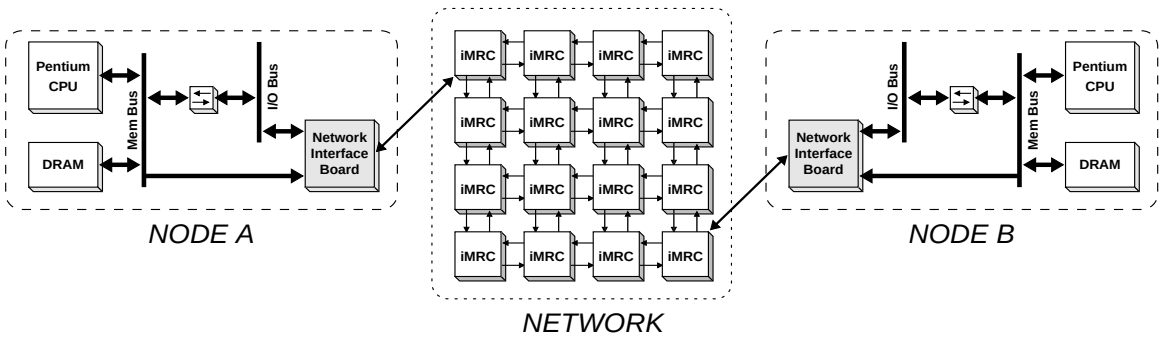


Figure 12: Architecture of the prototype SHRIMP system.

complex interface hardware is less portable to different system platforms. Finally, simple hardware costs less to build.

There are, however, cases where hardware complexity is necessary; specifically, when additional hardware can be used to meet design goals that software alone cannot. The prudent approach is to try and use as little hardware as possible unless it is clear that additional hardware is required. For instance, the SHRIMP network interface uses hardware to avoid a system call for every data transfer operation. The reduction in send latency achieved by this design was sufficient to justify the additional hardware.

4.2 Architecture of the SHRIMP Prototype

This section introduces the prototype SHRIMP system in order to provide a context for the rest of the chapter, which describes the network interface design in detail.

4.2.1 Hardware Components

Figure 12 shows the hardware architecture of the prototype system. It consists of four interconnected nodes, each of which is a DEC 560ST PC containing an Intel Pentium Xpress motherboard [Int93]. The motherboard has a 60 MHz Pentium CPU [Sai93] with an external 256 Kbyte second-level cache, and 40 Mbytes of main DRAM memory. The Xpress memory bus [Int] has a maximum burst write bandwidth of 73

Mbytes/sec, and is accessible through a memory expansion connector, which carries the majority of the bus signals. Peripherals are connected to the system through the EISA expansion bus [BCP92], which has main memory mastering capability and a maximum burst bandwidth of 33 Mbytes/sec.

Main memory data can be cached by the CPU as write-through or write-back on a per-page basis, as specified in process page tables. The caches snoop DMA transactions and automatically invalidate corresponding cache lines, thereby keeping consistent with *all* main memory updates, including those from EISA bus masters.

The network connecting the nodes is an Intel routing backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips [TD92], and is the same network used in the Paragon multicomputer [Int91b]. The Intel Mesh Routing Chip is essentially a wider and faster version of the Caltech Mesh Routing Chip [DS86]. The backplane supports deadlock-free, oblivious wormhole routing [DS87], and preserves the order of messages from each sender to each receiver.

Data is transferred by the backplane as a sequence of flits[1], each of which is two bytes in size. The backplane supports two modes of communication: interlocked and streaming. Interlocked mode is a "zero-slack" protocol where each flit is individually acknowledged, requiring a roundtrip transmission on the cable. Streaming mode is a four-slack protocol where up to four flits can be transmitted before the first acknowledgement is required by the sender. Streaming mode has lower latency and higher bandwidth than interlocked mode, but is less tolerant of skew between individual signals.

One of the SHRIMP nodes hosts a network adaptor board, which is used to boot and configure the backplane. This node also coordinates diagnostics and booting for the entire system. In order to simplify the startup procedures, the SHRIMP nodes are connected by a commodity Ethernet, which can also be used to exchange low-priority messages.

The only custom hardware components in the SHRIMP system are network interface boards, which connect the nodes to the backplane. The network interface boards are the key system components that provide the hardware support for VMMC.

[1] Flow control unit [AS88].

4.2.2 Software Components

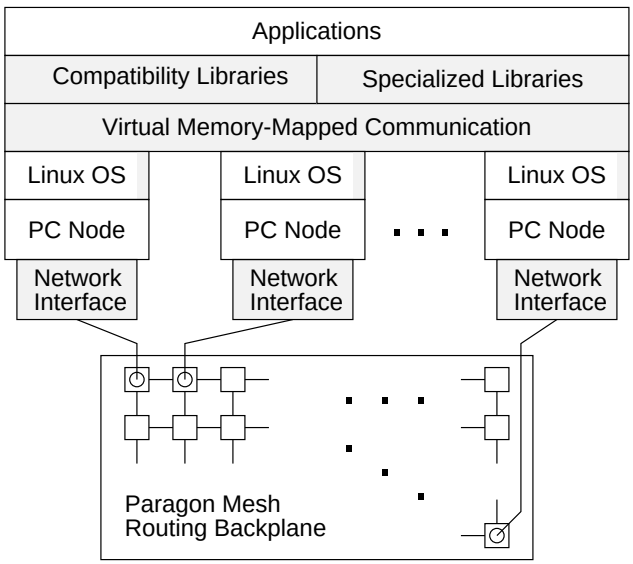


Figure 13: Hardware and software layers in the prototype SHRIMP system. Custom components are shaded.

Figure 13 shows the existing software components of the prototype system (and a bit of hardware for orientation). The Virtual Memory-Mapped Communication component is a "thin" library layer that implements the VMMC API described in Chapter 2, provides direct access to the network for data transfers between user processes, and handles communication with SHRIMP daemons (not shown). SHRIMP daemons are trusted servers (one per node) that cooperate to establish and destroy import-export mappings between user processes. The daemons use memory-mapped I/O to directly manipulate the network interface hardware. They also call SHRIMP-specific operating system calls to manage receive buffer memory and to influence physical memory management in order to ensure consistent virtual memory communication mappings across nodes.

A few system calls have been added to the Linux operating system kernel, as well as a few minor modifications to the scheduler and pager. These modifications provide necessary support for the network interface hardware and VMMC library.

There are two types of user-level communication libraries that utilize VMMC: the Compatibility Libraries and the Specialized Libraries. The compatibility libraries currently consist of NX/2 message passing [ADFL96], Sun RPC [BF96], and stream sockets [DDF96], all fully compatible with existing implementations. Existing applications using the standard interfaces can run with these compatibility libraries without any modification. There is currently one specialized library, which implements remote procedure call, taking full advantage of the SHRIMP VMMC mechanism [BF96]. Both the compatibility libraries and the specialized library run entirely at user level and are built on top of the VMMC layer.

4.2.3 The Network Interface

The SHRIMP network interface is designed to support protected, user-level virtual memory-mapped communication and provide the necessary support for eliminating system code from the critical communication path.

The prototype design is basically a combination of the designs presented in Sections 3.1.3, 3.2.4, and 3.3.2 of Chapter 3. That is, senders specify destination physical addresses, automatic update snoops the memory bus, and a dynamic deliberate update binding scheme is used. Although some of the information in this section duplicates information already presented in the previous chapter, it is reiterated for completeness and clarity here.

Figure 14 shows a block digram of the network interface and its principal datapaths. It is presented here for reference during the following discussions and will be fully explained in Section 4.6.

4.3 Implementation of Automatic Update

The prototype automatic update implementation is a page-based, eager-sharing mechanism whereby updates to virtual memory are duplicated to remote virtual memory through automatic update bindings, as shown in Figure 8. The remote update occurs as a side-effect of a local memory write.

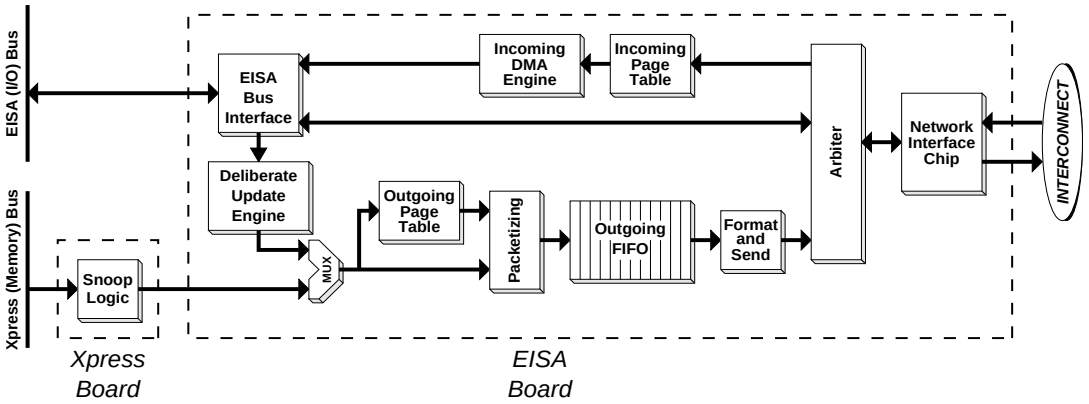


Figure 14: Datapath of the SHRIMP network interface.

The automatic update hardware snoops all writes on the memory bus, and if the written address has an automatic update binding, then a packet containing the data and the destination address is sent to the destination node.

4.3.1 Binding for Automatic Update

Although the VMMC API supports communication between virtual memory spaces, the underlying hardware mechanism operates in terms of physical pages. The virtual memory pages comprising the receive buffer are mapped to physical pages in the *destination physical address space* by the virtual memory system in the normal way. Likewise, virtual pages of the source data are mapped to physical pages in the *source physical address space*. The SHRIMP network interface maintains automatic update bindings from source physical memory pages to the bound destination physical memory pages.

The Outgoing Page Table

In order to maintain the binding between physical pages, the network interface contains a table of destination physical page locations called the Outgoing Page Table (OPT). Each entry in the OPT points to some physical page in the system by specifying a node and a single physical page on that node. The network interface defines a one-to-one association between source physical pages and entries in the

OPT, where entry "P" corresponds to physical page P. An automatic update binding between P and a destination physical page, "D", is created by writing a pointer to page D into entry P of the OPT, and specifying that it be used for automatic update by setting the "AUenable" bit in the entry.

Each entry in the OPT also contains a sender-specified interrupt bit, which can be used to trigger a notification on the receiving node. When the bit is set, every automatic update packet generated from that entry will contain a notification request bit in its trailer.

The Incoming Page Table

In addition to the OPT, the network interface contains an Incoming Page Table (IPT) with one entry for every page of physical memory. Each IPT entry contains two bits, one of which is used to enable the associated page to receive data. The receive enable bit allows a node to control which physical pages are modified by its network interface, since received data is transferred directly to main memory without CPU intervention. If data is received for a page that is not enabled, then the network interface will freeze the receive datapath and generate an interrupt to the node CPU. The CPU can then over-write the receive address and re-enable the incoming datapath, causing the packet to be transferred to the new address. The other IPT bit is used to enable notifications on the associated page and will be discussed in Section 4.5.

VMMC Binding

To create an automatic update binding in response to a VMMC `Bind` request (Section 2.2.5), the SHRIMP daemons running on the source and destination nodes cooperate to configure the network interface page tables and create mappings between individual physical pages, as shown in Figure 15.

The daemon running on the source node creates pointers to the destination physical pages in the OPT entries corresponding to the source physical pages, and the daemon running on the destination node enables the pages corresponding to the destination physical pages of the receive buffer in its IPT. Since the OPT entries

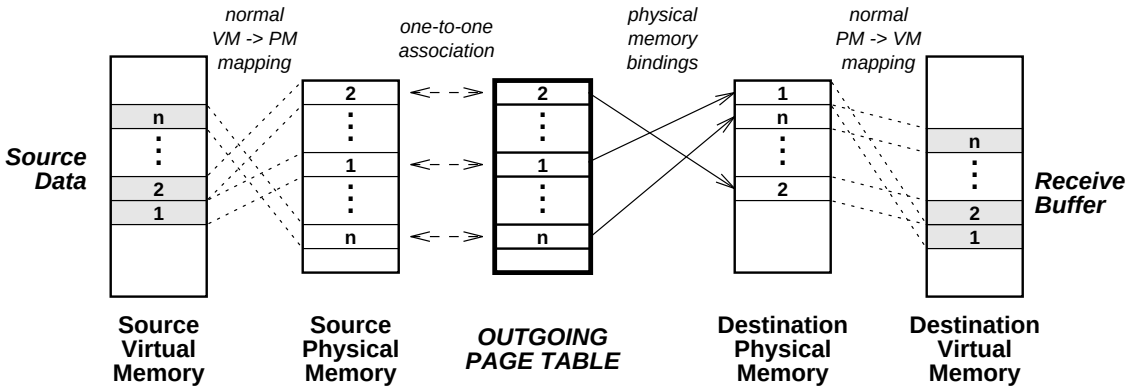


Figure 15: Automatic update binding between the physical pages of the source data and the receive buffer.

used for the import mapping must correspond to the physical pages that are to be bound, some rearrangement of OPT entries may be necessary. This is because a deliberate update mapping (Section 4.4.2) can utilize any OPT entry, and one may be inadvertently occupying the required entry. To complete the binding, the AUenable bits in the bound OPT entries are set.

4.3.2 Hardware Support for Automatic Update

In order to implement automatic update, the SHRIMP network interface board snoops all writes on the memory bus, and looks them up in the OPT to determine if an automatic update binding exists for the page written to. If so, a packet containing the written data is created and sent to the node and page specified in the OPT. Since the remote update occurs as a side-effect of the local memory write, the send overhead is merely the cost of propagating the write to the memory bus, which can be accomplished with write-through caching.

The hardware also has a combining mechanism, which can gather writes to consecutively *increasing* addresses into a single packet. This feature is selectable on a page-by-page basis by setting a bit in the OPT entry.

There are two types of combining pages supported: *standard* and *latency-bounded*. Standard combining simply combines all consecutive automatic update writes and launches a packet when a non-consecutive write occurs, a page boundary is crossed,

or the automatic update packet size parameter (Chapter 5) is exceeded. Latency-bounded combining is the same, but adds a programmable restriction on the maximum time between two automatic update writes. If that time is exceeded, the packet is launched.

4.3.3 Kernel Extensions for Automatic Update

Because the receiving network interface transfers data directly to physical memory, the pages of the receive buffer must be "pinned" to prevent the virtual memory manager from paging them out of memory. To support the pinning operation, the Linux kernel has been extended with two new system calls: `Mlock` and `Munlock`. `Mlock` pins a specified virtual page to a physical page, and returns the physical page number so that it can be used for automatic update binding. `Munlock` simply un-pins the specified virtual page.

To support the snooping operation on the source node, the Linux kernel has been extended with the `Cache_control` system call, which is used to set the cache update policy for a specified physical memory page. `Cache_control` is used by the SHRIMP daemon to configure source physical pages for write-through caching when an automatic update binding is created, and return them to write-back caching when the binding is destroyed.

4.3.4 Automatic Update Pros and Cons

Automatic update communication is designed for very low latency. The send operation is a side-effect of a normal physical memory store operation, so there is a little hardware overhead, but no software overhead whatsoever. The operating system is only involved in creating mappings, in order to provide protection between users and processes. Once a mapping has been established, the sending and receiving processes can use the mapped memory as send and receive buffers, and can communicate without any kernel involvement.

An advantage of automatic update communication is that it is stateless with respect to context in a multiprogrammed environment. That is, virtual memory

mappings for any number of users or processes can coexist, since the underlying physical memory mappings are protected by the MMU in the same manner as ordinary physical memory. There is no physical memory mapping state associated with context.

Another advantage of automatic update communication is that ordinary DRAM is used as send and receive buffers, with no copying required. Applications can map data structures directly between nodes and overlap communication with computation.

A disadvantage of automatic update communication is that it is limited to point-to-point transmission; multicast and broadcast are pushed into software.

A second disadvantage of automatic update communication is that no two data structures with different destinations can be mapped to the same page, since physical memory mapping only supports one destination per page. In principle, this drawback can be avoided, but it prevents the use of general-purpose compilers that do not pay attention to the mapping of data structures.

Finally, automatic update requires perfect alignment between the bound pages, which can be a drawback because it restricts generality.

4.4 Implementation of Deliberate Update

The prototype implements a user-level deliberate update mechanism with dynamic binding. That is, an application can elect to send data from any memory source region to any previously imported receive buffer without operating system involvement. Once initiated, the actual transfer of data from memory to the network interface is performed with standard DMA hardware.

The mechanism for starting a transfer is called User-Level DMA (UDMA) and it consists of minor extensions to traditional DMA hardware and system software. Of interest is how little additional hardware and software is needed to initiate protected transfers from user level.

Although UDMA is a general DMA mechanism, the implementation described in this section is specific to the SHRIMP network interface and is used to initiate data

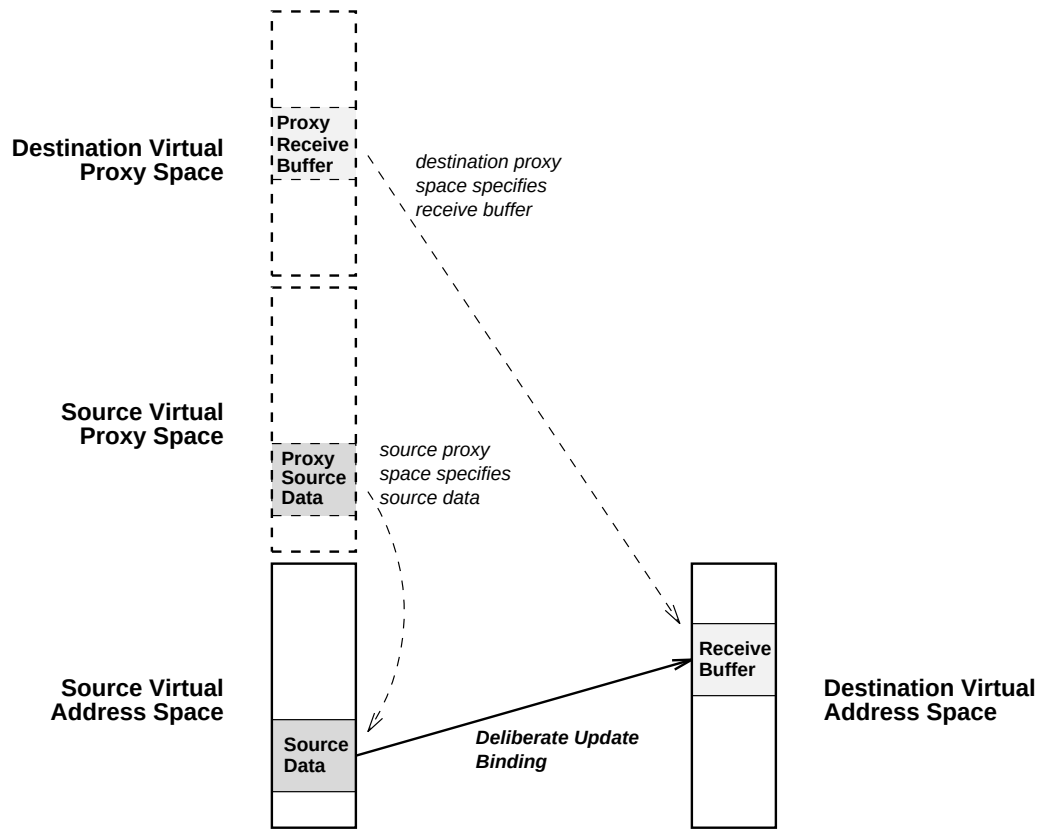


Figure 16: Specification of a deliberate update binding in virtual memory space.

transfers from memory to the network interface. A general description can be found in [BDFL96].

4.4.1 Memory Mapping for User-Level DMA

In order to specify the source memory and destination receive buffer for a data transfer from user level, the virtual address space of a process is divided into three logical subspaces: the *source virtual address space*, which is mapped to physical memory, the *source virtual proxy space*, and the *destination virtual proxy space*. Each of these three subspaces has a corresponding subspace in the physical address space of the node; namely, the *source physical address space*, the *source physical proxy space*, and the *destination physical proxy space*. The source physical address space is the portion of the node's physical address space that is backed by real, physical memory.

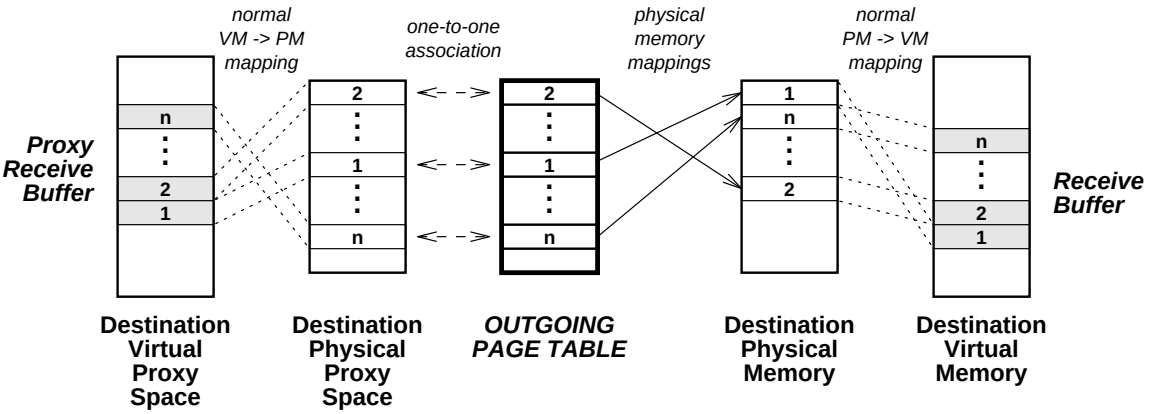


Figure 17: Mappings for UDMA destination specification.

When a receive buffer is imported, a representation of that buffer called the *proxy receive buffer* is created in destination proxy space, as shown in Figure 16. The proxy receive buffer is the same size as the actual receive buffer, and occupies contiguous virtual pages. Every virtual page of the proxy receive buffer is mapped to a physical page in destination physical proxy space, just as every virtual page of the actual receive buffer is mapped to a standard physical page on the receiving node, as shown in Figure 17.

4.4.2 Use of the Network Interface Page Tables

When a receive buffer is exported, the SHRIMP daemon on the exporting node `Mlock`s all of the buffer's physical pages, and receive-enables them in its IPT. When the receive buffer is subsequently imported, the SHRIMP daemon on the importing node creates pointers to all the physical pages of the receive buffer in its OPT.

4.4.3 Interpretation of Proxy Space

The SHRIMP network interface decodes destination physical proxy space and defines a one-to-one, linear correspondence between page numbers in this space and entries in the OPT, as shown in Figure 17. A store to destination physical proxy page "D"

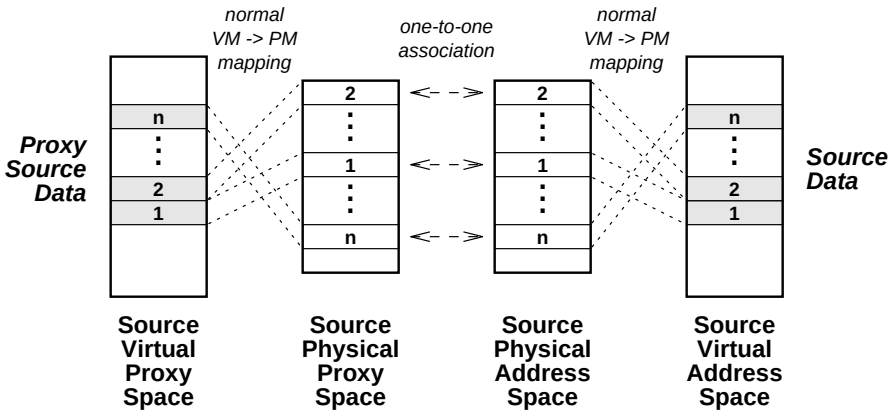


Figure 18: Mappings for UDMA source specification.

is interpreted as a request to use the contents of OPT entry D as the destination of the next UDMA transfer.

Similarly, the network interface decodes source physical proxy space and assigns a one-to-one, linear correspondence between page numbers in this space and page numbers in source physical memory, as shown in Figure 18. A load from source physical proxy page "S" is interpreted as a request to use source physical page S as the source of the next UDMA transfer.

4.4.4 Providing Protection

In order to access physical proxy pages from user level, they must have virtual proxy pages mapped to them. This is the essence of the protection scheme since virtual-to-physical page mappings are controlled by the Linux kernel. The kernel has been extended to map a source virtual proxy page to the appropriate source physical proxy page whenever a standard virtual page is mapped to a standard physical page. This is the same as granting the user permission to specify its page as a source for deliberate update transfer, as shown in Figure 16. For convenience, a one-to-one, linear correspondence is assigned between source virtual proxy pages and standard virtual pages so that source virtual proxy page "S" refers to standard virtual page S.

A similar protection scheme is used for imported receive buffers. Again, a user is granted permission to specify pages of a receive buffer as UDMA destinations by

mapping destination virtual proxy pages to the destination physical proxy pages of the imported buffer. Although the destination virtual proxy pages of the receive buffer are contiguous as shown in Figure 16, the destination physical proxy pages need not be. They must correspond to the OPT entries that point to the destination physical pages, as shown in Figure 17.

4.4.5 User-Level DMA Restrictions

A single UDMA transfer cannot exceed the size of a page, and cannot cross the boundaries of the source or destination physical pages. However, there is no restriction on the alignment of source data and receive buffers. The page limit restriction is necessary because mappings are maintained for individual physical pages, and the network interface does not have gather or scatter capability[2]. Therefore, larger data transfers are simply performed as a series of individual deliberate updates, as shown in Figure 11.

4.4.6 User-Level DMA Transfer Initiation

A single UDMA transfer initiation takes the following form:

```
STORE nwords TO DestProxyAddr
LOAD status FROM SrcProxyAddr
```

The network interface decodes the store to destination *physical* proxy space and uses the page number of the store to select the OPT entry specifying the destination page. The offset of the store is used as the base offset on the destination page, and `nwords` is the size of the transfer. Likewise, the network interface decodes the load from source physical proxy space and uses the corresponding address in physical memory as the base source address of the transfer. The network interface always returns `status` to indicate whether the transfer was started or not. If not, the store-load transfer initiation sequence must be re-issued in its entirety.

[2]Contiguous virtual pages need not be mapped to contiguous physical pages. Gathering refers to reading from non-contiguous physical pages, and scattering is the reverse.

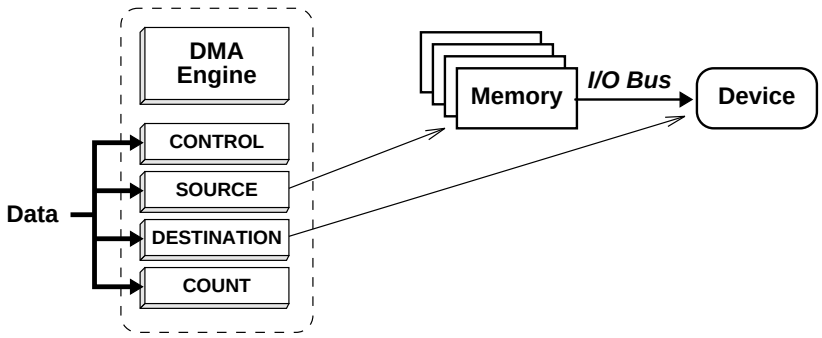


Figure 19: Traditional DMA hardware configured for a memory to device transfer. In general, transfer in both directions is possible.

A transfer is rejected if the deliberate update hardware transfer engine is busy, `nwords` is zero, `DestProxyAddr` or `SrcProxyAddr` are not word-aligned, or a load is performed without a previous store. To maintain atomicity of the store-load initiation sequence, the Linux scheduler has been modified to invalidate a partial initiation on every context switch. The hardware interprets a store with negative `nwords` to be an invalidation and will reject a subsequent load, presumably causing another initiation attempt. The software overhead of a successful transfer initiation is just the time to perform the two user-level memory references.

4.4.7 User-Level DMA Hardware

The hardware support for UDMA is a simple extension of the traditional DMA engine, first implemented on the IBM SAGE computer in 1955 [HP90].

Traditional DMA Hardware

Figure 19 shows a traditional DMA controller configured to perform DMA from memory to a device over a bus. The device typically consists of a single port to or from which the data is transferred. The DMA mechanism typically consists of a state machine and several registers including source and destination address registers, a counter, and a control register. To transfer data from memory to the I/O device, the CPU puts the physical memory address into the source register, the device address

into the destination register, sets the counter to the number of bytes to be transferred, and triggers the control register to start transferring the first datum. After the first datum is transferred, the state machine increments the source register, decrements the counter, and starts transferring the second datum. These transfers continue until the counter reaches zero. Note that the entire data transfer process requires no CPU intervention.

Although this mechanism is simple from a hardware point of view, it imposes several high-cost requirements on the operating system kernel. First, it requires the use of a system call to initiate the DMA operation, primarily to verify the user's permission to access the device and to ensure mutual exclusion among users sharing the device. User processes must pay the overhead of a system call to initiate a DMA operation.

A second requirement is virtual-to-physical memory address translation. Because the DMA controller uses physical memory addresses, the virtual memory addresses of the user programs must be translated to physical addresses before being loaded into the address registers. Virtual-to-physical address translation is performed by the operating system kernel.

A third requirement is page pinning. In order to prevent the virtual memory system from paging out virtual memory pages used for a DMA data transfer, they must be pinned until the transfer is complete. Since the software cost of pinning memory pages is high, most of today's systems reserve a certain number of pinned physical memory pages for each DMA device as I/O buffers. This method may require copying data between memory in user address space and the reserved, pinned DMA memory buffers.

Finally, if a transfer exceeds a single page in size, it must either be performed to or from consecutive physical pages, or a hardware mechanism for locating the physical pages of the transfer must be supported. Typically, such a mechanism consists of a "DMA descriptor" prepared by the operating system in memory and read by the DMA controller.

Altogether, a typical DMA transfer from memory to a device consists of the following steps:

1. A user process makes a system call, asking the kernel to do an I/O operation. The user process names a region in its virtual memory as the source of the transfer and a destination device.

2. The kernel translates the virtual addresses to physical addresses, verifies the user process's permission to perform the requested transfer, pins the physical pages into memory, builds a DMA descriptor specifying the pages to transfer, and then starts the DMA device.

3. The DMA device performs the requested data transfer, and then notifies the kernel by changing a status register or causing an interrupt.

4. The kernel detects that the transfer has finished, unpins the physical pages, and reschedules the user level process.

Starting a DMA transfer usually takes hundreds or thousands of CPU instructions. Therefore, DMA is beneficial only for infrequent operations that transfer a large amount of data, restricting its usefulness.

UDMA addresses the overhead of starting the transfer by eliminating the system call, address translation, protection checking, page pinning, and DMA descriptor creation from the transfer initiation operation. These services are not eliminated entirely, but provided by other mechanisms.

Hardware Extensions for User-Level DMA

The purpose of the UDMA hardware is to provide the minimum necessary support for the UDMA mechanism while reusing existing DMA technology. The UDMA hardware extends traditional DMA hardware to provide translation from physical proxy addresses to real addresses, to interpret the two-instruction transfer initiation instruction sequence, and to guarantee atomicity for context switches in operating systems. Figure 20 shows how the additional hardware is situated between the standard DMA engine and the CPU, and should be compared to Figure 19. Notice that the UDMA hardware utilizes both the CPU address and data in order to communicate very efficiently.

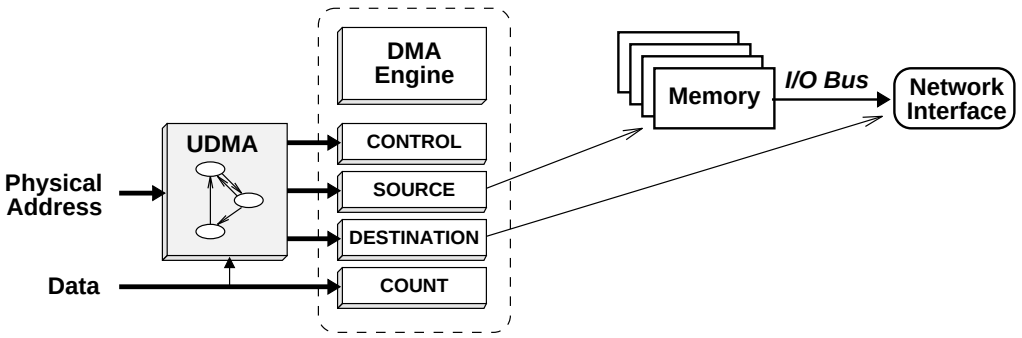


Figure 20: UDMA hardware configured for a memory to network interface transfer.

Address translation from source physical proxy addresses to real memory addresses consists of simple high-order bit substitution since the mapping between source proxy addresses and real memory addresses is one-to-one and linear. Likewise, address translation from destination physical proxy addresses to OPT entry numbers consists of direct selection by page number since the mapping between destination proxy page numbers and OPT entries is one-to-one and linear. The translated source and destination values are loaded into the source and destination registers of the traditional DMA engine, respectively.

User-Level DMA State Transitions

The `STORE`, `LOAD` transfer initiation instruction sequence described in Section 4.4.6 is interpreted by a simple state machine within the UDMA hardware as shown in Figure 21. In the diagram, if no transition is depicted for a given event in a given state, then that event does not cause a state transition.

The state machine manages the interaction between proxy-space accesses and the standard DMA engine. The machine has three states: *Idle*, *DestLoaded*, and *Transferring*. It recognizes three transition events: `Store`, `Load`, and `Inval`. `Store` events represent `STORE`s of positive values to proxy space. `Load` events represent `LOAD`s from proxy space. `Inval` events represent `STORE`s of negative values (passing a negative, and hence invalid, value of `nwords`) to proxy space.

To understand the state transitions, consider first the most common cases. When idle, the state machine is in the *Idle* state. It stays there until a `STORE` to destination

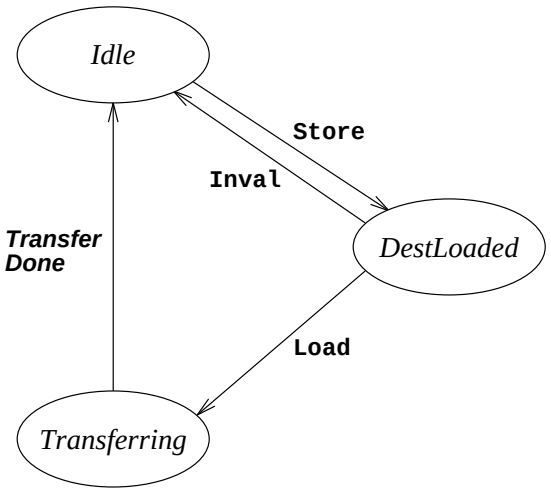


Figure 21: State transitions in the UDMA hardware state machine.

proxy space is performed. When this occurs, the referenced proxy address is translated to an OPT address and put into the destination register, the value stored by the CPU is put into the count register, and the hardware enters the *DestLoaded* state.

The next relevant event is a `LOAD` from source proxy space. When this occurs, the referenced proxy address is translated to a real memory address and put into the source register, and the hardware enters the *Transferring* state. This causes the UDMA state machine to write a value to the control register to start the standard DMA transfer.

When the transfer finishes, the UDMA state machine moves from the *Transferring* state back into the *Idle* state, allowing user processes to initiate further transfers.

Several other, less common transitions are also possible. In the *DestLoaded* state, a `Store` event does not change the state, but overwrites the destination and count registers. An `Inval` event moves the machine into the *Idle* state and is used to terminate a partially completed transfer initiation sequence.

A fourth transition event not shown on the state transition diagram is the `BadLoad` event, which causes a transition from the *DestLoaded* to the *Idle* state. This represents a load from an address in the proxy destination space, and corresponds to a user process asking for a device-to-device transfer, which is not supported.

Status Returned by Proxy Loads

A `LOAD` instruction can be performed at any time to any proxy address in order to check the status of the UDMA hardware. The `LOAD` will only initiate a transfer under the conditions described above. Every `LOAD` returns the following information to the user process:

- INITIATION FLAG (1 bit): set if the access causes a transition from the *DestLoaded* state to the *Transferring* state (i.e. if the access started a DMA transfer); clear otherwise.
- TRANSFERRING FLAG (1 bit): set if the UDMA state machine is in the *Transferring* state; clear otherwise.
- INVALID FLAG (1 bit): set if the UDMA state machine is in the *Idle* state; clear otherwise.
- MATCH FLAG (1 bit): set if the UDMA state machine is in the *Transferring* state and the address referenced is equal to the base address of the transfer in progress; clear otherwise.
- WRONG-SPACE FLAG (1 bit): set if the access is a `BadLoad` as defined above; clear otherwise.
- BAD-ALIGNMENT FLAG (1 bit): set if either the requested source or destination transfer offsets are not word-aligned; clear otherwise.
- REMAINING-WORDS (11 bits): the number of words remaining to transfer if in the *DestLoaded* or *Transferring* state; zero otherwise.

The return value of the `LOAD` instruction that attempts to start a transfer is zero if the transfer was successfully initiated. If not, the user process can check the individual bits of the return value to figure out what went wrong. If the TRANSFERRING or INVALID flags are set, the user process may want to re-try its two-instruction transfer initiation sequence. If other error flags are set, a real error has occurred.

To check for completion of a successfully initiated transfer, the user process should repeat the `LOAD` instruction that it used to start the transfer. If this `LOAD` instruction returns with the MATCH flag set, then the transfer has not completed; otherwise it has.

4.4.8 Operating System Support for UDMA

The UDMA mechanism requires support from the operating system kernel to guarantee the atomicity of DMA transfer initiations, to create virtual memory mapping for UDMA, and to maintain memory proxy mapping during virtual memory paging. The operating system maintains three invariants:

Inv1 If a `LOAD` instruction initiates a UDMA transfer, then the destination address and word count must have been `STORE`d by the same process.

Inv2 If there is a mapping from a source virtual proxy address to a source physical proxy address, then there must be a corresponding mapping from the associated virtual memory address to the associated physical memory address.

Inv3 If a physical memory address is in the hardware source or destination registers, then the address will not be remapped.

These invariants are explained in detail in the following subsections.

Maintaining *Inv1*: Atomicity

The operating system must guarantee invariant *Inv1* to support atomicity of the two-instruction transfer initiation sequence. Because the UDMA mechanism requires a program to use two user-level instructions to initiate a transfer, and because multiple processes may share a UDMA device, there exists a danger of incorrect initiation if a context switch takes place between the two instructions.

To avoid this danger, the operating system must invalidate any partially initiated UDMA transfer on every context switch. This can be done by generating a hardware `Inval` event (i.e. by storing a negative `nwords` value to any valid proxy address),

causing the UDMA hardware state machine to return to the *Idle* state. The context-switch code does this with a single store instruction.

When the interrupted user process resumes, it will execute the `LOAD` instruction of its transfer-initiation sequence, which will return a failure code signifying that the hardware is in the *Idle* state. The user process can deduce what happened and re-try its operation.

Note that the UDMA device is stateless with respect to a context switch. Once started, a UDMA transfer continues regardless of whether the process that started it is de-scheduled. The UDMA device does not know which user process is running, or which user process started any particular transfer.

Maintaining *Inv2*: Mapping Consistency

The virtual memory manager in the operating system must cooperate with the UDMA device to create virtual memory mappings for memory proxy and device proxy spaces, and must guarantee invariant *Inv2* to ensure that a virtual-to-physical memory proxy space mapping is valid only if the virtual-to-physical mapping of its corresponding real memory is valid.

In order for a process to perform DMA from a source virtual memory page, the operating system must create a virtual-to-physical mapping for the associated source proxy page. Each such mapping maps a source virtual proxy page to a source physical proxy page. These mappings are created on demand. If the user process accesses a source virtual proxy page that has not been set up yet, a normal page-fault occurs. The kernel responds to this page-fault by trying to create the required mapping. Three cases can occur, based upon the state of the source virtual page.

- *The source virtual page is currently in memory and accessible.* In this case, the kernel simply creates a virtual-to-physical mapping from the associated source virtual proxy page to the appropriate source physical proxy page.

- *The source virtual page is valid but is not currently in memory.* The kernel first pages in the virtual page and then behaves as in the previous case.

- *The source virtual page is not accessible for the process.* The kernel treats this like any illegal memory access.

The kernel must also ensure that *Inv2* continues to hold when pages are remapped. The simplest way to do this is by invalidating a virtual-to-physical proxy mapping whenever the mapping of the associated virtual memory page is changed.

Maintaining *Inv3*: Register Consistency

The operating system cannot remap any physical page that is involved in a pending transfer, because doing so would cause data to be transferred to or from an incorrect virtual address. Since transfers are started without kernel involvement, the kernel does not get a chance to "pin" the source pages into physical memory.

Invariant *Inv3* makes sure that pages involved in a transfer are never remapped. To maintain *Inv3*, the kernel much check before remapping a page to make sure that that page's address is not in the hardware's source or destination registers (The kernel reads the two registers to perform the check.). If the page is indeed in one of the registers, the kernel must either find another page to remap, or wait until the transfer finishes. If the hardware is in the *DestLoaded* state, the kernel may also cause an `Inval` event in order to clear the destination register.

Although this scheme has the same effect as page pinning, it is much faster. Pinning requires kernel action to change the page table on every DMA, while this mechanism does not, optimizing the common case. The inconvenience imposed by this mechanism is small, since the kernel usually has several pages to choose from when looking for a page to remap. In addition, remapped pages are usually those which have not been accessed for a long time, and such pages are unlikely to be used for DMA.

4.4.9 Summary of UDMA

A deliberate update UDMA transfer requires a mapping step followed by an independent transfer initiation step. The mapping step can be performed at any time prior to the transfer initiation, and provides most of the essential operating system services

that add overhead to traditional DMA transfer initiation. Protection is enforced because the mapping step is performed by the operating system or other trusted code, and transfers cannot be performed without proper mappings. The mapping itself provides address translation and user-level access to the UDMA hardware. Page pinning is unnecessary because transfers cannot cross page boundaries and access to the source and destination registers of the DMA controller is allowed.

The UDMA hardware is memory-mapped into the physical proxy spaces in order to allow processes to access it in a restricted manner from user level. The two-instruction transfer initiation sequence is the most efficient means of specifying two addresses and a transfer size to the hardware, and obtaining confirmation and status from it.

At the heart of the UDMA hardware is a traditional DMA controller. UDMA is simply a safe and efficient mechanism for loading the registers of this controller in order to start a DMA transfer. The transfer itself is no different from a traditional transfer.

4.4.10 Deliberate Update Pros and Cons

Deliberate update shares many of the advantages of automatic update including low latency, statelessness with respect to context, and zero-copy communication. Low latency is achieved through the user-level transfer initiation interface, which does not require a context switch to the operating system in order to initiate a data transfer.

Zero-copy communication is supported since ordinary memory can be used as send and receive buffers. The network interface copies data directly from memory to the network on the sending node, and from the network to memory on the receiving node, without CPU intervention.

Unlike automatic update, deliberate update communication provides the ability to specify the receive buffer as well as the source region at the time of a transfer initiation. This is because receive buffers are not bound to source regions by a physical memory mapping. Instead, the network interface maintains mappings to

many potential destinations, and the selection of which source and which destination to use is done at the time of the send, from user level.

Furthermore, deliberate update has no alignment restrictions between source regions and receive buffers. However, an individual deliberate update cannot cross a source or destination page boundary, so large, unaligned transfers require two deliberate updates per page.

A disadvantage of deliberate update when compared with automatic update is its latency, especially for small transfers. This is because deliberate update requires the two-instruction transfer initiation sequence followed by a DMA data transfer using the relatively slow expansion bus. A detailed performance comparison between these two transfer strategies is presented in Chapter 5.

4.5 Implementation of Notifications

VMMC notifications (Section 2.1.3) are supported through a hardware interrupt mechanism, which is controlled by both senders and receivers. Senders request interrupts and receivers block or unblock them.

A sender requests a notification interrupt by sending an ordinary packet with an active notification request bit. Such a *notification packet* can be generated using either automatic update or deliberate update. Automatic update notifications are supported through a notification request bit in each OPT entry. Any automatic updates performed to a page with the request bit set will generate notification packets.

Deliberate update notifications are supported through a notification request bit included with each `STORE` to destination proxy space. That is, the stored value contains a single-bit field (in addition to the transfer size) which, when set, causes the resulting deliberate update packet to carry a notification request.

A receiver blocks or unblocks notifications using the incoming page table. Every entry in the IPT contains a notification enable bit which, when set, allows notifications for the corresponding physical page. A notification is only generated when a notification packet has been received on a notification-enabled page.

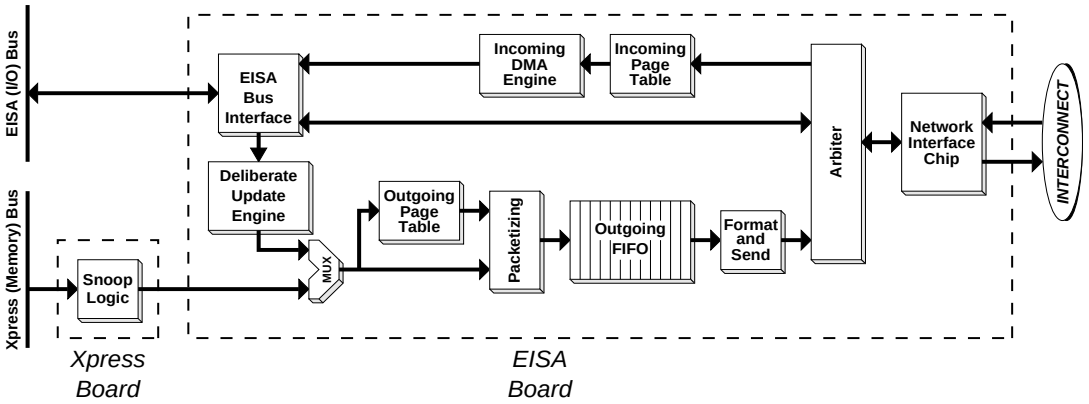


Figure 22: Datapath of the SHRIMP network interface.

A notification interrupt is generated *after* the notification packet data has been transferred to memory. The incoming datapath is frozen until the interrupt has been cleared, so that the interrupt handler has a chance to obtain the last word of the packet and its address. These values are eventually passed to the user-level notification handler, as described in Section 2.2.7.

Each entry in the IPT contains two interrupt bits: the notification enable bit and the receive enable bit (described in Section 4.3.1). The only difference between these two bits is whether they generate an interrupt before or after the packet data has been transferred to memory. In either case, they both freeze the incoming datapath until the interrupt has been serviced.

4.6 SHRIMP Network Interface Datapath

Figure 22 is a block diagram of the SHRIMP network interface, and is identical to Figure 14. The network interface hardware consists of two printed circuit boards, since it connects to both the Xpress memory bus (through the memory expansion connector) and the EISA expansion bus. The Xpress card is extremely simple; the EISA card contains almost all of the logic.

The network interface has two principal datapaths: outgoing and incoming. Outgoing data comes either from the memory bus Snoop Logic (automatic update) or from the EISA bus under control of the Deliberate Update Engine, and incoming

data comes from the Interconnect. The Network Interface Chip (NIC) is an Intel component that was designed to interface a router on the backplane to a bi-directional, 64-bit processor bus [Tra93]. Therefore, the Arbiter is needed to share the NIC's bi-directional processor port between outgoing and incoming transfer.

Consider, first, the outgoing automatic update datapath. After the address and data of the automatic update pass through the multiplexer, the page number portion from the address is used to directly index into the Outgoing Page Table, which maintains bindings to remote destination pages. If an automatic update binding exists, then the Packetizing hardware uses the destination page pointer from the OPT together with the offset portion of the automatic update address to form a packet header in the Outgoing FIFO. The data is appended to this header to form a packet.

If the indexed page has been configured for combining in the OPT entry, then the packet is not immediately sent. Rather, it is buffered in the Outgoing FIFO and if the next automatic update address is consecutive (and greater), the new data is simply appended to the existing packet. If not, a new packet is started. Both standard and latency-bounded combining, as described in Section 4.3.2, are supported.

The outgoing deliberate update datapath is identical to the automatic update datapath, except for the source of the address and data. The Deliberate Update Engine interprets the two-access transfer initiation sequence described in Section 4.4.6, and performs DMA through the EISA bus to read the source data from main memory. The data is passed through the multiplexer along with an address derived from the destination physical proxy reference. The page number portion of this address is used as an index into the OPT.

The incoming datapath is driven by the Incoming DMA Engine, which transfers packet data from the NIC to main memory over the EISA bus using the destination base address contained in the packet header. Before performing the transfer, however, the destination address is used to index directly into the Incoming Page Table to determine if the specified page is enabled to receive data.

The EISA Bus Interface contains logic for mastering the EISA bus and responding as a slave. Mastering capability is shared between the Deliberate Update Engine and

the Incoming DMA Engine, and the slave circuitry responds to page table accesses, local register accesses, and proxy memory references.

4.6.1 Arbitration

The network interface must arbitrate between outgoing and incoming data traffic at two points: the NIC and the EISA bus. At the EISA bus, local arbitration is required between the Deliberate Update DMA Engine and the Incoming DMA Engine to determine which one is permitted to master the bus when it is granted.

Arbitration at the NIC is required because the part has only one bi-directional port for sending and retrieving data. Ideally, a SHRIMP-specific version of the NIC would have two separate 32-bit ports, one for sending and the other for receiving data, obviating the need for an external arbiter. The existing NIC was used in the prototype design because it greatly simplified the interface to the backplane.

The current arbitration policy at both the NIC and the EISA bus gives absolute priority to incoming transfers as a simple way to avoid system deadlock. Results of analytical and simulation-based studies have shown this to be an excellent policy for many communication patterns on a SHRIMP system [FFM96].

A significant architectural ramification of the arbitration policy at the NIC is the inclusion of the Outgoing FIFO, which is required in order to buffer automatic updates when the NIC is not available. If the NIC had separate ports for outgoing and incoming data, then no arbitration would be required and its internal FIFO could be used for automatic update buffering and combining, although a large external FIFO might still be desirable.

The current network interface design stages packets in the Outgoing FIFO and transfers them to the NIC at a rate of 320 Mbytes/sec when they are complete. The NIC does not stage packets, but begins to inject them into the network immediately. Therefore, the network is essentially driven at the rate at which data is transferred to the NIC. If packets were not staged in the Outgoing FIFO, then the network would be driven at EISA bus speed (approximately 33 Mbytes/sec peak) for deliberate update transfers, and Xpress bus copy speed (approximately 40 Mbytes/sec) for automatic

update transfers. Therefore, staging packets in the Outgoing FIFO leads to rapid bursts of network traffic, while not staging would lead to more network utilization.

The tradeoff between these two approaches is between parallelism and network utilization. Staging results in less parallelism between the sender and receiver, and higher packet latency. However, the bursts of traffic utilize the network for less time and allow other packets from other nodes through. Staging probably benefits applications that offer a high network load, while not staging benefits those with less interprocessor communication. Since neither is predictably better, staging was implemented in the current design because it is simpler.

Because of the absolute priority given to incoming transfers, an outgoing transfer can be interrupted by a received packet before it completes. The current design does not break the outgoing packet when this occurs but simply surrenders the NIC, leaving the transfer incomplete. This has the drawback of holding down a path in the wormhole-routed network until the NIC is re-acquired, possibly blocking other transfers since routing is oblivious.

Packet breaking was not implemented in the Format and Send hardware because it requires a more complex design. In addition, there is the potential drawback of more receiver overhead when packets are divided. Experience with the system will help determine the impact of this decision.

The arbitration policy at the NIC could be modified to alleviate the network holding problem by re-arbitrating only between transfers, so that individual outgoing transfers are never interrupted. Such a policy could lead to system deadlock if caution is not exercised. For instance, if full-page (4 Kbyte) transfers are allowed, then a circular deadlock dependency could be formed because each NIC only has 2 Kbytes of FIFO buffer space for incoming transfers. Once the FIFOs became full, the flow control mechanisms would stop incoming transfers at each receiver, and eventually outgoing transfers at each sender.

A scheme for eliminating the arbiter at the NIC altogether would be to interleave outgoing and incoming data transfers on a quadword-by-quadword basis. That is, treat the processor port of the NIC like a bus and perform individual read and write transactions on it. The NIC is driven by a 40 MHz oscillator and is completely

synchronous with the outgoing datapath. The processor port requires two clock cycles (50 ns) to begin a read or write burst, and will produce or accept eight bytes of data on each subsequent clock cycle. Since a single clock is required for bus turnaround, the part can interleave single read and write transactions every eight cycles (200 ns).

Since the incoming data rate can never exceed that of an EISA bus burst (33 Mbytes/sec), eight bytes of incoming data are required every 250 ns (10 NIC clock cycles). Therefore, it is feasible to perform three write transactions (24 bytes of outgoing data) between every two read transactions, effectively interleaving bidirectional data transfers and eliminating the need for an arbiter. The reason this approach was not used in the current design is the added complexity.

4.6.2 Flow Control

Flow control is provided along the entire data communication path in order to insure that no data is ever lost due to lack of downstream buffering capacity. The flow control responsibility is shared between the commodity hardware, the network interface hardware, and the system software. There are several points in the datapath between a sender and receiver where flow control is required:

1. Between the Deliberate Update Engine and the Outgoing FIFO (deliberate update), or between the CPU and the Outgoing FIFO (automatic update).
2. Between the Outgoing FIFO and the sending NIC.
3. Between the sending NIC and a router on the backplane.
4. Between each pair of routers on the backplane.
5. Between a router on the backplane and the receiving NIC.
6. Between the receiving NIC and the receiver's memory.

Because all of the flow control mechanisms operate by "backing up", the following description traces these mechanisms backwards from a receiving node to a sending node.

The data transfer rate from the network interface to main memory is determined by the memory system, which can cause the EISA bus master to wait when the memory is not ready to receive new data. For instance, cache invalidations and refresh cycles can interrupt block transfers for several cycles. This type of flow control is typical on I/O buses and is fully supported by the network interface.

Flow control between the NIC and main memory is simple: when the memory is not available, the Incoming DMA Engine simply stops reading new data from the NIC. The NIC has an internal FIFO to buffer incoming data from the network until it is read by the DMA engine. If the FIFO becomes full, the NIC withholds acknowledgements to the network until space is available.

The routing network implements flow control by providing an acknowledgement for every flit transmitted on every channel, where a channel is a connection between any two routers or between a NIC and a router. In interlocked mode, each flit must be acknowledged before another can be transmitted, while in streaming mode up to four flits can be outstanding at any given time before an acknowledgement is required. Each router has the capability to buffer up to four flits on each incoming channel, and simply withholds incoming acknowledgements when the outgoing channel is blocked (signified by no outgoing acknowledgements). This simple protocol guarantees that data is never lost by the routing network.

The NIC has a second internal FIFO, which buffers outgoing data before injection into the network. As long as this FIFO is not full, data can be written to the sending NIC by the Format and Send hardware even when the network has stopped accepting data. When the outgoing NIC FIFO becomes full, a flow control signal stops the Format and Send hardware, which in turn stops reading data from the Outgoing FIFO.

The Outgoing FIFO has a programmable "almost full" threshold, which triggers software flow control by causing a flow control interrupt to the node CPU when it is exceeded. This interrupt is primarily responsible for halting automatic updates to prevent them from overflowing the Outgoing FIFO. When the Linux kernel receives the interrupt, it de-schedules all applications that have automatic update bindings and monitors the amount of data in the Outgoing FIFO. Only when the FIFO has drained

sufficiently are the applications re-scheduled and the network interface returned to normal operation.

The flow control interrupt also halts the Deliberate Update DMA Engine if it is active, causing it to relinquish control of the EISA bus and freeze until the interrupt has been acknowledged and the FIFO has drained below the almost-full threshold.

All of the flow control mechanisms are transparent to user-level applications, which experience nothing worse than a drop in bandwidth or a rise in latency when the mechanisms are engaged. Data is never lost due to contention in the network or a slow receiver.

4.6.3 Support for Performance Monitoring

The SHRIMP network interface provides several features to support external performance monitoring. The features that were considered most important are accurate timing of packet latency, providing significant packet characteristics to the performance monitor, and programmability. These features are utilized in the design of the SHRIMP performance monitor, described in [MCM96].

Accurate latency timing is useful for calculating packet bandwidths as well as latencies. SHRIMP takes advantage of a global 10 MHz clock signal, provided by the Paragon backplane, to maintain a 44-bit timestamp counter on each network interface board. The backplane also provides a clock reset signal to synchronize all of the individual counters.

The network interface includes the current timestamp value in the header of every packet started in the Outgoing FIFO, thereby marking each packet, as early as possible, with the time of its creation. The network interface does not mark the time that packets are received, however. Instead, it provides the global clock and clock reset signals to the performance monitor so that it can maintain its own timestamp counter.

The time that a received packet was created is one of the significant characteristics that are provided to the performance monitor. Other significant characteristics are the destination address and size of each packet. These characteristics are provided to

the performance monitor by simply allowing it to synchronously read the incoming datapath at the NIC. Since the packet format is fixed, the performance monitor can extract all the information it needs.

For example a packet's total latency is simply the difference between the timestamp value carried in the packet and the value of the performance monitor's timestamp counter when the *last* word of the packet is read from the NIC (and transferred to memory)[3]. Bandwidth is easily calculated using the latencies and sizes of packets.

Programmability of performance monitor support is provided in two ways. First, packet timestamping is optional and can be disabled to reduce the size of packets. Although timestamps are always inserted into the Outgoing FIFO, they need not be included in network packets. Second, the IPT includes a 6-bit, writable field in each entry that can be used to classify packets as members of one of 64 different groups. The address of every incoming packet is used to index into the IPT and retrieve the group identifier, which is passed to the external performance monitor along with the packet header.

4.6.4 Un-Supported Features

The SHRIMP network interface hardware was designed to be minimal in that features not considered essential for performance were relegated to software. Many features were considered but not included because of insufficient evidence of their benefit. In this section, the ones that "almost made it" into the implementation are discussed.

The UDMA deliberate update mechanism described in Section 4.4 restricts individual transfers to single pages in order to take advantage of the MMU protection mechanism and avoid the need for hardware scatter-gather capability. Although this leads to a simple implementation, it requires software to perform large transfers as a series of small transfers.

Since the software overhead is so low, multiple transfer initiations are not a problem, but waiting for each transfer to complete before initiating the next could

[3]There are, of course, some fixed latencies at the sender and receiver that must be added to this difference to obtain the true latency.

be. An elegant mechanism for avoiding this potential drawback is UDMA request queueing.

The queueing scheme would simply have the network interface hardware queue UDMA transfer requests, allowing applications to specify multi-page transfers in the same way as the present scheme but in immediate succession. A valid user transfer request would only be refused if the queue was full; otherwise it would be accepted and performed when it reached the head of the queue. To wait for the completion of a multi-page transfer, the application need only check for the completion of the last individual transfer.

Queueing makes it easy to do scatter-gather transfers, and it allows unrelated transfers, perhaps initiated by separate processes, to be outstanding at the same time. The disadvantage of queueing is that it makes it more difficult to check whether a particular page is involved in a pending transfer. To address this problem, the hardware could either keep a counter for each physical memory page of how often that page appears in the queue, or the hardware could support an associative query that searches the queue for a particular page, similar to the present MATCH flag scheme (Section 4.4.7).

Another feature that could have improved the efficiency of polling for UDMA transfer completion was described in Section 3.3.2. The idea is to have the network interface maintain a large integer value in some fixed, cacheable memory location, and return the current value of the integer to the user whenever a transfer initiation is accepted. The user then polls on the fixed memory location (in the cache) until the network interface changes it, which occurs as soon as the transfer is complete.

Another feature that was almost included in the SHRIMP network interface is a method for performing hardware multicast. At one point, the design included a multicast mechanism similar to the virtual sharing rings of Galactica Net [JJT93], where the remote physical page pointers in the NIPT could be used to chain multicast pages together. The additional complexity imposed by this mechanism includes a method for forwarding multicast packets on the network interface, and for removing them when they returned to the source node. The multicast mechanism was eventually

removed because a potentially serious deadlock problem was discovered, and there was not enough evidence for its superiority over software multicast.

A fairly simple feature that was strongly considered was support for user-level competitive spinning [KLMO91]. Competitive spinning is used by synchronization code to avoid polling indefinitely for some event, such as a message arrival. At some point, a process that is spinning should block and switch to an interrupt strategy so that other processes can run. For example, in a user-level, multi-threaded environment, competitive spinning can be used to rapidly block a waiting thread and switch to some other thread.

Competitive spinning requires an efficient mechanism for giving applications control over which notifications they receive. In the current implementation, a system call is required because the IPT is memory-mapped and only accessible by the system. Competitive spinning could be implemented more efficiently if applications could set or clear specific receive interrupt enable bits in the IPT from user-level.

The long list of other features that were briefly considered includes remote atomic operations, remote reads, and hardware barriers. In every case, these features were not included because there was little evidence for their advantage over software mechanisms in terms of actual application performance. The SHRIMP network interface simply concentrates on moving data as fast as possible.

4.7 Fault Tolerance

The prototype SHRIMP system was not designed to be particularly fault tolerant because it operates in a cleanroom environment, like a multicomputer. Because errors are assumed to be rare and hardware error correction is complex, software is given the optional responsibility for error correction. The hardware provides error detection, however, to maintain the integrity and correct delivery of network packets.

Every packet begins with a routing header, which is stripped by the network as the packet is transmitted. In order to detect misrouted packets, the SHRIMP network interface includes the intended destination within each packet and verifies it at the

receiver. This is possible because absolute cartesian coordinates uniquely identify the position of each node in the two-dimensional mesh network.

The NIC provides every packet with CRC protection, which covers the entire packet, except for the routing header. The CRC-32 code used by the NIC detects all single and double bit errors, all errors with an odd number of bits, all error bursts of 32 bits or less, and 99.997% of error bursts exceeding 32 bits in length [Tra93]. For architectural reasons, the NIC actually appends two CRC words to each packet, increasing the packet length by eight bytes.

Byte parity is generated and checked by each individual transmission channel (between NICs and routers) in order to detect faulty links to the network, or within the network. If a NIC detects an error, it causes an interrupt to the host CPU. If a router detects an error, it causes an interrupt to the node that hosts the network adaptor board (Section 4.2.1).

Besides network errors, the only other fault detected by the network interface hardware is an Outgoing FIFO overflow, which causes an interrupt to the CPU. In some situations, specifying a deliberate update transfer that crosses the boundaries of the source or destination pages could be considered an error, so an optional interrupt is provided to notify the host when this occurs.

4.8 Conclusions

Although the design described in this chapter is specific to available devices, the architecture and design methodology apply equally well to other situations. After all, low-overhead communication is desirable on all DMPPs.

The prototype design satisfies the goals and constraints outlined in Section 4.1. Protected, low overhead communication is provided by both the eager-sharing automatic update mechanism and the UDMA-based deliberate update mechanism. Because the network interface is context-independent, general multiprogramming is allowed on each node without requiring coordination between the nodes.

The memory bus bandwidth goal is met by both update strategies. The automatic update datapath can keep up with the maximum write bandwidth of the memory

bus, even when creating an individual packet for every write. The deliberate update DMA operates on the memory bus through the EISA expansion bus, so its maximum bandwidth requirement is that of the EISA bus, which it satisfies.

The implementation of the SHRIMP network interface is board-level and utilizes primarily commodity components. All of the control logic and some of the datapath is implemented in programmable logic (PLDs and FPGAs). The only non-commodity component on the board is the Intel NIC chip, which is required for interfacing with the Paragon backplane. However, the network interface design also takes advantage of the FIFOs within the NIC. So, replacing the Paragon backplane and the NIC with some other network would require adding at least one FIFO (incoming) to the design.

The rest of the system is truly commodity, including most of the software. Although minor kernel modifications are required, the operating system on each node is regular PC Linux.

Although the design of the SHRIMP network interface could not be called simple, the architecture certainly could. Design simplicity, however, is a relative term based largely upon manpower and the availability of sophisticated design tools. When compared with the complexity of a PC motherboard, for example, the SHRIMP network interface is a very simple design.

Chapter 5

Prototype Performance

The primary goal of the SHRIMP project was to reduce or eliminate the overhead of communication so that user-level applications could achieve the latency and bandwidth of the underlying communication hardware. This chapter begins with a presentation of some actual measurements to show how well the goal was achieved. The rest of the chapter presents more detailed measurements and discussions to demonstrate performance characteristics of the prototype system, and reveal the strengths and weaknesses of the network interface design.

Low Latency

The SHRIMP network interface has proved successful in eliminating most of the software overhead of communication. An optimized "ping-pong" program[1] that transfers a single integer repeatedly back and forth between two nodes using automatic update, achieves a roundtrip time of just 7.42 μs. For this program, the time to run the code could be considered communication software overhead because the send and receive operations are explicit. In general, however, automatic update can be a side-effect of any main memory write, and polling for received data is not explicitly required.

Hardware-level measurement of a single-word, automatic update transfer between the two nodes reveals a latency of about 2.5 μs from the snoop on the sending node

[1]The program on each node simply polls for the arrival of the integer, and immediately sends it back out after it arrives. All caches were disabled.

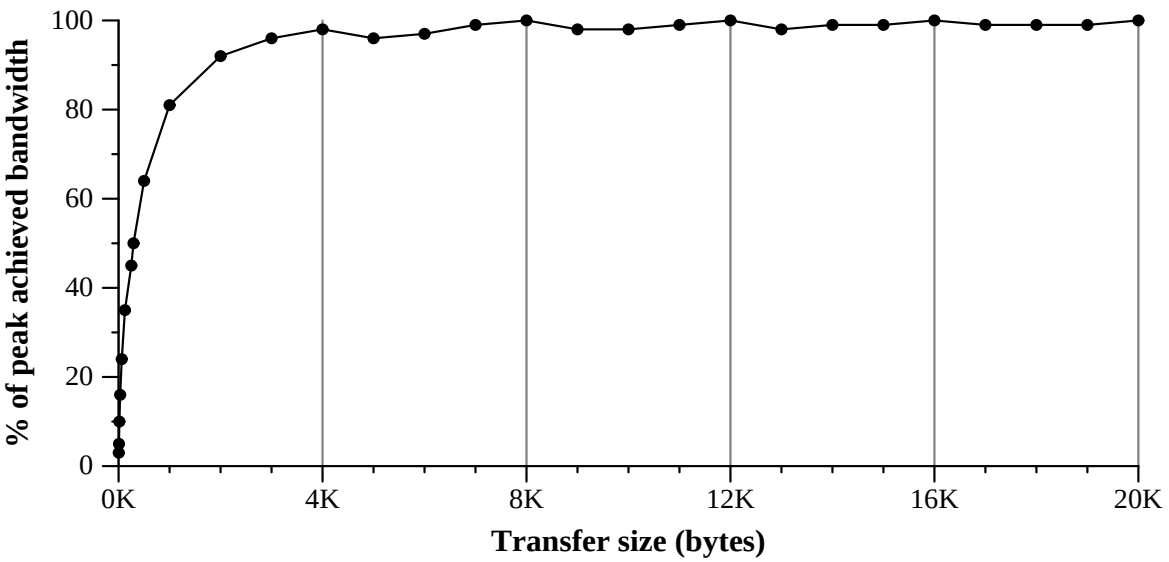


Figure 23: Bandwidth of deliberate update transfers as a percentage of the maximum measured bandwidth.

until the memory write on the receiving node is complete. This latency depends on the response time of the EISA bus on the receiving node, which is most significantly affected by CPU activity and memory refresh.

Hardware-level measurement of the ping-pong program shows about 0.93 μs of activity from the time the integer is received until it is written back to memory (to send it). So, the expected roundtrip time is $2 \times 2.5\mu s + 2 \times 0.93\mu s = 6.86\ \mu s$. The 0.56 μs difference between this result and the measured time has not been explained, but is probably due to periodic system events such as refresh.

High Bandwidth

Software overhead affects bandwidth as well as latency, especially for small transfers. For large transfers—typically performed with DMA—the startup overhead is amortized and applications can achieve the full hardware bandwidth. In order to determine how much of the hardware bandwidth was available to SHRIMP applications, the simple ping-pong program was extended to send a block of data back and forth between two nodes using deliberate update.

The curve in Figure 23 plots the ping-pong bandwidth of several transfer sizes as a percentage of the maximum bandwidth measured for large transfers. The bandwidth was calculated for each transfer size by dividing the running time of the program by the total number of bytes transmitted over all iterations.

The rapid rise of the curve highlights the low cost of initiating deliberate update transfers. The bandwidth reaches 50% of the maximum at a transfer size of only 300 bytes. A single-page (4K byte) transfer achieves 98% of the maximum bandwidth, and a two-page transfer reaches the maximum. For comparison, the Intel Paragon—a traditional message-passing machine—reaches 50% of it's peak bandwidth at a message size of about 6 Kbytes, and 100% at about 4 Megabytes[2].

An individual deliberate update can transfer at most a page, so transfer sizes in Figure 23 that exceed 4 Kbytes require multiple transfer initiations. The effect of this can be seen in the slight dip of the curve after each transfer size that is an even multiple of the page size (indicated by the vertical lines). The transfers that are an even multiple of a page in size amortize the total initiation overhead best.

5.1 Network Interface Hardware Parameters

The remainder of this chapter presents various measurements and analyses of the SHRIMP network interface and the prototype system. A good deal of the experimentation is devoted to determining the effects of varying a number of network interface hardware parameters that were designed to be programmable because their optimal values were not known. These parameters are:

- *EISA bus possession* - The number of EISA bus clocks that the network interface will continue to master the bus after it has been revoked, before relinquishing control and re-arbitrating. Possible values are 12 clocks, 60 clocks, or "unlimited." The maximum value—unlimited—allows the bus to be held for the entire duration of any transfer, which is at most a page.

[2]Measurements were performed using csend and crecv for small messages, and irecv for large messages

- *Combining delay* - The length of time to delay launching a combined automatic update packet while waiting for a potentially consecutive update. The possible values range from 75 ns to 12800 ns in increments of 25 ns.

- *Deliberate update packet size* - Specifies aligned address boundaries that individual deliberate update transfers cannot cross, thereby constraining the maximum packet size. Possible values are 256 bytes, 512 bytes, 1024 bytes, or 4096 bytes (page boundaries).

- *Automatic update packet size* - Specifies aligned address boundaries that individual combined automatic update transfers cannot cross, thereby constraining the maximum packet size. Possible values are 256 bytes, 512 bytes, 1024 bytes, 2048 bytes, or 4096 bytes (page boundaries).

- *Timestamping* - Specifies whether packets carry a global timestamp or not, thereby affecting packet size.

Micro-benchmarks were used to characterize latency and bandwidth in relation to these programmable parameters in an effort to determine their optimal settings and gain more understanding of the network interface behavior.

5.2 Experimental Prototype Configuration

The configuration of the SHRIMP prototype used for the performance studies presented in this chapter was described in Section 4.2.1 and consists of two nodes positioned at opposite corners of a 4-by-4 mesh, as shown in Figure 12. Although a 4-node machine has been constructed, its reliability is not yet sufficient for extensive experimentation.

Because the two nodes are positioned at opposite corners of the mesh, latency results can be considered worst-case for a 16-node machine in the absence of network contention.

With only two nodes, it is impossible to generate network contention, no matter where the nodes are positioned in the mesh. Therefore, studies of network interface

behavior in the presence of contention will be postponed until larger systems become available.

For all the experiments in this chapter, the SHRIMP routing backplane was configured to use streaming mode (Section 4.2.1) for the highest possible bandwidth.

5.3 Hardware Datapath Latency

The results in this section were derived from a detailed examination of the network interface and network hardware performed with a Hewlett Packard 16500B logic analyzer [Hew94]. Each step in a single word data transfer through the outgoing network interface datapath, the routing backplane, and the incoming network interface datapath is described. All measurements were made with timestamping disabled, and CRC checking between NICs enabled, yielding a network packet size of 32 bytes.

The latencies presented in this section are derived from single, typical traces, and are not averages. The latency that varies the most among traces (assuming no network contention) is the response time of the EISA bus to a mastering request.

The majority of the hardware datapath is shared by automatic update and deliberate update transfers. This section begins with a description of an automatic update transfer because it is more straightforward.

5.3.1 Outgoing Automatic Update Send

The lowest latency is achieved with a single-word, automatic update transfer as shown in the timeline of Figure 24. The portion labeled "SENDER" shows the steps performed on the sending node, and described in this section. Note that solid lines indicate fixed latencies and dashed lines indicate variable latencies.

The address and data of the write cycle are snooped off of the memory bus and synchronized to a 40 MHz clock (25 ns cycle time). At the first clock cycle, they are passed to the network interface board where the OPT lookup is performed in 2 clock cycles. The translated address, data, OPT information, and timestamp are written to the Outgoing FIFO, requiring 2 more clock cycles. Two cycles after the second FIFO

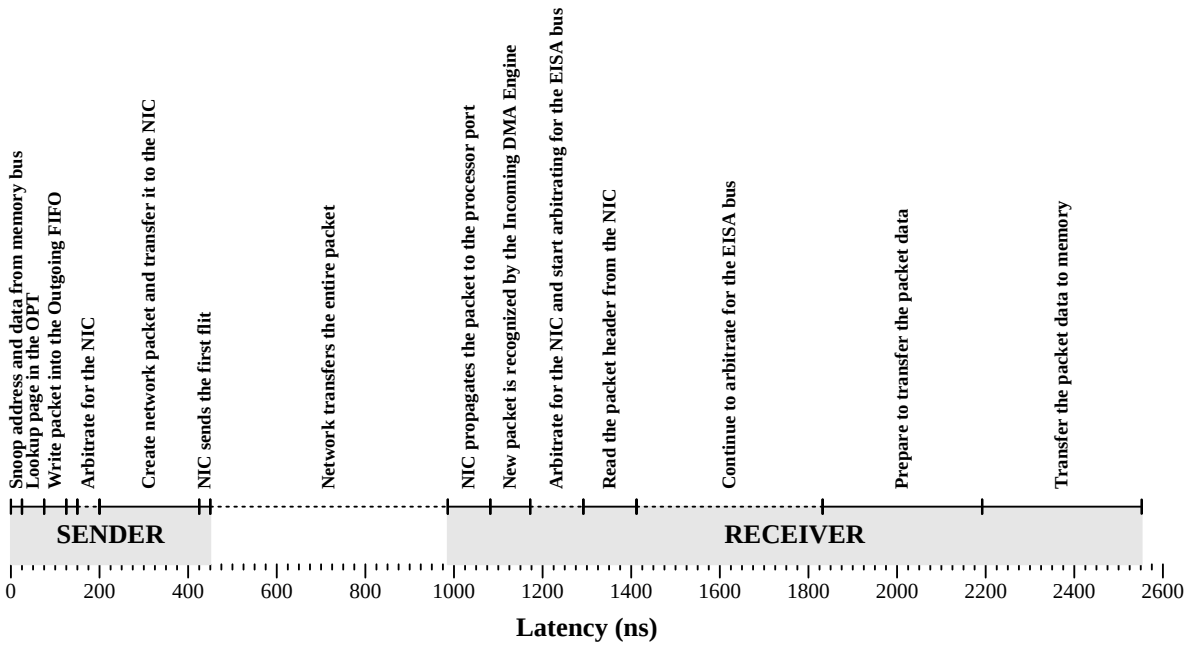


Figure 24: Hardware latency components of a single-word, automatic update transfer. Solid lines indicate fixed latencies and dashed lines indicate variable latencies.

write, the Packetize and Send state machine starts and transfers the packet to the NIC, requiring a total of 11 cycles. One cycle later, the first flit of the packet leaves the NIC. Therefore, the total outgoing latency of the header of a single automatic update transfer is about 450 ns.

5.3.2 Network Transfer

The Intel Paragon network has a peak bandwidth of about 200 Mbytes/sec, but SHRIMP cannot achieve this peak performance because five-foot cables are used to connect the PC nodes to the backplane. This is not a problem because the measured bandwidth between the nodes and the network comfortably exceeds the memory bus bandwidth of the nodes.

Both modes of backplane operation described in Section 4.2.1—interlocked and streaming—were measured for a one-word packet with timestamping disabled and CRC checking enabled.

Interlocked Mode

Interlocked mode has a bandwidth of 37.27 Mbytes/sec, measured at the sending NIC. The time from a flit leaving the NIC to its corresponding handshake acknowledgement arriving from the backplane is 44 ns, and 804 ns are required to transfer a 32-byte packet to the network.

On a 4-by-4 mesh, the corner-to-corner latency of a flit from the sending NIC to the receiving NIC is 352 ns, so a 32-byte packet has a corner-to-corner interlocked-mode latency of about 1.156 μs.

Streaming Mode

Recall that streaming mode is a four-slack protocol, meaning that up to four flits can be sent by the NIC before the first acknowledgement is required. Streaming mode has a bandwidth of 113.21 Mbytes/sec. The time from a flit leaving the NIC to its corresponding acknowledgement arriving is 56 ns, but only 248 ns are required to transfer a 32-byte packet to the network.

On a 4-by-4 mesh, the corner-to-corner latency of a flit from the sending NIC to the receiving NIC is 288 ns, so a 32-byte packet has a corner-to-corner streaming-mode latency of about 536 ns, as shown in Figure 24.

5.3.3 Incoming Receive

The incoming datapath operates synchronous to the EISA bus clock doubled to a frequency of 16.67 MHz (60 ns cycle time), and is labeled "RECEIVER" in Figure 24. The NIC signals reception of a small packet at the processor port 96 ns after the *last* flit is received. Large packets can trigger a receive FIFO threshold signal before they are completely received, but the threshold cannot be set low enough to trigger on portions of small packets.

The data reception signal from the NIC is synchronized to the double-clock and starts the incoming DMA engine one cycle later. The DMA engine immediately begins arbitrating for the EISA bus and the NIC data bus simultaneously. When

the NIC data bus is obtained (often 2 double-clock cycles later), the packet header is read, requiring another 2 double-clock cycles.

The EISA bus is usually granted about 3 or 4 EISA bus clock cycles after the request, but because there is a separate state machine to handle EISA bus protocol, the response time experienced by the deliberate update engine is about 5.5 bus clocks. This is about 7 double-clock cycles after the header has been read from the NIC.

If the packet is *not* accepted for transfer to memory, then the EISA bus is relinquished as soon as it has been granted and an interrupt is generated to the node CPU. Otherwise, a pipelined transfer is started from the NIC to main memory. The startup overhead of this transfer is 6 double-clock cycles, and the transfer of the first word requires 3 EISA bus clock cycles (6 double-clock cycles) because a single wait state is inserted by the PC's memory system.

Therefore, the total latency of the incoming datapath for a single word, from the reception of the last flit of the packet until the data is in memory, is about 1.566 μs.

5.3.4 Outgoing Deliberate Update Send

A deliberate update transfer is started with the two-instruction transfer initiation sequence described in Section 4.4.6, and causes the network interface to fetch the data from memory using DMA as shown in Figure 25.

Because deliberate update and automatic update data are handled the same by the network interface, they share the outgoing datapath, and every step in Figure 24 except the very first snoop operation applies to a single-word deliberate update transfer as well. The snoop operation is basically replaced by the steps described in this section. The long vertical line in Figure 25 indicates where the deliberate update transfer enters the shared outgoing datapath of Figure 24.

The store-load deliberate update transfer initiation sequence is performed across the EISA bus since the proxy spaces are mapped to the network interface card. Each reference requires 3 EISA bus clocks (120 ns cycle time) because the network interface inserts a single wait state on all references. The EISA bus requires a single clock cycle

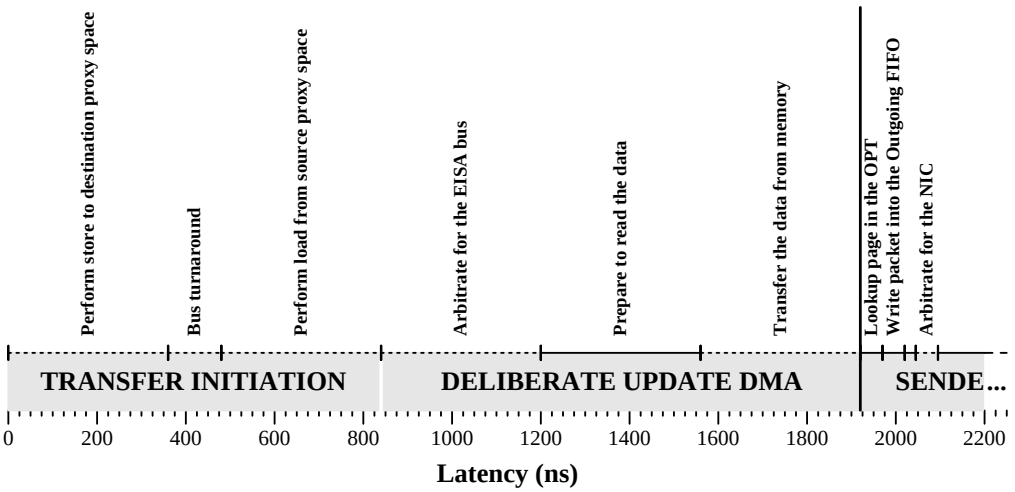


Figure 25: Latency components of a single-word, deliberate update transfer.

to turn around from the store to the load, so the transfer initiation sequence can be performed in 7 EISA bus clock cycles, or 840 ns.

After the transfer initiation, the Deliberate Update DMA Engine requests the EISA bus and usually must wait 3 or 4 EISA bus clock cycles before receiving a grant. The actual deliberate update transfer begins 3 bus clocks later due to startup overhead, and requires 3 bus clocks to complete because the PC's memory system always inserts a single wait state on the first transfer of a burst.

From this point on, the address of the transfer and the data read from memory enter the same pipeline shown in Figure 24, beginning with the OPT lookup. Therefore, the total outgoing latency of the header of a single deliberate update transfer (including the transfer initiation) is about 2.35 μs, depending on the response time of the EISA bus.

5.3.5 Summary

The end-to-end latency of a single-word, automatic update transfer is the sum of the outgoing header latency from Section 5.3.1, the streaming mode packet latency from Section 5.3.2, and the incoming packet latency from Section 5.3.3, or about 2.5 μs. Similarly, the end-to-end latency of a single-word, deliberate update transfer, including the initiation time, is about 4.5 μs.

5.4 Automatic Update Latency

A common metric for minimum latency is the roundtrip time of a single word of data in a "ping-pong" program, which repeatedly transfers a single integer back and forth between two nodes. To measure the minimum latency, the code shown in Figure 26 was used.

Node 1:

```
1  start_time = current_time;
2  for(i = 0; i < ITERATIONS; i++) {
3       buffer[0] = 1;
4       while(buffer[0] != 0) ;
5  }
6  end_time = current_time;
7  latency = (end_time - start_time) / ITERATIONS;
```

Node 2:

```
1  for(i=0; i < ITERATIONS; i++) {
2       while(buffer[0] == 0) ;
3       buffer[0] = 0;
4  }
```

Figure 26: The inner-loop code for optimized ping-pong.

Each node maps its buffer to the other's in automatic update mode with no combining, which has the lowest latency. The data sent by each node also acts as the

receive flag, so the send operation doubles as the flag reset. The interesting tradeoff is whether to cache `buffer` as write-through, or not cache it at all.

Application-level measurements of the ping-pong program reveal roundtrip latencies of 7.57 μs with `buffer` cached write-through, and 7.42 μs with `buffer` uncached. In order to understand the difference between these two measurements, hardware traces of the system behavior were analyzed.

When `buffer` is cached write-through, the `while` loop spins entirely in the cache, freeing up the memory bus for the incoming transfer. When the integer arrives, the cache line[3] containing `buffer[0]` is invalidated, leading to a read miss on the following iteration of the `while` loop. The read miss causes a cache line fill that takes place about 450 ns after the incoming transfer and takes about 450 ns. About 300 ns later the update to `buffer[0]` occurs, causing the next automatic update transfer. Therefore, the total time required to receive a word and send one back out with `buffer` cached write-through is about 1.2 μs.

When `buffer` is not cached at all, the `while` loop polls main memory and causes read cycles every 300 ns, each lasting about 130 ns. The incoming transfer must compete with this polling for use of the memory bus. Once it succeeds, the next iteration of the loop finds the update in memory, about 460 ns after it occurs. About 470 ns later, the update to `buffer[0]` occurs, causing the next automatic update transfer. Therefore, the total time required to receive a word and send one back out with buffer uncached is about 930 ns.

These two analyses suggest an expected application-level difference in roundtrip times of 540 ns, although only a 150 ns difference was measured. This discrepancy is probably caused by the interaction between the CPU and the network interface on the memory bus, particularly when `buffer` is not cached. The latency of incoming DMA and the time between the DMA and the next memory polling cycle are dependent upon bus availability, and vary on each loop iteration.

For this simple program, not caching `buffer` is better because it avoids the cost of the cache line fill. Since there are no other applications running and the ping-pong application has no other work to do, polling on the memory bus is harmless.

[3]Both the first- and second-level caches have a line size of 32 bytes.

In general, this might not be such a good strategy for polling because of the high memory bus utilization. Say, for example, that an application transfers a large buffer followed by a flag to notify the receiver, and the page containing the flag is not cached. Even if no other applications are running and there is no other work to do until the buffer is received, the constant main memory polling cycles will interfere with the large incoming data transfer, possibly causing an overall delay even though the completion flag is recognized sooner.

To measure this effect, the ping-pong program was modified to perform large transfers, as described in Section 5.6. A page containing a receive flag was uncached, and both the sender and receiver used a realistic EISA bus possession time of 60 clocks. The result, for a 16 Kbyte transfer size, was a drop in bandwidth of 2.36 Mbytes/sec, from 21.35 Mbytes/sec to 18.99 Mbytes/sec. The drop occurred because the CPU performed a single polling operation every time the receiving network interface gave up the bus. With the EISA bus possession parameter set to unlimited, the drop in bandwidth was only 0.19 Mbytes/sec, from 23.57 Mbytes/sec to 23.38 Mbytes/sec, because no polling cycles interfered with the individual page transfers.

5.5 Deliberate Update Bandwidth

In order to determine the maximum deliberate update bandwidth of the network interface, a repetitive block copying program was written. One node sent a large volume of data, and the other node simply did nothing, letting its network interface continually receive.

The program shown in Figure 27 was used by the sender. Lines 1 and 2 set up pointers to the proxy addresses used for the load-store transfer initiation sequence. The `attempts` variable records the total number of transfer initiation attempts, and is used in line 13 to compute the average number of attempts per successful transfer initiation. Line 8 performs the store to destination proxy space, and line 9 performs the load from source proxy space as well as testing for successful initiation. The entire program performs `TRANSFERS` successful transfers since the `for` loop terminates after

```
1   dp = destination_proxy_address;
2   sp = source_proxy_address;
3   attempts = 0;
4   start_time = current_time;
5   for(i = 0; i < (TRANSFERS+1); i++) {
6        do {
7             attempts++;
8             *dp = NWORDS_TO_SEND;
9        } while( ((*sp) & 0xF) != 0 );
10  }
11  end_time = current_time;
12  bandwidth = ((4*NWORDS_TO_SEND)*TRANSFERS)/(end_time-start_time);
13  attempts_per_transfer = attempts/TRANSFERS;
```

Figure 27: Block sending program used to measure deliberate update bandwidth.

the last transfer is accepted (but not completed). Line 12 computes the cumulative bandwidth in bytes per second.

5.5.1 Effect of EISA Bus Possession

Recall that the EISA bus possession parameter can be set to 12 clocks, 60 clocks, or "unlimited" (which results in at most full page bursts). The EISA specification states that a bus master should not hold the bus for longer than 64 bus clock cycles after it has been revoked [BCP92]. The 60 clock EISA bus possession parameter option is intended to maximize use of the bus while staying within the specified limit. The four additional clocks are reserved for transfer startup and termination overhead.

The 12-clock EISA bus possession option was provided in order to break deliberate update bursts frequently and allow greater CPU access to the memory during DMA transfers. The hope was that this could potentially increase application throughput, although data transfer bandwidth would be reduced.

Attempts to create synthetic applications for which the 12-clock option outperformed the other options were unsuccessful, however. The 12-clock option results in a considerable increase in hardware overhead due to frequent bus arbitrations

followed by DMA startups. The experiments that were performed indicate that the tradeoff between transfer overhead and CPU progress favors low overhead. Perhaps future studies of real applications will reveal situations that benefit from the 12-clock option.

The unlimited EISA bus possession option allows a transfer to violate the specified 64 clock limit in order to achieve better bandwidth. It was observed that the PC "queues" refresh cycles during a long burst on the memory bus and then "catches up" by performing them in succession when the bus is released. Assuming this does not violate the refresh requirements, retaining the bus for an entire page burst might not have any detrimental effects. It is not clear whether long memory bus bursts are the best strategy for application throughput, however, since CPU memory accesses are blocked.

Table 3 shows how the bandwidth varies with the EISA bus possession parameter on the sender and receiver. For this experiment, the deliberate update packet size was set to full pages (4096 bytes).

Sender possession	Receiver possession	Bandwidth (MB/s)	Attempts/transfer
unlimited	unlimited	28.13	1.98
unlimited	60 clocks	25.31	4.42
unlimited	12 clocks	23.34	7.94
60 clocks	unlimited	21.60	10.40
60 clocks	60 clocks	21.60	10.40
60 clocks	12 clocks	21.59	10.40
12 clocks	unlimited	11.31	51.41
12 clocks	60 clocks	11.31	51.40
12 clocks	12 clocks	11.30	51.38

Table 3: Maximum deliberate update bandwidth vs. EISA bus possession time.

The last column in Table 3 reflects the frequency with which the network interface on the sending node relinquishes the EISA bus. The bus is relinquished either because the EISA bus possession parameter is reached, or because the entire datapath from sender to receiver backs up, and flow control prevents the deliberate update DMA

engine from acquiring the bus and overflowing the Outgoing FIFO. The effect of flow control can be seen in rows one through three of Table 3 where the bandwidth is receiver limited, while the effect of EISA bus possession can be seen in rows four through nine where the bandwidth is sender limited.

The first row in the table shows the absolute maximum bandwidth achieved by deliberate update, but the unlimited EISA bus possession violates the maximum EISA specification of 64 clocks. Bandwidth is limited by the EISA bus, which has a peak bandwidth of 33.33 Mbytes/sec. Realistically, however, this peak could only be achieved by an infinitely long burst. The bandwidth in the first row, with unlimited bus possession, comes very close to the *practical* maximum.

The difference between the measured bandwidth and the theoretical peak was explored in the laboratory by analyzing the behavior of the PC system during a single DMA burst transfer (one loop iteration) in terms of EISA bus clocks:

- 5 clocks to start the burst transfer
- 1024 clocks to transfer the data
- 3 clocks to break bursts on 1024-byte aligned boundaries (1 each time)
- 128 clocks for wait states (one on every address evenly divisible by 32)
- 56 clocks of system activity between transfers (accumulated refreshes and code)

The 3 clocks spent breaking burst transfers are required by the EISA specification, presumably for page-mode DRAM support. The single wait states on 32-byte aligned addresses are undoubtably related to the design of the PC's memory system since the cache line size is 32 bytes, but the precise reason is unknown. Since the EISA bus possession parameter was set to unlimited, the system performs several accumulated refreshes between network interface bursts.

Summing up these individual components yields a total of 1216 clocks to transfer 4 Kbytes, resulting in an expected bandwidth of 28.07 Mbytes/sec. This is very close to the measurement of 28.13 Mbytes/sec.

The last column in the table shows the average number of attempts per transfer, or the average number of times that the loop in lines 6–9 of Figure 27 was executed. This includes the attempt that succeeded.

The first row of Table 3 shows that with the EISA bus possession parameter set to unlimited, there was approximately one unsuccessful transfer initiation attempt for each successful attempt. The unsuccessful attempt probably occurs before the successful deliberate update transfer gains control of the bus because the transfer itself is indivisible when the EISA bus possession is set to unlimited.

The first three rows of Table 3 highlight the receiver's bandwidth for each of the bus possession options. The bandwidth drops as the sender is forced to give up the bus more often, but even in the worst case, the receiver's bandwidth exceeds 23 Mbytes/sec.

The rest of the measurements, with the sender set to 60 and 12 clocks, are sender-limited, as can be seen in the best of these cases (row 4) where the sender is set to 60 clocks and the receiver is unlimited. Here, the bandwidth is still lower than the worst receiver-limited case (row 3) because of the transfer initiation overhead and code activity on the sender, which occupies the memory bus whenever it is relinquished.

The fifth row of Table 3 shows the best bandwidth achieved *without* violating the EISA bus possession specification of 64 clocks. This value is 35% less than the theoretical bus peak, but only 23% less than the practical maximum bandwidth achieved in row 1.

5.5.2 Effect of Deliberate Update Packet Size

The deliberate update packet size parameter is used to divide large deliberate update transfers into smaller packets in order to increase parallelism between the sender and receiver, since packets are staged at the sender. A particular value for this parameter causes deliberate update transfers to be broken into individual packets at addresses evenly divisible by the parameter value.

Although a smaller packet size value results in more packets and more overhead, Figure 28 shows that this can lead to higher bandwidth. The figure shows a 4K-byte

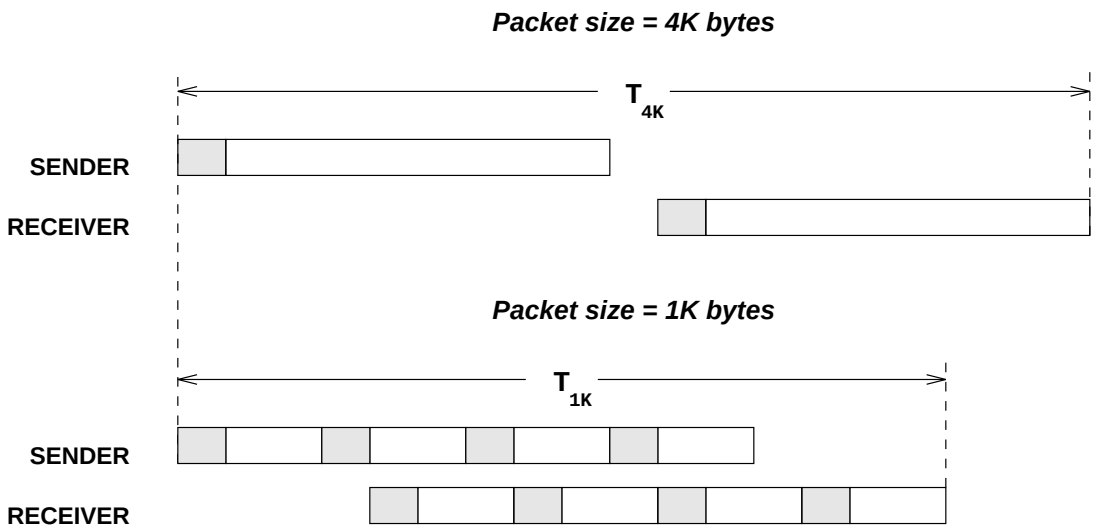


Figure 28: Latency of a 4 Kbyte transfer with packet sizes of 4 Kbytes and 1 Kbyte.

transfer performed as a single transfer, and as four 1K-byte transfers. Each individual transfer consists of some overhead followed by the data burst into the Outgoing FIFO. Packets are staged in the FIFO and only launched when completed. Therefore, parallelism between the sender and receiver is achieved by breaking transfers into smaller packets.

Clearly the situation would be different if Figure 28 showed a larger transfer size (with the same packet sizes). Eventually, the overhead of the smaller packets would accumulate and dominate the savings due to parallelism. Therefore, one can surmise that the optimal packet size depends on the total transfer size.

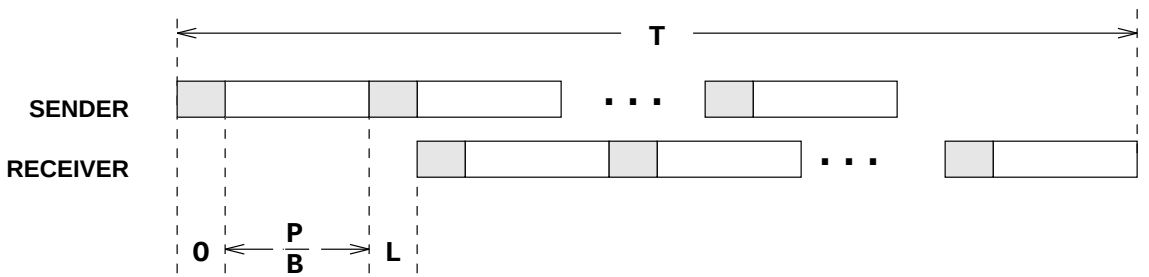


Figure 29: Variable definitions for the latency components of a packet transfer.

The relationship between transfer size and optimal packet size can be examined with a simple mathematical model based upon the situation shown in Figure 28. Figure 29 shows variable assignments where B is the EISA bus bandwidth, P is the packet size parameter in bytes, O is the overhead of a deliberate update transfer, and L is the network latency of a packet transmission. If N is the total number of bytes to transfer, then the time to complete an entire transfer, T, is

$$T = (\frac{N}{P})(O + \frac{P}{B}) + L + (O + \frac{P}{B})$$

$\frac{N}{P}$ is the number of individual packets, and $(O + \frac{P}{B})$ is the latency of transferring a single packet into the Outgoing FIFO at the sender or into memory at the receiver. Rearranging yields:

$$T = \frac{NO}{P} + \frac{P}{B} + (L + O + \frac{N}{B}) \quad (1)$$

The optimal value of P is found as the zero of the derivative of T with respect to P:

$$\frac{dT}{dP} = \frac{-NO}{P^2} + \frac{1}{B} = 0$$

$$P = \sqrt{OB}\sqrt{N}$$

Measurements in the laboratory (Section 5.5.1) indicate that B is about 30 Mbytes/sec and O is about 1.2 μs. Therefore, the optimal expected packet size value with respect to the total transfer size for the SHRIMP prototype is roughly $6\sqrt{N}$.

The program described is Section 5.5 is not sophisticated enough to reveal this phenomenon experimentally because there is no feedback between the nodes. Feedback is essential for determining application bandwidth because it reveals the latency component of a transfer. Therefore, a "ping-pong" program that simply sends a fixed-size chunk of data back and forth between two nodes many times was used. The bandwidth was calculated by multiplying the chunk size by twice the number of iterations, and dividing that product by the running time of the program.

The top graph in Figure 30 shows the measured bandwidth at various transfer sizes for each of the possible deliberate update packet size parameter values. The

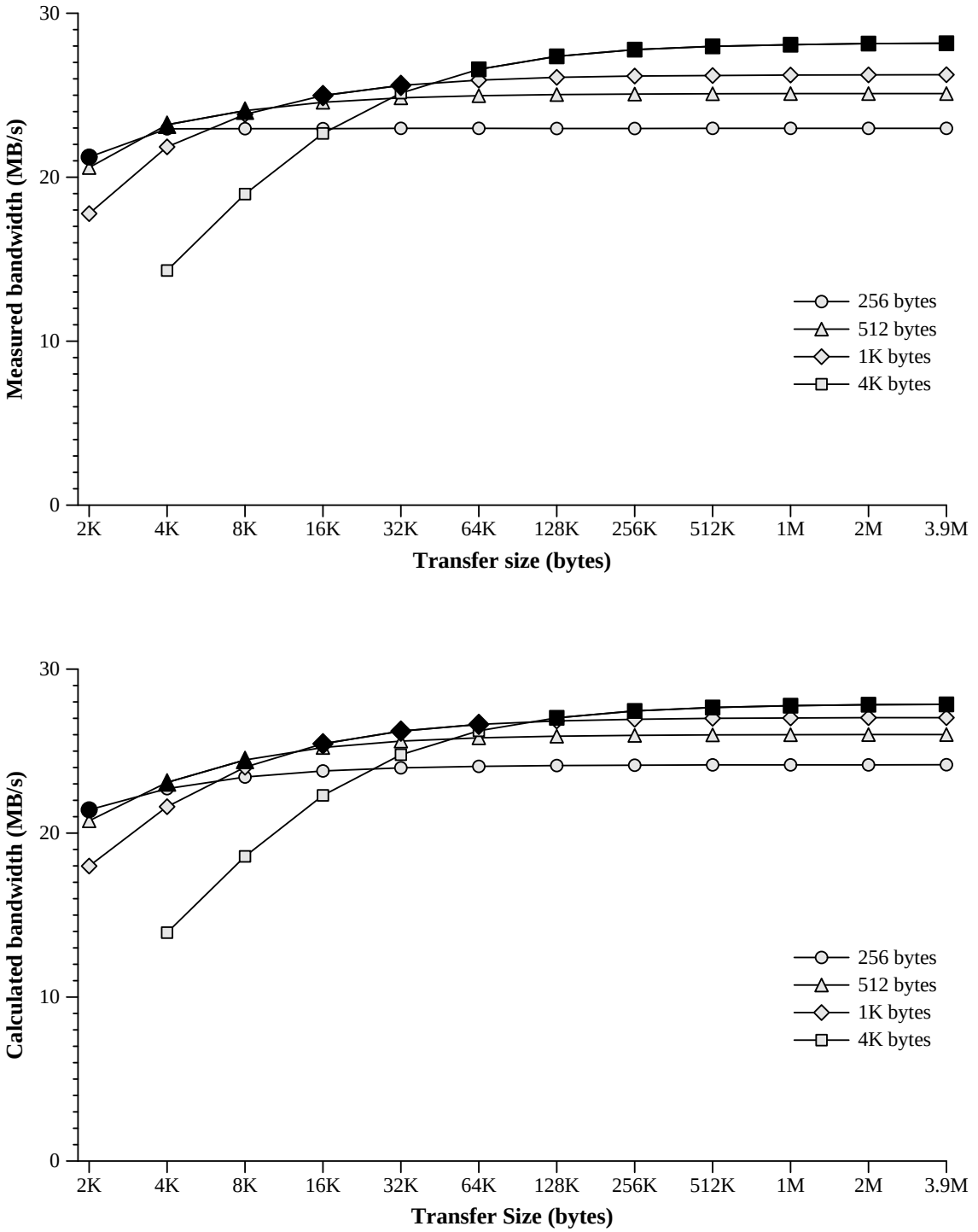


Figure 30: Bandwidth vs. transfer size for various settings of the deliberate update packet size parameter.

curve that has the highest bandwidth at each of the transfer sizes (highlighted in black) represents the best parameter choice for that transfer size. Not surprisingly, the best packet size increases as the size of the transfer increases, as predicted by the mathematical model.

In order to assess the accuracy of the mathematical model, bandwidth curves were calculated for each of the four packet size parameter values with the equation

$$Bandwidth = \frac{N}{T} \tag{2}$$

where T is given by Equation 1. The results are plotted in the bottom graph of Figure 30 using the constants L=288 ns, O=1.505 μs, and B=28.17 Mbytes/sec. Each curve is calculated with a constant value for the packet size, P, while the transfer size, N, is varied.

Visual inspection shows a strong correlation between the measured results and the calculated results. To assess whether the calculation is a good predictor of the optimal packet size, the measured and calculated points that achieved the highest bandwidth for each of the transfer sizes in Figure 30 have been highlighted in solid black. The prediction is perfect, except at the 64 Kbyte transfer size. For this size, the calculated bandwidth with P=1K is only 0.39 Mbytes/sec higher than the bandwidth with P=4K.

This experiment suggests a dynamic scheme for adjusting the packet size parameter to its optimal value based upon the transfer size. For instance, the packet size value could be set from user level as part of the transfer initiation sequence, or the network interface could automatically increase the value steadily as a transfer progressed in an attempt to adjust. Note that the curves formed by the solid black highlighted points in Figure 30 represents the bandwidth of a dynamic packet size adjustment scheme over the various transfer sizes.

5.6 Automatic Update Bandwidth

Automatic update can be used to perform large transfers by copying on the sender from un-mapped source data to an imported receive buffer. This can be a good

strategy because the bandwidth of `bcopy`[4] on the PCs used in the SHRIMP prototype is 40 Mbytes/sec from cached source data to a mapped receive buffer. The drawbacks are that the CPU is dedicated to performing the copy, and the bandwidth is much worse when the source data is not cached.

Unfortunately, the automatic update software flow control mechanism described in Section 4.6.2, which prevents the Outgoing FIFO from overflowing, was not implemented at the time these experiments were performed. Therefore, the uni-directional transfer strategy of Section 5.5 could not be used to measure the bandwidth of large automatic update transfers since the sender's higher bandwidth (40 Mbytes/sec) would cause the datapath to quickly back up and overflow the Outgoing FIFO.

Instead, a ping-pong program similar to the one described in Section 5.5.2, which sends a fixed-size chunk of data back and forth using the `bcopy` function, was used. Although the bandwidth is still receiver-limited, the smaller transfers of the ping-pong program avoid overflowing the Outgoing FIFO.

The maximum automatic update bandwidth is achieved when combining is used to build large packets from individual, consecutive automatic updates, as discussed in Section 4.3.2. Using standard combining and unlimited EISA bus possession, the maximum automatic update bandwidth achieved by the ping-pong program was 28.49 Mbytes/sec.

5.6.1 Effect of Automatic Update Packet Size

The top graph in Figure 31 shows the results of varying the automatic update packet size parameter in an experiment similar to that described in Section 5.5.2. Some of the curves are incomplete because the large sizes of the transfers at the missing data points overflowed the Outgoing FIFO.

Not surprisingly, the measured results in Figure 31 are very similar to those measured for deliberate update (Figure 30), although the crossover points are slightly different.

[4] A C-language byte copying function.

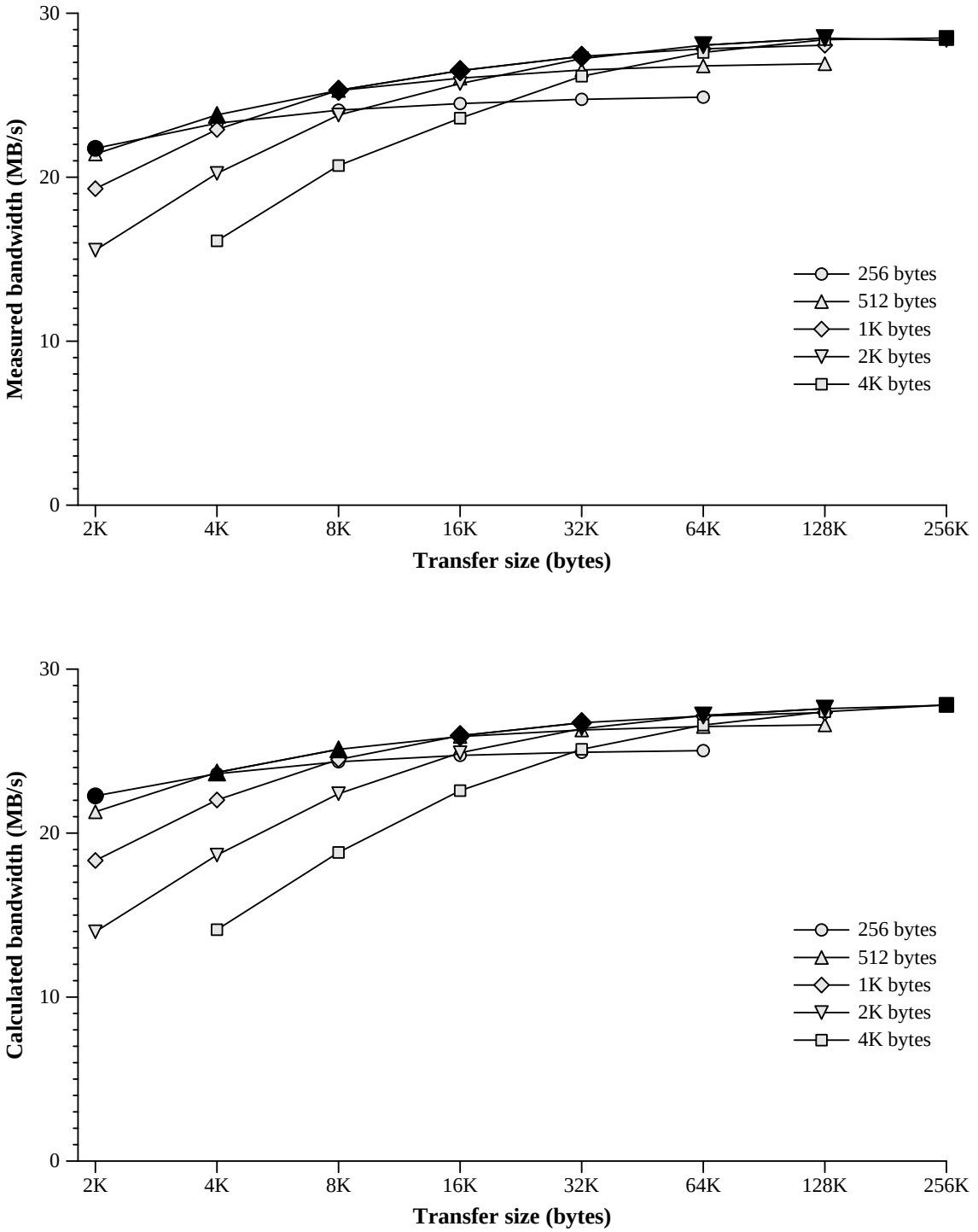


Figure 31: Bandwidth vs. transfer size for various settings of the automatic update packet size parameter.

Again, to assess the accuracy of the mathematical model for optimal packet size, Equation 2 was used to calculate bandwidth curves for each of the five packet size parameter values. The results are plotted in the bottom graph of Figure 31 using the constants L=288 ns, O=1.2 μs, and B=28.49 Mbytes/sec. Note that O is the overhead of the receiver since it dominates the sender's for automatic update.

As was the case for deliberate update, visual inspection shows a strong correlation between the measured results and the calculated results. Again, the measured and calculated points that achieved the highest bandwidth for each of the transfer sizes in Figure 31 have been highlighted in solid black. The prediction is correct, except at the 8 Kbyte transfer size, where the calculated bandwidth with P=512 is only 0.62 Mbytes/sec higher than the bandwidth with P=1K.

5.6.2 Latency-Bounded Combining

The same ping-pong program was used to experiment with latency-bounded combining and determine the effects of varying the automatic update combining delay parameter described in Section 5.1. Figure 32 shows how bandwidth varies with combining delay for a transfer size of 4096 bytes (1 page). The automatic update packet size was set to 4 Kbytes so as not to cause further subdivision of packets and interfere with the experiment.

The combining delay on the horizontal axis of Figure 32 is the time that the hardware will wait for a consecutive write before launching a combining packet. The minimum value is 75 ns because of the design of the timer and its interaction with a control state machine.

The results in Figure 32 are caused by the memory bus behavior of the program and the system. The `bcopy` function, which is used to transfer the data, repeatedly reads a word of data from source memory and then writes it to destination memory. Since the destination memory is mapped for automatic update, it is cached write-through and the 40 Mbytes/sec write bandwidth means there is a 4-byte write cycle every 100 ns on the bus.

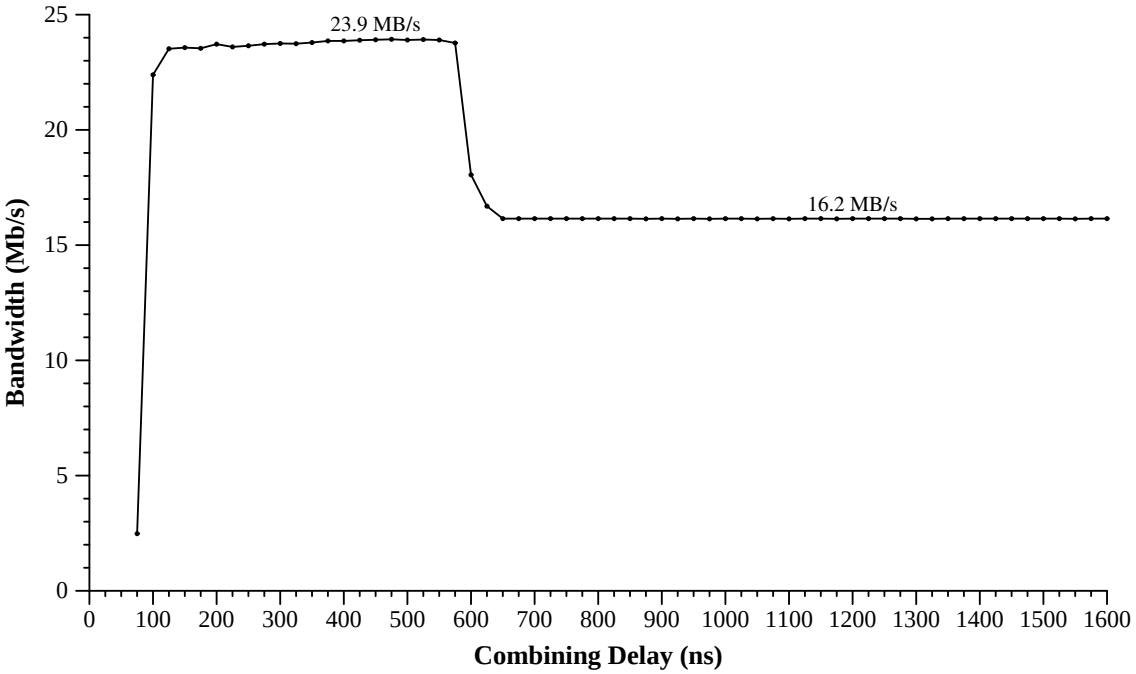


Figure 32: Bandwidth of a one-page (4096-byte) automatic update transfer as the combining delay is varied.

Because the first-level data cache is 8 Kbytes in size and 2-way set associative, both the 4-Kbyte source region and 4-Kbyte destination memory can be cached. Therefore, in its steady state, the `bcopy` function reads from the first level cache and writes through to memory. Since the reads are so fast and the CPU has a write buffer, the memory bus behavior is just a steady stream of word writes every 100 ns. The only interruption in the stream of writes is memory refresh.

With automatic update writes occurring every 100 ns, the minimum combining delay of 75 ns results in poor bandwidth because individual packets are generated for every write. When the combining delay is set to 100 ns or more, the bandwidth increases dramatically since write bursts are combined into large packets.

The drop in bandwidth from 23.9 Mbytes/sec to 16.2 Mbytes/sec at a combining delay of 600 ns is a result of main memory refresh, which occurs every 15.1 μs and utilizes the memory bus for about 600 ns. When the combining delay is less than 600 ns, then packets are accumulated only between refresh operations because the combining timer expires during refresh and launches the packet. When the combining

delay exceeds the refresh time, entire pages (4 Kbytes) are transferred as single packets.

Since there are about 580 bytes of automatic updates between refresh operations, the bandwidth at a combining delay below 600 ns is about the same as that of the 512 byte curve in Figure 31 for a transfer size of 4 Kbytes. At a combining delay of 600 ns or greater, 4 Kbyte packets are formed so the bandwidth drops to that of the 4 Kbyte curve in Figure 31.

Combining Larger Transfers

Similar behavior results when the transfer size is set to 16 Kbytes (4 pages); however, misses in the first level cache caused by the larger volume of data create an interesting effect as shown in Figure 33. In this case, the 8-Kbyte first level cache is too small to contain the 16 Kbytes of source memory and 16 Kbytes of destination memory. The 256 Kbyte second-level cache, however, can contain the entire data set.

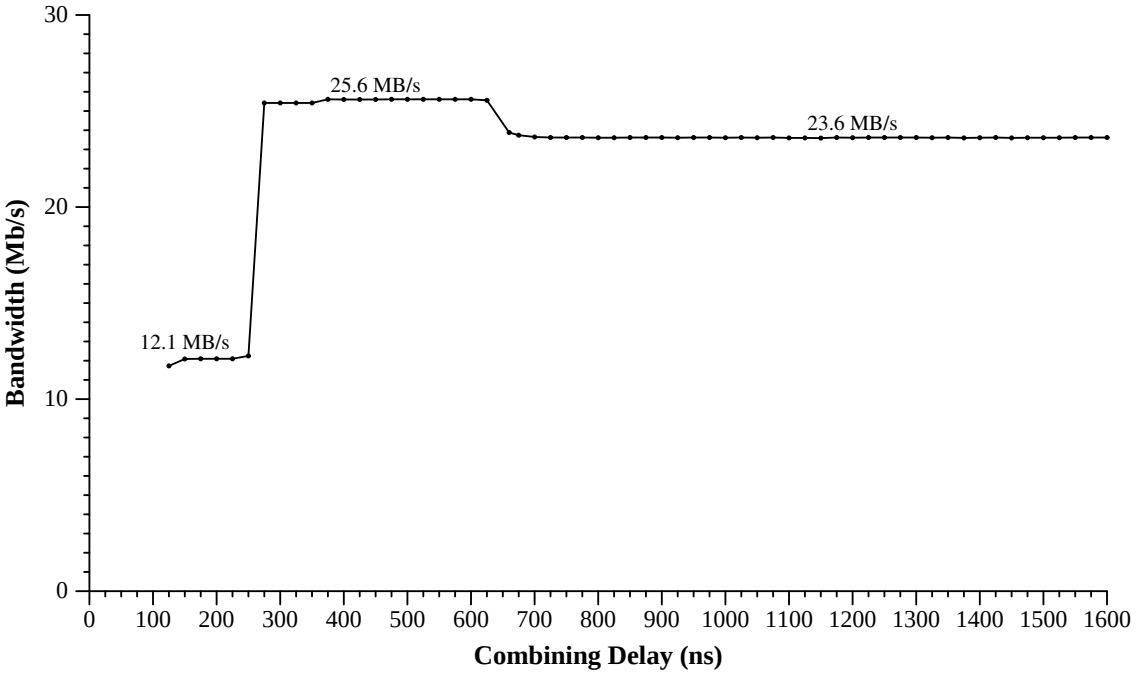


Figure 33: Bandwidth of a four-page (16384-byte) automatic update transfer as the combining delay is varied.

In steady state, the first-level cache is constantly being overwritten, causing the first `bcopy` read of a cache line to miss. A cache line fill from the second-level cache takes about 150 ns, causing a 250 ns gap in the write stream. So, the `bcopy` memory bus behavior consists of a stream of writes every 100 ns, with a 250 ns separation every eight writes (32 bytes).

Therefore, small combining delays in Figure 33 result in low bandwidth because packets are terminated every time a second-level cache read occurs. Once the combining delay is long enough to span the cache read (275 ns), large packets are created and the bandwidth immediately increases. Again, the drop in bandwidth around 600 ns is caused by memory refresh, and matches the behavior in Figure 31 for a transfer size of 16 Kbytes.

5.7 The Effect of Timestamping

When timestamping is enabled, every network packet carries an eight byte timestamp, increasing its size. Regardless of whether timestamping is enabled or not, the timestamp is included in every packet queued in the Outgoing FIFO.

For the bandwidth experiments presented in this chapter, enabling timestamping did not have any measurable effect. It did, however, affect the latency of small packets somewhat, as expected. Interestingly, timestamping increased the roundtrip latency of the single-integer ping-pong program described in Section 5.4 from 7.42 μs to 7.5 μs with the caches disabled, but had no effect with the caches enabled.

Figure 34 plots roundtrip latencies for several small transfers, with timestamping both enabled and disabled. The program used to obtain these results sent the data using a simple series of integer assignments to consecutive automatic update addresses, with standard combining enabled. A non-consecutive write was used to launch the combined packet. Note that the ordinate scale begins at 10 μs.

Clearly, the effect of timestamping on packet latency is very small. This graph highlights another, unrelated behavior of the network interface. The curve appears to step up sharply at 8 bytes, 16 bytes, and 24 bytes. This is because the data is read from the NIC eight bytes at a time. The higher steps are caused when the packet

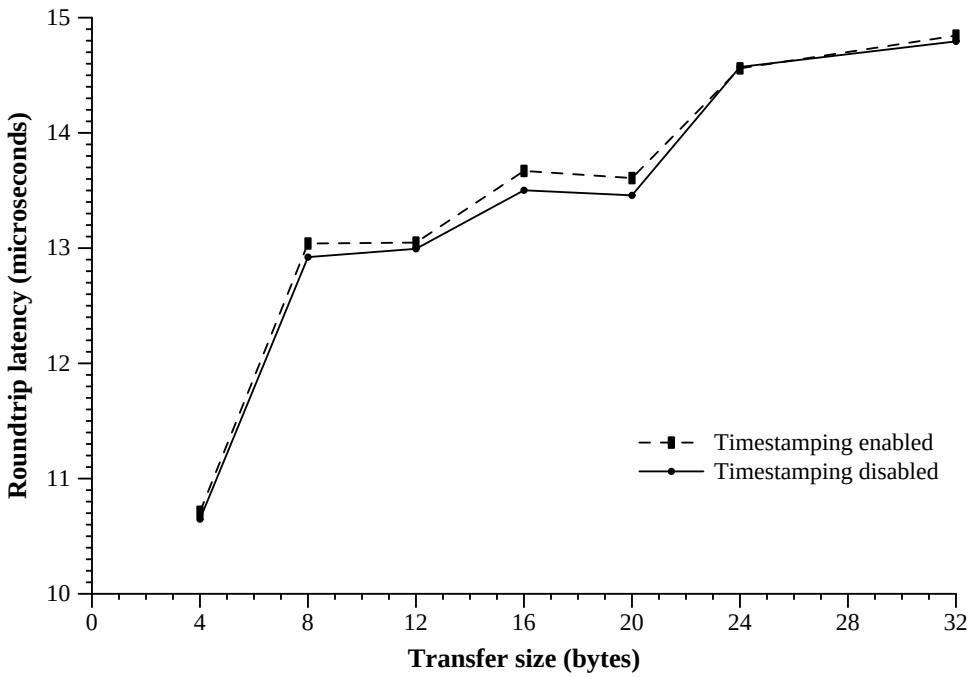


Figure 34: The effect of timestamping on the latency of various small transfers.

length is increased by eight bytes to accommodate additional data, leading to an extra NIC read. The packet length is not increased at 12 bytes, 20 bytes, or 28 bytes.

5.8 Cache Effects

The experiments presented in this chapter so far have largely ignored the performance effects of the caches, which is not insignificant. To study these effects, various experiments were performed with the simple deliberate update program of Section 5.5 and with the `bcopy` function.

5.8.1 PC Cache Architecture

The PCs used to build SHRIMP contain a two-level data cache hierarchy consisting of an 8-Kbyte, 2-way associative first-level cache, and a 256-Kbyte, 2-way associative second-level cache (which is also used for instructions). The cache update policy can

be configured on a per-page basis, and the choices are write-back, write-through, or no caching at all.

The memory system maintains consistency between the caches and main memory, even in the presence of DMA. When a DMA read is attempted from memory that is cached and modified, the memory system stalls the read, writes the line containing the modified data back to memory, and then lets the read proceed. If the modified line is in the first-level cache, a cascading write-back is performed. The same consistency action is taken for DMA writes.

5.8.2 Automatic Update

In order to send automatic update data, it must be written to memory so that the network interface can snoop it. Although this can be accomplished with write-back caching by flushing the data from the cache, it is expected that in most cases automatic update-bound pages will simply be cached write-through or not at all, since the Pentium has no individual line flushing capability.

Therefore, large automatic update transfers are expected to be performed by copying source data to bound pages using the optimized `bcopy` function. Angelos Bilas performed measurements of the `bcopy` bandwidth obtained when copying from variously cached sources to a write-through destination on a Pentium PC, and the results are presented in Table 4. Naturally, the bandwidth drops as the source data is located farther from the CPU.

Source	Bcopy bandwidth (MB/s)
First-level cache	38.0
Second-level cache	32.3
Memory	19.2

Table 4: Bandwidth of the `bcopy` function with various source caching. The source is cached write-back and the destination is cached write-through.

For all of these measurements, the source memory is cached write-back, so the first cache miss on a line brings the entire line (32 bytes) into the first-level cache. An un-cached memory-to-memory `bcopy` achieves a bandwidth of only 8 Mbytes/sec.

The `bcopy` bandwidth is highest when copying from the first-level cache, so automatic update should perform best for small transfers of data that is actively used or copied shortly before the transfer, since that data is likely to be in the first-level cache. For large transfers, the source data is unlikely to be in the relatively small first-level cache. However, the 256-Kbyte second-level cache could retain a good portion of a large transfer, especially if the source data had been utilized before the transfer.

Keep in mind that a high `bcopy` bandwidth affects the sending rate, but the achievable transfer bandwidth is still limited by the incoming bandwidth of the receiving network interface.

5.8.3 Deliberate Update and Incoming DMA

Both incoming DMA and deliberate update use DMA to transfer data between memory and the network interface. The caches significantly affect the performance of these transfers when the target memory is in the cache and modified. Note that "target memory" refers to the source memory for deliberate update send and the destination memory for incoming DMA receive.

In order to measure the effect of the caches on DMA, the simple program of Section 5.5 was used to transfer 4096 bytes of data between two nodes with the deliberate update packet size parameter set to 4096 bytes, the EISA bus possession parameter set to unlimited, and the source and destination memory cached write-back. The results are summarized in Table 5. The first column in the table indicates the location of the most current source or destination data.

The bandwidths presented in the table were obtained with hardware measurements and represent the actual bandwidths of the DMA transfers themselves. These numbers do not include any software overhead or transfer startup overhead such as bus arbitration; hence, the inconsistency with Table 3. A measurement for the

Target memory caching	Outgoing bandwidth (MB/s)	Incoming bandwidth (MB/s)
First-level cache	N/A	15.94
Second-level cache	16.65	16.65
Un-cached	29.47	29.47

Table 5: Bandwidth of 4096-byte DMA transfers between memory and the network interface.

first-level outgoing case is not presented because consistent behavior could not be achieved, despite considerable effort. However, the bandwidth measured was always either 15.94 Mbytes/sec or 16.65 Mbytes/sec.

The target memory was brought into the first-level write-back cache by reading it and then modifying it. DMA to or from this modified memory was the slowest because a cascading writeback was required for every line of the target memory. Since an entire line (8 words) is written back at once, every eighth DMA transfer was stalled by the writeback.

Similarly, the target memory was brought into the second-level write-back cache by reading it, modifying it, and then reading 8 Kbytes of some other consecutive physical memory, causing the target memory to be evicted from the first-level cache. As expected, DMA in this case is faster then the previous case, but only marginally so.

To obtain the measurements in the last row of Table 5, both caches were flushed of the target memory by reading 256 Kbytes of some other consecutive physical memory. The results in this case are consistent with the application-level measurements of Section 5.5.

5.8.4 Discussion

The measurements presented in this section represent extremes, with *all* the data either cached or un-cached. In practice, performance would more likely be somewhere between the extremes. For example, average hardware DMA bandwidth is likely to

be between 16.65 Mbytes/sec and 29.47 Mbytes/sec, and automatic update `bcopy` is likely to achieve between 19.2 Mbytes/sec and 38 Mbytes/sec on average.

The best deliberate update bandwidth is obtained when the most current source data is in memory at the time of a transfer initiation, as was the case for the measurements reported in Table 3. Unfortunately, source data is likely to be cached write-back in the interest of computational efficiency, so this condition is unlikely to occur.

Conversely, the best automatic update `bcopy` bandwidth is obtained when the most current source data is in the cache at the time of the copy. In this case, write-back caching of the source data is an advantage.

Source memory caching	Automatic update bandwidth (MB/s)	Deliberate update bandwidth (MB/s)
Cached	32.3	16.7
Un-cached	19.2	29.5

Table 6: Bandwidth comparison between automatic update and deliberate update with source data caching considered.

Table 6 summarizes the tradeoff between automatic update and deliberate update bandwidths in the presence of source memory caching. The table suggests a dynamic strategy for obtaining optimal bandwidth for every transfer. If there was a way to determine the amount of source data in the cache or in memory, then a dynamic selection could be made between automatic update and deliberate update at the time of a transfer (assuming appropriate mappings and bindings were in place).

The effect of caching on the latency of a small transfer was discussed in Section 5.4. For larger transfers, latency is proportional to bandwidth, so the results in this section can be used to estimate the relative effects of caching on the latency of larger transfers.

It should be pointed out that these cache-effect results are dependent upon the specific memory system of the PC used for the prototype SHRIMP system. More sophisticated memory systems allow caches to "snarf" DMA data being written to memory in order to avoid stalling writes. Conversely, sophisticated caches can avoid

stalling DMA reads by supplying modified data during the read cycles. There is no reason to assume that such performance-enhancing features will not make their way into future PC memory systems.

5.9 Conclusions

The single-word, automatic update hardware latency of 2.5 μs is very close to the predicted goal of 2 μs, stated in [BLA+94]. Some design decisions that compromised performance for ease of implementation were made, so a 2 μs latency is not out of reach for a board-level network interface design.

Figure 24 shows that packet latency is limited mostly by the incoming datapath, which is dependent upon the performance of the EISA bus. In addition, the bandwidth of large transfers is limited by the EISA bus for deliberate update and by the memory bus `bcopy` bandwidth for automatic update. It is interesting to consider what the effects of designing to current bus technology would be.

As far as I/O expansion buses are concerned, the currently emerging standard for PCs is the PCI bus [PCI95], which is typically implemented with a 33 MHz bus clock, yielding a maximum burst bandwidth of 132 Mbytes/sec. For reasons similar to those discussed in Section 5.5 such as refresh and bus sharing, the actual realizable bandwidth of large transfers is probably on the order of 100 Mbytes/sec. This is still less than the 113.21 Mbytes/sec measured bandwidth of the SHRIMP network in streaming mode (Section 5.3.2), so the full practical bandwidth of the PCI bus would be available for transfers, in the absence of network contention.

In order for the network interface to support a data rate of 132 Mbytes/sec, some re-design would be necessary. The outgoing datapath of the current design accepts a word of data every 50 ns in order to keep up with the maximum bandwidth of the Xpress memory bus. By simply widening the Outgoing FIFO to 128 bits, a word of data could be accepted every 25 ns, which is less than the 30 ns imposed by the PCI bandwidth. The incoming datapath, however, would need a more significant revision because the state machines are implemented in FPGAs, which have limited speed.

However, the re-design could be as simple as using fast PLDs instead of FPGAs for the state machines.

A rough estimate of the latency attainable with a PCI-bus design can be obtained by assuming that a re-designed board operates the same as the present EISA-bus design, but with the quadrupled PCI-bus clock rate. Referring to Figure 24, the incoming datapath latency would be reduced from 1.566 μs to about 600 ns, while the outgoing datapath and network latency would remain unaffected. Therefore, the latency of a single-word automatic-update transfer would be reduced from about 2.5 μs to about 1.6 μs. Similarly, the latency of a deliberate update transfer initiation, shown in Figure 25, would be reduced from 1.92 μs to about 480 ns, so the latency of a single-word deliberate update transfer would be reduced from about 4.5 μs to about 2.1 μs.

In truth, achieving these reduced latencies with the PCI bus would require a good deal of redesign in order to accommodate the higher clock rate. Alternatively, the hardware could be designed to operate at the 33 MHz PCI frequency rather than the doubled 66 MHz frequency assumed in these simple estimates. In that case, the latency reductions would not be as dramatic.

The currently emerging standard for PC memory buses is the 66 MHz Pentium Pro memory bus [Gwe95], which is 64 bits wide and has a maximum write bandwidth of 528 Mbytes/sec. In order to accommodate such a data rate, the outgoing datapath would have to be able to create a new packet containing 8 bytes of data every 15 ns. This could be accomplished with a 144-bit-wide Outgoing FIFO and a pipelined OPT lookup design, assuming a slightly smaller timestamp was acceptable. Furthermore, a 66 MHz oscillator would be required to control the pipeline. This would be a difficult design to implement with discrete logic, but not particularly challenging for a custom integrated circuit, or perhaps even a gate array.

Programmable Parameters

Designing significant hardware parameters to be programmable proved to be invaluable for investigating the interaction between the network interface and the PC hardware. Varying the parameters revealed surprising network interface behaviors.

For example, the effect of the combining delay on automatic update bandwidth was unexpected, but explainable through careful analyses of the bus and memory system details. In general, attention to these hardware details is very important for achieving the highest performance from a network interface with low software overhead because applications basically experience the hardware performance directly.

The experiments presented in this chapter helped to choose good values for the programmable parameters, and showed where it might be useful to dynamically adjust a parameter, or perhaps even design it to be self-adjusting.

Table 3 in Section 5.5.1 shows that the unlimited setting for the EISA bus possession parameter yields the highest bandwidth, but it violates the EISA bus specification. It was observed, however, that the PC's memory system performs blocked refreshes immediately after the bus is released, presumably maintaining the memory contents. Without further experimentation or information about the memory system, setting the bus possession parameter to unlimited should be considered risky. The safe choice with the highest bandwidth is 60 clocks.

Optimal settings for the deliberate update and automatic update packet size parameters have been shown to be dependent upon the transfer size. As Figures 30 and 31 show, the choices for these parameters have the most significant effect on smaller transfers. The best setting for transfers under 16 Kbytes in size seems to be 512 bytes, and for transfers under 2 Kbytes in size the best setting seems to be 256 bytes.

In [CHKM93], eight scientific applications were studied, and their communication characteristics measured. Of the average message sizes transferred by all eight applications, it was found that 78% were under 1K in size and 54% were under 128 bytes in size. This would suggest that the packet size parameters should be set to 256 bytes; however, 512 bytes could be a better choice because the loss in bandwidth for small

transfers is not significant, and the 512-byte setting performs well up to a message size of 16 Kbytes.

Figures 32 and 33 clearly indicate that the best setting for the automatic update combining delay parameter is around 550 ns.

Finally, it was found that timestamping causes a slight increase in packet latency, especially for small transfers, so it is best disabled unless performance is being monitored.

Chapter 6

Conclusions

6.1 Communication Performance Gap Revisited

The communication performance gap between custom and commodity interconnects, introduced in Chapter 1, is caused by both software overhead and raw communication hardware performance. In many previous systems, the balance of the problem has rested with the software.

The SHRIMP network interface shifts this balance back to the hardware by eliminating almost all of the software overhead from the critical path of communication. Fortunately, a migration of interconnection technology from multicomputers to LANs is simultaneously shrinking the hardware communication performance gap, so networks of PCs based on SHRIMP technology will be fast, inexpensive, and scalable.

Using a network interface built largely with commodity components and minimal software support, the prototype SHRIMP sytem demonstrates that low-overhead communication is feasible, practical, and cost-efficient. The key to this achievement is the SHRIMP network interface, which provides protected, user-level data transfer. Protection allows general multiprogramming to be used, extending its benefits from uniprocessors to multiprocessors. User-level data transfer eliminates the costly involvement of the operating system from the critical path of communication.

The prototype SHRIMP system is a very cost effective implementation of the main ideas of this dissertation; however the ideas are applicable to virtually any DMPP

network interface. In fact, a highly custom design, such as that of a multicomputer, could improve upon the techniques described in this dissertation by allowing, for example, memory bus mastering for deliberate update and incoming DMA.

6.2 Two Low-Overhead Transfer Strategies

The SHRIMP network interface provides two transfer strategies called automatic update and deliberate update. Automatic update is an eager-sharing mechanism where communication is generated as the side-effect of an ordinary main memory write. The write itself is considered software overhead only if it is performed to explicitly send the data. Otherwise, automatic update has no software overhead whatsoever.

Measurements of the actual prototype system show that a single integer automatic update has a hardware latency of just 2.5 μs from the time of the memory write until the integer is deposited into remote memory. A program that repeatedly passes a single integer back and forth between two nodes achieves an average application-level transfer latency of just 3.71 μs, including the program overhead and periodic system events such as memory refresh.

The automatic update mechanism can be used to perform large data transfers using memory-to-memory copying, because the hardware has the capability to combine consecutive memory writes into large packets. This method has the drawbacks that the CPU must be devoted to the transfer, the bandwidth can be poor if the source data is not cached, and extra memory is used for the automatic update bindings.

Deliberate update is a high-performance mechanism for specifying a larger transfer with just two user-level memory references. Once specified, the transfer is performed by a DMA controller on the network interface board, allowing the CPU to continue with computation.

Deliberate update is the preferred mechanism for large data transfers because it can achieve the maximum practical bandwidth of the EISA expansion bus and it allows communication to overlap computation. Measurements of the prototype system show that deliberate update achieves 50% of the maximum bus bandwidth

for a transfer size of just 300 bytes, and 98% of the maximum at just 4 Kbytes (one page).

The reason for this excellent performance is the low overhead of initiating deliberate update DMA transfers, which is done with a sequence of two user-level memory references. Even on the relatively slow EISA bus, the time to perform these two references was measured as just 0.84 μs, and the time to fetch the first word of memory was just 1.08 μs.

The high performance of deliberate update is encouraging because it demonstrates that SHRIMP network interface technology is applicable to systems that do not provide the memory bus access needed for automatic update. A network interface supporting only deliberate update would be highly portable between systems and across system generations since expansion bus specifications are standardized and relatively long-lived.

Memory bus specifications, on the other hand, are proprietary and often change with every system generation. In addition, memory bus loading restrictions, high speeds, split transactions, and pipelined protocols increase the complexity of the memory bus snooping operation required for automatic update. Finally, some processors, such as the Intel Pentium Pro [Int96] do not provide write-through caching, severely limiting the side-effect initiation advantage of automatic update.

Future SHRIMP network interfaces could still support both buses because the bulk of the hardware is utilized for both deliberate update and automatic update. In fact, the prototype implementation consists of two boards, one that plugs into the expansion bus and contains almost all of the hardware, and the other that plugs into the memory bus. The prototype can be operated with deliberate update only by simply disconnecting the memory bus board.

6.3 A Flexible Interface

The VMMC API described in Chapter 2 provides an efficient means for expressing the functionality of the network interface hardware. The import-export concept is a

convenient communication abstraction that can be used alone or as a foundation for sophisticated parallel programming models, as shown in Figure 13.

Toward this end, the SHRIMP group has developed several user-level communication libraries, supporting both standard (compatible) and specialized APIs. The compatible libraries include Intel NX/2 message passing, Sun RPC (remote procedure call), and Unix sockets, while the specialized libraries are shared virtual memory and SHRIMP RPC. Performance results in the following descriptions were all taken from [FAB+96].

The NX/2 message passing library, developed by Richard Alpert [ADFL96], uses two protocols, depending upon the message size. Small messages use a simple protocol with pre-allocated buffers on the receiver, and large messages use a more complex zero-copy protocol, which transfers data directly between the application's source and destination data structures in memory.

The small message protocol has a latency cost of just over 6 μs above the hardware limit, including the time to manage communication buffers. The zero-copy protocol achieves the bandwidth of the hardware for large messages where the time required for exchanging the receive buffer address is fully amortized.

The Sun RPC library, developed by Angelos Bilas [BF96], achieves high performance by reimplementing the lower layers of the RPC communication protocol stack using VMMC. The resulting implementation performs a null RPC with no arguments in just 29 μs, and with a single integer argument in just 41 μs. The bandwidth for large transfers is comparable with that of the raw hardware when the required copying is taken into account.

Probably the most generally useful compatibility library is Unix stream sockets, developed by Stefanos Damianakis [DDF96]. Because of protection requirements in the sockets model, it is not possible to use a zero-copy communication protocol. Still, small messages incur just a 13 μs latency cost above that of the raw hardware, and large messages achieve the bandwidth of the raw hardware, taking the cost of copying into account.

The specialized SHRIMP RPC library, developed by Edward Felten [BF96], demonstrates the performance achievable with a protocol library when compatibility is not

an issue. The SHRIMP RPC library is complete, however, with a stub generator that reads an interface definition file and generates code to marshal and un-marshal complex data types.

SHRIMP RPC performs a null RPC in just 9.5 μs, which is comparable with the minimum hardware latency and is more than a factor of three faster than the compatible library's null RPC. For large transfers, the difference between the two libraries is roughly a factor of two. Most of this performance advantage is achieved by transferring both procedure arguments and return values as they are marshaled using automatic update.

In addition to explicit communication protocols, the SHRIMP network interface can be used to support the implicit communication of shared virtual memory, as demonstrated by Liviu Iftode [IDFL96]. The SHRIMP shared virtual memory implementation uses a new variation of lazy release consistency [KCZ92] called automatic update release consistency (AURC). AURC takes advantage of the eager-sharing capability of automatic update to propagate updates between shared memory pages. In addition, deliberate update is used to distribute copies of entire pages.

Although an actual implementation of AURC has not been tested, simulations of up to sixteen processors [IDFL96] show good speedups on several applications from the Splash-2 [WOT+95] benchmark suite. The simulations compared AURC with lazy release consistency and found an average speedup improvement from 5.9 to over 8.3.

The key to the excellent performance of all the SHRIMP communication libraries is the protected, user-level communication provided by the SHRIMP network interface. Rather than impose a complex communication protocol upon applications, user-level communication allows customized libraries to use as little buffering and copying as they need.

6.4 The Hard Choices

Chapter 3 presented several ideas for increasing the performance of VMMC with network interface hardware. These various ideas essentially document much of the

thinking that went into the design of the SHRIMP network interface. The ones that were selected for implementation satisfied the constraints of the project but are not necessarily the best for every situation.

For example, deliberate update request queueing was not implemented because of the additional complexity involved. With a modest amount of additional manpower or time it probably could have been included. Experience with the prototype system has yet to determine whether this, and other such tradeoffs, turn out to be drawbacks or not.

6.5 A Few More Words About UDMA

User-Level DMA [BDFL96], presented in Chapter 4, was described as a mechanism for transferring deliberate update data from memory to the network interface. UDMA, however, is a general-purpose mechanism that can be applied to any situation where low-overhead DMA transfer initiation is desired. Recall that UDMA is simply a method for loading the configuration registers of a standard DMA controller from user level in a restricted manner. The price to pay for the advantages of UDMA is address space, the overhead of maintaining proxy mappings, and some additional hardware.

In general, UDMA is used to initiate transfers between regions of bus addresses. The interpretation of bus addresses can be either real DRAM addresses or device I/O ports, depending on the implementation of the DMA controller.

The UDMA scheme presented in Chapter 4 used two proxy spaces, one for the source memory and one for NIPT destination pointers, in order to provide protection for both. It is possible, in some situations, to make do with only one proxy space because the store and load references provide sufficient differentiation between source and destination. In this case, however, protection is restricted because a single virtual-to-physical proxy mapping allows a user-level process to specify both the source and destination associated with the mapped page. It is, of course, possible to restrict the mapping to read-only access, but write-only would be needed as well in order to

generalize this mechanism. With two proxy spaces, separate mappings are required for each and protection can be maintained separately for the source and destination.

6.6 A Project with a Future

At the time of this writing, the SHRIMP project is still in its adolescence. There are several known bugs in the network hardware that need to be dealt with before a stable 4-node system can be assembled. The project's transition to maturity will come with the assembly of a 16-node system, for which an effort has already begun.

Future plans for the next-generation network interface are PCI bus compatibility and the use of an emerging high-performance LAN such as Myrinet [BCF+95]. Experience gained with the current system will determine the capabilities relegated to the next-generation hardware.

On a more philosophical note, the future of SHRIMP lies not so much in this one project, but in the ideas that it has generated. Foremost among these is the idea of stripping away the overhead of communication from the time-critical data transfer operation itself without sacrificing all the services that systems have come to depend upon. This dissertation demonstrates the viability of this idea, and it is my hope that future systems will benefit from the work that I have done.

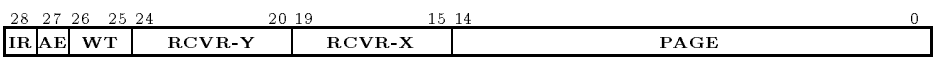
Appendix A

Network Interface Details

This appendix describes the Outgoing Page Table (OPT) format, the Incoming Page Table (IPT) format, the network packet format, and the deliberate update transfer initiation sequence in detail.

A.1 Outgoing Page Table Entry

Each entry in the OPT specifies a mapping to a remote physical page. In the case of automatic update, each entry is associated with a unique page of physical memory, and the entry is referenced when a write is performed to the associated page. In the case of deliberate update, each entry is associated with a unique page of destination physical proxy space, and the entry is referenced when a valid store is performed to the associated page. An OPT entry has the following format:



RCVR-X and **RCVR-Y** specify the absolute cartesian coordinates of the receiving node in the two-dimensional mesh. **PAGE** specifies the destination physical page on the receiver.

WT (Write Type) specifies whether and how local writes are sent to the receiver. If **WT** is *disable* (00), then no writes to the local page will generate remote updates. If **WT** is *single* (01) then each individual write to the page will generate a network

packet. If **WT** is *block* (10) or *timer* (11) then writes to consecutively increasing addresses within the page will be combined into one network packet. If the write type is *timer*, then the hardware only waits a finite amount of time for a consecutive write before terminating the packet and sending it (latency-bounded combining).

When **AE** (Automatic update Enable) is set, automatic update writes will generate remote updates. In practice, this bit is used to *prevent* writes to deliberate update pages from causing remote updates since automatic update permission is actually controlled by virtual-to-physical page mappings.

When **IR** (Interrupt Request) is set, packets generated by automatic update will carry a sender-specified interrupt request. In VMMC, **IR** is used to generate notification request packets.

A.2 Incoming Page Table Entry

The IPT supports the notification mechanism, restricts network access to a host's physical memory, and supports page grouping for performance monitoring. An IPT entry has the following format:

When the **RE** (Receive Enable) bit is set, incoming packet data is transferred directly to the local sender-specified physical memory page. When it is clear, the incoming datapath is frozen *before* the data is transferred, and an interrupt is generated to the CPU. The data transfer will not continue until the corresponding interrupt flag has been cleared.

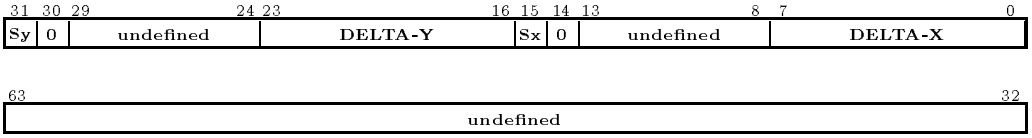
When the **IE** (Interrupt Enable) bit is set, sender-specified, user-level interrupts are enabled for the corresponding page. Any packet received with an interrupt request will freeze the incoming datapath and generate an interrupt *after* the data has been transferred to memory. Again, clearing the corresponding interrupt flag will re-enable the incoming path. In VMMC, **IE** is used to enable notifications.

The **COLOR** field is only used for an external performance monitoring board, and allows software to mark pages for statistical gathering.

A.3 Network Packet Format

The network packet format is determined by the requirements of the routing backplane and the network interface hardware. Because the processor port on the NIC is 64 bits wide, each packet is some multiple of 8 bytes in length.

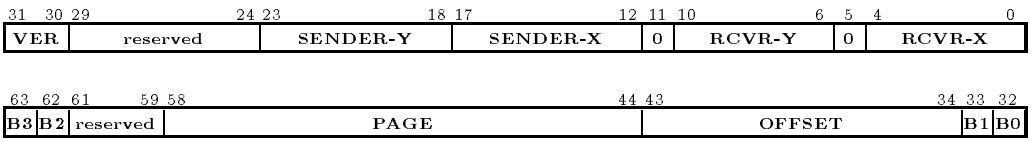
The first qword of a packet is used by the Paragon routing backplane to deliver the packet and has the following format:



The low-order word is consumed by the routers as the packet is routed, and the high-order word is stripped by the NIC on the receiving node. Therefore, neither of these words is delivered to the network interface hardware on the receiving node.

A cartesian coordinate system is used to assign a unique (X,Y) position to each of the routers in the backplane and hence, to each of the SHRIMP nodes. **DELTA-X** specifies the positive displacement in the east/west direction between the sending and receiving nodes, and the corresponding sign bit, **Sx**, specifies whether to route east or west. Similarly, **DELTA-Y** and **Sy** specify the displacement in the north/south direction.

The rest of the packet is created by the sending network interface and interpreted by the receiving network interface. The first two words form the *SHRIMP header* and have the following format:



The values for the **RCVR-X** and **RCVR-Y** fields are taken directly from the OPT and specify the *absolute* mesh coordinates of the receiving node. They are used by the network interface hardware to verify that the packet was delivered correctly by the backplane. This is an important feature because the actual routing information

is stripped by the backplane and is not available to the receiver. In the wormhole-routed backplane, it is conceivable that an entire packet could be mis-routed due to a mis-routed header, and there is no way for the receiving NIC to detect this. Since the **RCVR-X** and **RCVR-Y** fields lie within the checksummed payload of the packet, they provide a reliable mechanism to detect this error.

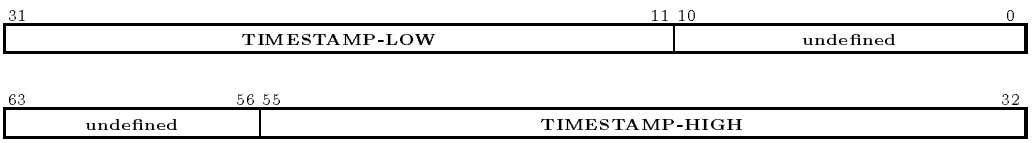
The **SENDER-X** and **SENDER-Y** fields specify the absolute mesh coordinates of the sending node, and are currently unused.

The **VER** field specifies the version of the network interface for compatibility with future systems, and is forced to "10" by the current implementation.

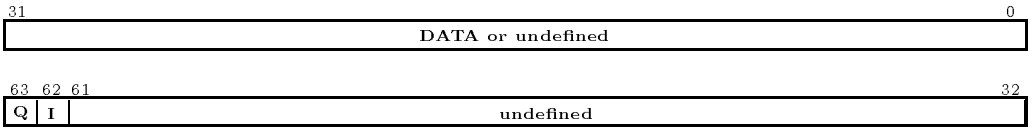
The **PAGE** and **OFFSET** fields form the base destination physical memory address for the packet data. The value for **PAGE** is taken directly from the OPT. The value for **OFFSET** comes from a write address if the packet is generated by automatic update, or a transfer initiation store operation if the packet is generated by deliberate update.

The data is transferred directly to consecutively increasing memory addresses, beginning with the base address, specified by **PAGE** and **OFFSET**, at the receiving node. The byte enable fields, **B0** - **B3**, are used only by automatic-update, single-write packets to specify which of the four bytes of the packet's data word are to be transferred to memory. This feature accommodates all valid types of byte and multiple-byte transfers.

If timestamping is enabled, then the SHRIMP header will be followed by a 44-bit timestamp with the following format:



The body of the packet consists of some number of quadwords (8 bytes each) of data. Since the NIC is 64-bit-aligned and SHRIMP packets are word-aligned, every packet has a trailer, which either contains a word of data or not. The trailer has the following format:



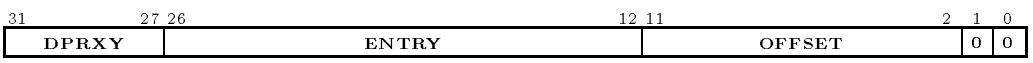
If the **Q** bit is set, then the packet data is word aligned, and the low-order word of the trailer is undefined (i.e. empty trailer). Otherwise, the low-order word contains the last word of data. Therefore, the smallest possible packet, containing one word of data, consists of two words of routing information, a two-word SHRIMP header, and a two-word SHRIMP trailer. In practice, every packet will also have a two-word CRC appended by the sending NIC, although this can be disabled.

If the **I** bit is set, the packet will cause an interrupt (i.e. a notification) at the receiver, provided that the receiver has enabled interrupts on the destination page. The reason for putting the interrupt bit in the trailer as opposed to the header is that single transfers can be broken into multiple packets dependent upon EISA bus availability, and only one interrupt should occur after *all* the data has been received.

The final two words of a network packet are the CRC appended by the sending NIC. The CRC is checked and stripped by the receiving NIC, and an interrupt is generated if it is incorrect.

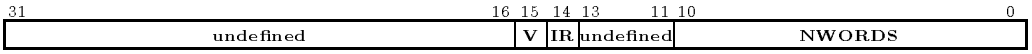
A.4 Deliberate Update

The deliberate update transfer initiation sequence consists of a store to destination proxy space followed by a load from source proxy space. The store specifies the remote destination address and the number of words to transfer. The destination physical proxy address of the store has the following format:



DPRXY specifies the base of the destination physical proxy space, which is 128 Mbytes in size, and 128 Mbyte-aligned. **ENTRY** specifies the OPT entry that contains the desired remote page number, and **OFFSET** specifies the destination offset for the packet on the remote page. The lowest two bits are zero because the address of the store must be word-aligned.

The value that is stored to destination proxy space has the following format:

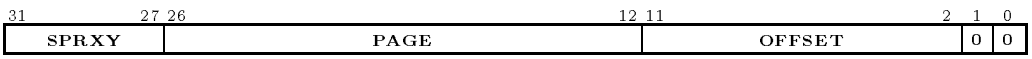


The **V** bit can be used to invalidate a partially-completed transfer initiation sequence and is typically used by the operating system on context switches to maintain atomicity of the sequence. A store to any destination proxy address with this bit set causes an invalidation.

When the **IR** (Interrupt Request) bit is set, the requested transfer will carry a sender-specified interrupt request. In VMMC, **IR** is used to generate notification request packets.

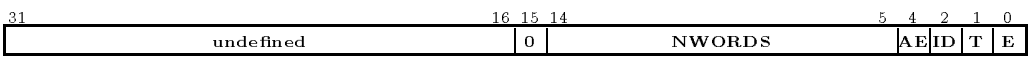
Finally, **NWORDS** specifies the number of words to transfer. A transfer of zero bytes can be specified and will not cause an error, or a transfer.

A deliberate update transfer is initiated after a valid load from source physical proxy space. The address of the load specifies the base source address of the transfer and has the following format:



SPRXY specifies the base of the source physical proxy space, which is 128 Mbytes in size, and 128 Mbyte-aligned. **ENTRY PAGE** and **OFFSET** specify the source address of the transfer in DRAM. Again, the lowest two bits are zero because the address of the load must be word-aligned.

The transfer initiation load returns the following status value:

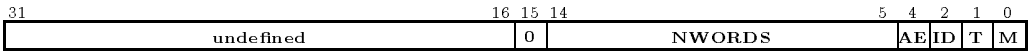


NWORDS is the number of words remaining to be transferred (which will be the same as the value stored to destination proxy space).

The **AE** (Alignment Error) flag is active if either the store or the load reference was not word-aligned. The **ID** (Invalid Destination) flag is active if the load reference was not proceeded by a valid store reference. The **T** (Transferring) flag is active if the Deliberate Update Engine is currently busy and cannot accept the transfer. Finally,

the **E** (Error) flag is active if any of the other three status flags are active and can be used as a simple check to see if the transfer was accepted. If so, all four of the error flags will be inactive (zero).

A load from source proxy space can be used at any time to check the status of the Deliberate Update Engine, and returns the following value:



This value is the same as the value returned by the transfer initiation load, described above, with the exception of the lowest-order bit. In this case it is the Match flag, **M**, which is active if the Deliberate Update Engine is busy transferring and the address of the load matches the base address of the load that initiated the transfer in progress. The **M** bit can be used by a sending process to poll for completion of its transfer, which is indicated by an *inactive* **M** flag. Note that the **NWORDS** field will reflect the number of words remaining to be transferred when the status load is issued, not the number originally requested.

Bibliography

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of International Conference on Supercomputing*, pages 1–6, 1990.

[ACJ+92] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed Memory Multiprocessor. In M. Dubois and S.S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 239–262. Kluwer Academic Publishers, May 1992.

[ACP95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[ADFL96] Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li. Design and Implementation of NX Message Passing Using SHRIMP Virtual Memory Mapped Communication. In *Proceedings of the 1996 International Conference on Parallel Processing*, August 1996. To appear.

[Arl88] Ramune Arlauskas. *iPSC/2 System: A Second Generation Hypercube*, pages 9–13. Intel Corporation, 1988.

[AS88] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, pages 9–24, August 1988.

[BCC+90] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H.T.Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 70–81, May 1990.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kuwalik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BCP92] BCPR Services Inc. *EISA Specification, Version 3.12*, 1992.

[BDF+94] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Two Virtual Memory Mapped Network Interface Designs. In *Proceedings of Hot Interconnects II Symposium*, pages 134–142, August 1994.

[BDF+95] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[BDFL96] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 154–165, February 1996.

[BF96] Angelos Bilas and Edward W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. Technical Report TR-512-96, Department of Computer Science, Princeton University, February 1996.

[BJMW95] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: a High-Performance Network Interface with Sender-Based Memory Management. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995. Also available as HP Laboratories technical report HPL-95-86, July, 1995.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 142–153, April 1994.

[BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.

[Bro89] E. Brooks. The Attack of the Killer Micros. Teraflop Computing Panel, Supercomputing '89, 1989.

[CHKM93] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 2–13, May 1993.

[DDF96] Stefanos Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. Technical Report TR-513-96, Department of Computer Science, Princeton University, February 1996.

[DEIK92] W. E. Denzel, A. P. J. Engbersen, I. Iliadis, and G Karlsson. A Highly Modular Packet Switch for Gb/s Rates. In *Proceedings of XIV*

International Switching Symposium, volume 2, pages 236–240, October 1992.

[Del88] Gary S. Delp. *The Architecture and Implementation of Memnet: A High-Speed Shared-Memory Computer Communication Network*. PhD thesis, Dept. of Electrical Engineering, University of Delaware, May 1988.

[DIFL96] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the IEEE 10th International Parallel Processing Symposium*, April 1996.

[DS86] William J. Dally and Charles L. Seitz. The Torus Routing Chip. *Distributed Computing*, 1:187–196, 1986.

[DS87] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[ECGS92] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[FAB+96] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of 23rd International Symposium on Computer Architecture*, May 1996. To appear.

[Fel93] Edward W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. PhD thesis, Dept. of Computer Science and Engineering, University of Washington, August 1993. Available as technical report 93-09-09.

[FFM96] Wenjia Fang, Edward W. Felten, and Margaret Martonosi. Contention and Queueing in an Experimental Multicomputer: Analytical and Simulation-based Results. Technical Report TR-508-96, Department of Computer Science, Princeton University, January 1996.

[FKD+95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, MI, December 1995. ACM.

[Gil95] Richard Gillett. Memory Channel Network for PCI. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995.

[Gil96] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[Gwe95] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, pages 9–15, February 1995.

[HB90] Jiun-Ming Hsu and Prithviraj Banerjee. A Message Passing Coprocessor for Distributed Memory Multicomputers. In *Proceedings of Supercomputing '90*, pages 720–729, November 1990.

[Hew94] Hewlett Packard Company. *HP 16500B/16501A Logic Analysis System User's Reference*, April 1994. Publication number 16500-97010.

[HGDG94] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1994.

[Hil92] Mark D. Hill. What is Scalability? In M. Dubois and S.S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 89–96.

Kluwer Academic Publishers, May 1992. Also published in Computer Architecture News, 18(4):18-21, December 1990.

[HJ92] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.

[HM93] Mark Homewood and Moray McLaren. Meiko CS-2 Interconnect Elan – Elite Design. In *Proceedings of Hot Interconnects '93 Symposium*, August 1993.

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

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, second edition.* Morgan Kaufmann, 1996.

[IDFL96] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 14–25, February 1996.

[Int] Intel Corporation. *Asynchronous Modular Architecture Memory Bus Interface Specification, Rev 1.4.* Document number 130-1354-44.

[Int91a] The iPSC/860. Technical Reference Manual, 1991.

[Int91b] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[Int93] Intel Corporation. *Express Platforms Technical Product Summary: System Overview*, April 1993.

[Int96] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 1: Specifications*, 1996. Order number 242690.

[JJT93] Andrew W. Wilson Jr., Richard P. LaRowe Jr., and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[JPCI94] Richard P. LaRowe Jr., Thomas H Probert, James G. Crocker, and Robert J. Ionta. Jacquard: An Architecture for Shared Memory Networks of Workstations. Technical Report JT.011, Enterprise Computing Institute, November 1994.

[KCZ92] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Ken92] Kendall Square Research. *KSR1 Technical Summary*, 1992.

[KG91] Vipin Kumar and Anshul Gupta. Analyzing the Scalability of Parallel Algorithms and Architectures. In *International Conference on Supercomputing*, June 1991.

[KLMO91] A.R. Karlin, Kai Li, M.S. Manasse, and S. Owicki. Emprical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[LAD+92] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S.

Wells, M.C. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, Sep 1988.

[MCM96] Margaret Martonosi, Douglas W. Clark, and Malena Mesarina. The SHRIMP Performance Monitor: Design and Applications. In *Proceedings of the 1996 Symposium on Parallel and Distributed Tools*, May 1996. To appear.

[MK96] Evangelos P. Markatos and Manolis G. H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 144–153, February 1996.

[NA91] Daniel Nussbaum and Anant Agarwal. Scalability of Parallel Machines. *Communications of the ACM*, 34(3):57–61, March 1991.

[NPA92] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of*

20th International Symposium on Computer Architecture, pages 224–235, May 1993.

[Osb94] Randy Osborne. The Direct Deposit Model for Low Overhead Communication in High Speed LANs. Technical Report 94-02v2, Mitsubishi Electric Research Laboratories, Cambridge Research Center, April 1994.

[Pal88] John Palmer. The NCUBE Family of High-Performance Parallel Computer Systems. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 845–851, January 1988.

[PCI95] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*, June 1995.

[Pie88] Paul Pierce. The NX/2 Operating System. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, January 1988.

[Res93] Cray Research. Cray T3D System Architecture Overview. Technical Report HR-04033, Cray Research Inc, September 1993.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared memory. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 325–336, April 1994.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of 23rd International Symposium on Computer Architecture*, May 1996. To appear.

[RSV87] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication. In *Proceedings of 14th International Symposium on Computer Architecture*, June 1987.

[Sai93] Avtar Saini. An Overview of the Intel Pentium Processor. In *Compcon Spring '93*, pages 60–62, February 1993.

[San91] J. Sandberg. The PRAM Protocol. Technical Report MITL-TR-02-91, Matsushita Information Technology Laboratory, Oct 1991.

[SHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, 27(7):42–50, July 1993.

[SHI92] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-Latency Message Communication Support for the AP1000. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 288–297, May 1992.

[Spe82] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.

[Ste90] W. Richard Stevens. *UNIX Network Programming*. PTR Prentice Hall, 1990.

[TD92] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[Thi92] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, November 1992.

[TLL93] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, April 1993.

[TLL94] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems.

In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1994.

[Tra93] Roger Traylor. *Network Interface Chip (NIC) Architectural Specification, Rev 1.1.* Intel Corporation, December 1993. Document number 316009.

[vEBBV96] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, 1996. To appear.

[WHL92] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager Sharing for Efficient Massive Parallelism. In *Proceedings of the 21st International Conference on Parallel Processing, Vol. II*, pages 251–255, August 1992.

[Wil92] John Wilkes. Hamlyn—An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1992.

[WM89] Larry Wittie and Creve Maples. Merlin: Massively Parallel Heterogeneous Computing. In *Proceedings of the International Conference on Parallel Processing*, pages I–142–I–150, 1989.

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